PostgreSQL 8.3.23 Documentation
by The PostgreSQL Global Development Group
Copyright © 1996-2013 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2013 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.
Postgres95 is Copyright © 1994-5 by the Regents of the University of California.
Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN “AS-IS” BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
# Table of Contents

<table>
<thead>
<tr>
<th>Preface</th>
<th>xliv</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. What is PostgreSQL?</td>
<td>xliv</td>
</tr>
<tr>
<td>2. A Brief History of PostgreSQL</td>
<td>xlv</td>
</tr>
<tr>
<td>2.1. The Berkeley POSTGRES Project</td>
<td>xlv</td>
</tr>
<tr>
<td>2.2. Postgres95</td>
<td>xlv</td>
</tr>
<tr>
<td>2.3. PostgreSQL</td>
<td>xlvi</td>
</tr>
<tr>
<td>3. Conventions</td>
<td>xlvi</td>
</tr>
<tr>
<td>4. Further Information</td>
<td>xlvii</td>
</tr>
<tr>
<td>5. Bug Reporting Guidelines</td>
<td>xlviii</td>
</tr>
<tr>
<td>5.1. Identifying Bugs</td>
<td>xlviii</td>
</tr>
<tr>
<td>5.2. What to report</td>
<td>xlviii</td>
</tr>
<tr>
<td>5.3. Where to report bugs</td>
<td>xlviii</td>
</tr>
</tbody>
</table>

## I. Tutorial

| 1. Getting Started | 1 |
| 1.1. Installation | 1 |
| 1.2. Architectural Fundamentals | 1 |
| 1.3. Creating a Database | 2 |
| 1.4. Accessing a Database | 3 |
| 2. The SQL Language | 6 |
| 2.1. Introduction | 6 |
| 2.2. Concepts | 6 |
| 2.3. Creating a New Table | 6 |
| 2.4. Populating a Table With Rows | 7 |
| 2.5. Querying a Table | 8 |
| 2.6. Joins Between Tables | 10 |
| 2.7. Aggregate Functions | 12 |
| 2.8. Updates | 13 |
| 2.9. Deletions | 14 |
| 3. Advanced Features | 15 |
| 3.1. Introduction | 15 |
| 3.2. Views | 15 |
| 3.3. Foreign Keys | 15 |
| 3.4. Transactions | 16 |
| 3.5. Inheritance | 18 |
| 3.6. Conclusion | 19 |

## II. The SQL Language

| 4. SQL Syntax | 21 |
| 4.1. Lexical Structure | 23 |
| 4.1.1. Identifiers and Key Words | 23 |
| 4.1.2. Constants | 24 |
| 4.1.2.1. String Constants | 24 |
| 4.1.2.2. Dollar-Quoted String Constants | 25 |
| 4.1.2.3. Bit-String Constants | 26 |
| 4.1.2.4. Numeric Constants | 26 |
| 4.1.2.5. Constants of Other Types | 27 |
| 4.1.3. Operators | 28 |
| 4.1.4. Special Characters | 28 |
| 4.1.5. Comments | 29 |
5. Data Definition

5.1. Table Basics .............................................................................................................. 39
5.2. Default Values ........................................................................................................... 40
5.3. Constraints ................................................................................................................ 41
   5.3.1. Check Constraints ........................................................................................ 41
   5.3.2. Not-Null Constraints .................................................................................... 43
   5.3.3. Unique Constraints ....................................................................................... 44
   5.3.4. Primary Keys ................................................................................................... 44
   5.3.5. Foreign Keys ...................................................................................................... 45
5.4. System Columns ......................................................................................................... 48
5.5. Modifying Tables ....................................................................................................... 49
   5.5.1. Adding a Column ........................................................................................ 49
   5.5.2. Removing a Column ..................................................................................... 50
   5.5.3. Adding a Constraint ..................................................................................... 50
   5.5.4. Removing a Constraint ................................................................................ 51
   5.5.5. Changing a Column’s Default Value ............................................................ 51
   5.5.6. Changing a Column’s Data Type ................................................................. 51
   5.5.7. Renaming a Column .................................................................................... 52
   5.5.8. Renaming a Table ....................................................................................... 52
5.6. Privileges .................................................................................................................. 52
5.7. Schemas ..................................................................................................................... 53
   5.7.1. Creating a Schema ...................................................................................... 53
   5.7.2. The Public Schema ...................................................................................... 54
   5.7.3. The Schema Search Path .......................................................................... 54
   5.7.4. Schemas and Privileges ............................................................................. 56
   5.7.5. The System Catalog Schema .................................................................... 56
   5.7.6. Usage Patterns ............................................................................................ 56
   5.7.7. Portability ...................................................................................................... 57
5.8. Inheritance .................................................................................................................. 57
   5.8.1. Caveats .......................................................................................................... 60
5.9. Partitioning ................................................................................................................ 60
   5.9.1. Overview ....................................................................................................... 60
   5.9.2. Implementing Partitioning .......................................................................... 61
   5.9.3. Managing Partitions .................................................................................... 64
   5.9.4. Partitioning and Constraint Exclusion ...................................................... 65
   5.9.5. Alternative Partitioning Methods ............................................................... 66
   5.9.6. Caveats .......................................................................................................... 67
5.10. Other Database Objects .......................................................................................... 67
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.8.4.</td>
<td>Paths</td>
</tr>
<tr>
<td>8.8.5.</td>
<td>Polygons</td>
</tr>
<tr>
<td>8.8.6.</td>
<td>Circles</td>
</tr>
<tr>
<td>8.9.</td>
<td>Network Address Types</td>
</tr>
<tr>
<td>8.10.</td>
<td>Bit String Types</td>
</tr>
<tr>
<td>8.11.</td>
<td>Text Search Types</td>
</tr>
<tr>
<td>8.12.</td>
<td>UUID Type</td>
</tr>
<tr>
<td>8.13.</td>
<td>XML Type</td>
</tr>
<tr>
<td>8.15.</td>
<td>Composite Types</td>
</tr>
<tr>
<td>8.16.</td>
<td>Object Identifier Types</td>
</tr>
<tr>
<td>8.17.</td>
<td>Pseudo-Types</td>
</tr>
<tr>
<td>9.</td>
<td>Functions and Operators</td>
</tr>
<tr>
<td>9.1.</td>
<td>Logical Operators</td>
</tr>
<tr>
<td>9.2.</td>
<td>Comparison Operators</td>
</tr>
<tr>
<td>9.3.</td>
<td>Mathematical Functions and Operators</td>
</tr>
<tr>
<td>9.4.</td>
<td>String Functions and Operators</td>
</tr>
<tr>
<td>9.5.</td>
<td>Binary String Functions and Operators</td>
</tr>
<tr>
<td>9.6.</td>
<td>Bit String Functions and Operators</td>
</tr>
<tr>
<td>9.7.</td>
<td>Pattern Matching</td>
</tr>
<tr>
<td>9.7.1.</td>
<td>LIKE</td>
</tr>
<tr>
<td>9.7.2.</td>
<td>SIMILAR TO Regular Expressions</td>
</tr>
<tr>
<td>9.7.3.</td>
<td>POSIX Regular Expressions</td>
</tr>
<tr>
<td>9.7.3.1.</td>
<td>Regular Expression Details</td>
</tr>
<tr>
<td>9.7.3.2.</td>
<td>Bracket Expressions</td>
</tr>
<tr>
<td>9.7.3.3.</td>
<td>Regular Expression Escapes</td>
</tr>
<tr>
<td>9.7.3.4.</td>
<td>Regular Expression Metasyntax</td>
</tr>
<tr>
<td>9.7.3.5.</td>
<td>Regular Expression Matching Rules</td>
</tr>
<tr>
<td>9.7.3.6.</td>
<td>Limits and Compatibility</td>
</tr>
<tr>
<td>9.7.3.7.</td>
<td>Basic Regular Expressions</td>
</tr>
<tr>
<td>9.8.</td>
<td>Data Type Formatting Functions</td>
</tr>
<tr>
<td>9.9.</td>
<td>Date/Time Functions and Operators</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9.9.1. EXTRACT, date_part</td>
<td>174</td>
</tr>
<tr>
<td>9.9.2. date_trunc</td>
<td>178</td>
</tr>
<tr>
<td>9.9.3. AT TIME ZONE</td>
<td>178</td>
</tr>
<tr>
<td>9.9.4. Current Date/Time</td>
<td>179</td>
</tr>
<tr>
<td>9.9.5. Delaying Execution</td>
<td>181</td>
</tr>
<tr>
<td>9.10. Enum Support Functions</td>
<td>181</td>
</tr>
<tr>
<td>9.11. Geometric Functions and Operators</td>
<td>182</td>
</tr>
<tr>
<td>9.12. Network Address Functions and Operators</td>
<td>186</td>
</tr>
<tr>
<td>9.13. Text Search Functions and Operators</td>
<td>188</td>
</tr>
<tr>
<td>9.14. XML Functions</td>
<td>192</td>
</tr>
<tr>
<td>9.14.1. Producing XML Content</td>
<td>192</td>
</tr>
<tr>
<td>9.14.1.1. xmlcomment</td>
<td>193</td>
</tr>
<tr>
<td>9.14.1.2. xmlconcat</td>
<td>193</td>
</tr>
<tr>
<td>9.14.1.3. xmlelement</td>
<td>194</td>
</tr>
<tr>
<td>9.14.1.4. xmlforest</td>
<td>195</td>
</tr>
<tr>
<td>9.14.1.5. xmlpi</td>
<td>195</td>
</tr>
<tr>
<td>9.14.1.6. xmlroot</td>
<td>196</td>
</tr>
<tr>
<td>9.14.1.7. xmlagg</td>
<td>196</td>
</tr>
<tr>
<td>9.14.1.8. XML Predicates</td>
<td>197</td>
</tr>
<tr>
<td>9.14.2. Processing XML</td>
<td>197</td>
</tr>
<tr>
<td>9.14.3. Mapping Tables to XML</td>
<td>197</td>
</tr>
<tr>
<td>9.15. Sequence Manipulation Functions</td>
<td>201</td>
</tr>
<tr>
<td>9.16. Conditional Expressions</td>
<td>203</td>
</tr>
<tr>
<td>9.16.1. CASE</td>
<td>203</td>
</tr>
<tr>
<td>9.16.2. COALESCE</td>
<td>204</td>
</tr>
<tr>
<td>9.16.3. NULLIF</td>
<td>205</td>
</tr>
<tr>
<td>9.16.4. GREATEST and LEAST</td>
<td>205</td>
</tr>
<tr>
<td>9.17. Array Functions and Operators</td>
<td>205</td>
</tr>
<tr>
<td>9.18. Aggregate Functions</td>
<td>207</td>
</tr>
<tr>
<td>9.19. Subquery Expressions</td>
<td>210</td>
</tr>
<tr>
<td>9.19.1. EXISTS</td>
<td>210</td>
</tr>
<tr>
<td>9.19.2. IN</td>
<td>211</td>
</tr>
<tr>
<td>9.19.3. NOT IN</td>
<td>211</td>
</tr>
<tr>
<td>9.19.4. ANY/SOME</td>
<td>212</td>
</tr>
<tr>
<td>9.19.5. ALL</td>
<td>212</td>
</tr>
<tr>
<td>9.19.6. Row-wise Comparison</td>
<td>213</td>
</tr>
<tr>
<td>9.20. Row and Array Comparisons</td>
<td>213</td>
</tr>
<tr>
<td>9.20.1. IN</td>
<td>213</td>
</tr>
<tr>
<td>9.20.2. NOT IN</td>
<td>214</td>
</tr>
<tr>
<td>9.20.3. ANY/SOME (array)</td>
<td>214</td>
</tr>
<tr>
<td>9.20.4. ALL (array)</td>
<td>214</td>
</tr>
<tr>
<td>9.20.5. Row-wise Comparison</td>
<td>215</td>
</tr>
<tr>
<td>9.21. Set Returning Functions</td>
<td>215</td>
</tr>
<tr>
<td>9.22. System Information Functions</td>
<td>216</td>
</tr>
<tr>
<td>9.23. System Administration Functions</td>
<td>223</td>
</tr>
<tr>
<td>10. Type Conversion</td>
<td>230</td>
</tr>
<tr>
<td>10.1. Overview</td>
<td>230</td>
</tr>
<tr>
<td>10.2. Operators</td>
<td>231</td>
</tr>
<tr>
<td>10.3. Functions</td>
<td>234</td>
</tr>
<tr>
<td>10.4. Value Storage</td>
<td>236</td>
</tr>
<tr>
<td>10.5. UNION, CASE, and Related Constructs</td>
<td>237</td>
</tr>
<tr>
<td>11. Indexes</td>
<td>240</td>
</tr>
</tbody>
</table>
11.1. Introduction .......................................................................................................... 240
11.2. Index Types .......................................................................................................... 241
11.3. Multicolumn Indexes .............................................................................................. 242
11.4. Indexes and ORDER BY .......................................................................................... 243
11.5. Combining Multiple Indexes .................................................................................. 244
11.6. Unique Indexes ...................................................................................................... 245
11.7. Indexes on Expressions .......................................................................................... 245
11.8. Partial Indexes ........................................................................................................ 246
11.9. Operator Classes and Operator Families .............................................................. 248
11.10. Examining Index Usage ....................................................................................... 249
12. Full Text Search ........................................................................................................ 251
  12.1. Introduction .......................................................................................................... 251
    12.1.1. What Is a Document? ...................................................................................... 252
    12.1.2. Basic Text Matching ...................................................................................... 252
    12.1.3. Configurations ............................................................................................... 253
  12.2. Tables and Indexes ............................................................................................... 254
    12.2.1. Searching a Table .......................................................................................... 254
    12.2.2. Creating Indexes ........................................................................................... 255
  12.3. Controlling Text Search ....................................................................................... 256
    12.3.1. Parsing Documents ....................................................................................... 256
    12.3.2. Parsing Queries ............................................................................................. 257
    12.3.3. Ranking Search Results ............................................................................... 258
    12.3.4. Highlighting Results ...................................................................................... 260
  12.4. Additional Features ............................................................................................. 262
    12.4.1. Manipulating Documents .............................................................................. 262
    12.4.2. Manipulating Queries ................................................................................. 263
      12.4.2.1. Query Rewriting ..................................................................................... 264
    12.4.3. Triggers for Automatic Updates ..................................................................... 265
    12.4.4. Gathering Document Statistics ...................................................................... 266
  12.5. Parsers ................................................................................................................ 267
  12.6. Dictionaries .......................................................................................................... 268
    12.6.1. Stop Words ...................................................................................................... 269
    12.6.2. Simple Dictionary .......................................................................................... 270
    12.6.3. Synonym Dictionary ....................................................................................... 271
    12.6.4. Thesaurus Dictionary ..................................................................................... 272
      12.6.4.1. Thesaurus Configuration ......................................................................... 273
      12.6.4.2. Thesaurus Example ................................................................................. 273
    12.6.5. Ispell Dictionary ............................................................................................. 274
    12.6.6. Snowball Dictionary ....................................................................................... 275
  12.7. Configuration Example ....................................................................................... 276
  12.8. Testing and Debugging Text Search ..................................................................... 277
    12.8.1. Configuration Testing ................................................................................... 277
    12.8.2. Parser Testing ............................................................................................... 279
    12.8.3. Dictionary Testing ........................................................................................ 281
  12.9. GiST and GIN Index Types .................................................................................. 281
  12.10. psql Support ....................................................................................................... 283
  12.11. Limitations ......................................................................................................... 286
  12.12. Migration from Pre-8.3 Text Search ................................................................ 286
13. Concurrency Control ................................................................................................. 288
  13.1. Introduction .......................................................................................................... 288
  13.2. Transaction Isolation ........................................................................................... 288
    13.2.1. Read Committed Isolation Level .................................................................. 289
13.2.2. Serializable Isolation Level ................................................................. 290
  13.2.2.1. Serializable Isolation versus True Serializability ......................... 291
13.3. Explicit Locking .................................................................................... 291
  13.3.1. Table-Level Locks ........................................................................... 292
  13.3.2. Row-Level Locks ............................................................................ 294
  13.3.3. Deadlocks ....................................................................................... 295
  13.3.4. Advisory Locks .............................................................................. 296
13.4. Data Consistency Checks at the Application Level .................................. 296
13.5. Locking and Indexes ............................................................................. 297
14. Performance Tips ...................................................................................... 299
  14.1. Using EXPLAIN .................................................................................. 299
  14.2. Statistics Used by the Planner .............................................................. 303
  14.3. Controlling the Planner with Explicit JOIN Clauses .............................. 305
14.4. Populating a Database .......................................................................... 307
  14.4.1. Disable Autocommit ........................................................................ 307
  14.4.2. Use COPY ...................................................................................... 307
  14.4.3. Remove Indexes ............................................................................ 307
  14.4.4. Remove Foreign Key Constraints ...................................................... 308
  14.4.5. Increase maintenance_work_mem ..................................................... 308
  14.4.6. Increase checkpoint_segments ......................................................... 308
  14.4.7. Turn off archive_mode .................................................................... 308
  14.4.8. Run ANALYZE Afterwards ............................................................ 308
  14.4.9. Some Notes About pg_dump .......................................................... 309
15. Installation Instructions ............................................................................. 312
  15.1. Short Version .................................................................................... 312
  15.2. Requirements ................................................................................. 312
  15.3. Getting The Source .......................................................................... 314
  15.4. Upgrading ....................................................................................... 314
  15.5. Installation Procedure ....................................................................... 315
  15.6. Post-Installation Setup ...................................................................... 323
  15.6.1. Shared Libraries ........................................................................... 323
  15.6.2. Environment Variables ................................................................. 324
  15.7. Supported Platforms ........................................................................ 324
16. Installation on Windows ............................................................................ 326
    16.1.1. Requirements ............................................................................ 326
    16.1.2. Building .................................................................................... 327
    16.1.3. Cleaning and installing ............................................................... 328
    16.1.4. Running the regression tests ...................................................... 328
    16.1.5. Building the documentation ...................................................... 329
  16.2. Building libpq with Visual C++ or Borland C++ .................................. 329
    16.2.1. Generated files ......................................................................... 330
17. Operating System Environment ............................................................... 331
  17.1. The PostgreSQL User Account .......................................................... 331
  17.2. Creating a Database Cluster ............................................................... 331
    17.2.1. Network File Systems ............................................................... 332
  17.3. Starting the Database Server .............................................................. 332
    17.3.1. Server Start-up Failures ............................................................. 334
    17.3.2. Client Connection Problems ...................................................... 334
  17.4. Managing Kernel Resources .............................................................. 335
IV. Client Interfaces ....................................................................................................................... 474

30. libpq - C Library ................................................................................................................... 476
  30.1. Database Connection Control Functions ................................................................. 476
  30.2. Connection Status Functions .................................................................................... 482
  30.3. Command Execution Functions ............................................................................... 485
    30.3.1. Main Functions ......................................................................................................... 486
    30.3.2. Retrieving Query Result Information .................................................................... 492
    30.3.3. Retrieving Result Information for Other Commands ........................................... 496
    30.3.4. Escaping Strings for Inclusion in SQL Commands ............................................... 497
    30.3.5. Escaping Binary Strings for Inclusion in SQL Commands .................................... 498
  30.4. Asynchronous Command Processing ........................................................................ 499
  30.5. Cancelling Queries in Progress ................................................................................... 503
  30.6. The Fast-Path Interface .............................................................................................. 504
  30.7. Asynchronous Notification .......................................................................................... 505
  30.8. Functions Associated with the COPY Command ....................................................... 506
    30.8.1. Functions for Sending COPY Data ................................................................. 507
    30.8.2. Functions for Receiving COPY Data ............................................................ 508
    30.8.3. Obsolete Functions for COPY ............................................................................ 508
  30.9. Control Functions ......................................................................................................... 510
  30.10. Miscellaneous Functions ............................................................................................. 511
  30.11. Notice Processing ........................................................................................................ 512
  30.12. Environment Variables .............................................................................................. 513
  30.13. The Password File ........................................................................................................ 515
  30.14. The Connection Service File ..................................................................................... 515
  30.15. LDAP Lookup of Connection Parameters ............................................................. 515
  30.16. SSL Support .................................................................................................................. 516
  30.17. Behavior in Threaded Programs ............................................................................... 517
  30.18. Building libpq Programs ............................................................................................ 518
  30.19. Example Programs ..................................................................................................... 519

31. Large Objects ....................................................................................................................... 528
  31.1. Introduction .................................................................................................................... 528
  31.2. Implementation Features ............................................................................................. 528
  31.3. Client Interfaces ............................................................................................................. 528
    31.3.1. Creating a Large Object ......................................................................................... 528
    31.3.2. Importing a Large Object ....................................................................................... 529
    31.3.3. Exporting a Large Object ....................................................................................... 529
    31.3.4. Opening an Existing Large Object ......................................................................... 529
    31.3.5. Writing Data to a Large Object ............................................................................ 530
    31.3.6. Reading Data from a Large Object ........................................................................ 530
    31.3.7. Seeking in a Large Object ..................................................................................... 530
    31.3.8. Obtaining the Seek Position of a Large Object .................................................... 530
    31.3.9. Truncating a Large Object ..................................................................................... 531
    31.3.10. Closing a Large Object Descriptor ...................................................................... 531
    31.3.11. Removing a Large Object .................................................................................... 531
  31.4. Server-Side Functions ..................................................................................................... 531
  31.5. Example Program ........................................................................................................... 532

32. ECPG - Embedded SQL in C ............................................................................................... 538
  32.1. The Concept .................................................................................................................... 538
  32.2. Connecting to the Database Server .............................................................................. 538
  32.3. Closing a Connection .................................................................................................. 539
  32.4. Running SQL Commands ............................................................................................. 540
V. Server Programming ................................................................................................................. 628
34. Extending SQL.................................................................................................................. 630
34.1. How Extensibility Works...................................................................................... 630
34.2. The PostgreSQL Type System........................................................................... 630
  34.2.1. Base Types ........................................................................................................... 630
  34.2.2. Composite Types .............................................................................................. 630
  34.2.3. Domains ............................................................................................................. 631
  34.2.4. Pseudo-Types ................................................................................................... 631
  34.2.5. Polymorphic Types ......................................................................................... 631
34.3. User-Defined Functions................................................................................... 632
34.4. Query Language (SQL) Functions .................................................................. 632
  34.4.1. SQL Functions on Base Types ........................................................................ 633
  34.4.2. SQL Functions on Composite Types ............................................................ 634
  34.4.3. Functions with Output Parameters ............................................................... 637
  34.4.4. SQL Functions as Table Sources ................................................................. 638
  34.4.5. SQL Functions Returning Sets ...................................................................... 639
  34.4.6. Polymorphic SQL Functions ........................................................................ 640
34.5. Function Overloading ....................................................................................... 641
34.6. Function Volatility Categories ........................................................................ 642
34.7. Procedural Language Functions ........................................................................ 643
34.8. Internal Functions ................................................................................................. 643
34.9. C-Language Functions........................................................................................ 644
  34.9.1. Dynamic Loading ............................................................................................. 644
<p>| 34.9.2. Base Types in C-Language Functions | 645 |
| 34.9.3. Version 0 Calling Conventions | 648 |
| 34.9.4. Version 1 Calling Conventions | 650 |
| 34.9.5. Writing Code | 652 |
| 34.9.6. Compiling and Linking Dynamically-Loaded Functions | 653 |
| 34.9.7. Extension Building Infrastructure | 655 |
| 34.9.8. Composite-Type Arguments | 657 |
| 34.9.9. Returning Rows (Composite Types) | 659 |
| 34.9.10. Returning Sets | 661 |
| 34.9.11. Polymorphic Arguments and Return Types | 665 |
| 34.9.12. Shared Memory and LWLocks | 666 |
| 34.10. User-Defined Aggregates | 667 |
| 34.11. User-Defined Types | 669 |
| 34.12. User-Defined Operators | 673 |
| 34.13. Operator Optimization Information | 673 |
| 34.13.1. COMMUTATOR | 674 |
| 34.13.2. NEGATOR | 674 |
| 34.13.3. RESTRICT | 675 |
| 34.13.4. JOIN | 676 |
| 34.13.5. HASHES | 676 |
| 34.13.6. MERGES | 677 |
| 34.14. Interfacing Extensions To Indexes | 678 |
| 34.14.1. Index Methods and Operator Classes | 678 |
| 34.14.2. Index Method Strategies | 678 |
| 34.14.3. Index Method Support Routines | 680 |
| 34.14.4. An Example | 681 |
| 34.14.5. Operator Classes and Operator Families | 684 |
| 34.14.6. System Dependencies on Operator Classes | 686 |
| 34.14.7. Special Features of Operator Classes | 687 |
| 35. Triggers | 689 |
| 35.1. Overview of Trigger Behavior | 689 |
| 35.2. Visibility of Data Changes | 690 |
| 35.3. Writing Trigger Functions in C | 691 |
| 35.4. A Complete Example | 693 |
| 36. The Rule System | 697 |
| 36.1. The Query Tree | 697 |
| 36.2. Views and the Rule System | 699 |
| 36.2.1. How SELECT Rules Work | 699 |
| 36.2.2. View Rules in Non-SELECT Statements | 704 |
| 36.2.3. The Power of Views in PostgreSQL | 705 |
| 36.2.4. Updating a View | 705 |
| 36.3. Rules on INSERT, UPDATE, and DELETE | 705 |
| 36.3.1. How Update Rules Work | 706 |
| 36.3.1.1. A First Rule Step by Step | 707 |
| 36.3.2. Cooperation with Views | 710 |
| 36.4. Rules and Privileges | 716 |
| 36.5. Rules and Command Status | 717 |
| 36.6. Rules versus Triggers | 717 |
| 37. Procedural Languages | 721 |
| 37.1. Installing Procedural Languages | 721 |
| 38. PL/pgSQL - SQL Procedural Language | 723 |
| 38.1. Overview | 723 |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIII. Appendixes</td>
<td>1415</td>
</tr>
<tr>
<td>V. How the Planner Uses Statistics</td>
<td>1393</td>
</tr>
<tr>
<td>51. GiST Indexes</td>
<td>1399</td>
</tr>
<tr>
<td>51.1. Introduction</td>
<td>1393</td>
</tr>
<tr>
<td>51.2. Extensibility</td>
<td>1393</td>
</tr>
<tr>
<td>51.3. Implementation</td>
<td>1393</td>
</tr>
<tr>
<td>51.4. Examples</td>
<td>1394</td>
</tr>
<tr>
<td>51.5. Crash Recovery</td>
<td>1395</td>
</tr>
<tr>
<td>52. GIN Indexes</td>
<td>1396</td>
</tr>
<tr>
<td>52.1. Introduction</td>
<td>1396</td>
</tr>
<tr>
<td>52.2. Extensibility</td>
<td>1396</td>
</tr>
<tr>
<td>52.3. Implementation</td>
<td>1397</td>
</tr>
<tr>
<td>52.4. GIN tips and tricks</td>
<td>1397</td>
</tr>
<tr>
<td>52.5. Limitations</td>
<td>1397</td>
</tr>
<tr>
<td>52.6. Examples</td>
<td>1398</td>
</tr>
<tr>
<td>53. Database Physical Storage</td>
<td>1399</td>
</tr>
<tr>
<td>53.1. Database File Layout</td>
<td>1400</td>
</tr>
<tr>
<td>53.2. TOAST</td>
<td>1400</td>
</tr>
<tr>
<td>53.3. Database Page Layout</td>
<td>1402</td>
</tr>
<tr>
<td>54. BKI Backend Interface</td>
<td>1406</td>
</tr>
<tr>
<td>54.1. BKI File Format</td>
<td>1406</td>
</tr>
<tr>
<td>54.2. BKI Commands</td>
<td>1406</td>
</tr>
<tr>
<td>54.3. Structure of the Bootstrap BKI File</td>
<td>1407</td>
</tr>
<tr>
<td>54.4. Example</td>
<td>1408</td>
</tr>
<tr>
<td>55. How the Planner Uses Statistics</td>
<td>1409</td>
</tr>
<tr>
<td>55.1. Row Estimation Examples</td>
<td>1409</td>
</tr>
<tr>
<td>E. Release Notes</td>
<td>1473</td>
</tr>
<tr>
<td>E.1. Release 8.3.23</td>
<td>1473</td>
</tr>
<tr>
<td>E.1.1. Migration to Version 8.3.23</td>
<td>1473</td>
</tr>
<tr>
<td>E.1.2. Changes</td>
<td>1473</td>
</tr>
<tr>
<td>E.2. Release 8.3.22</td>
<td>1474</td>
</tr>
<tr>
<td>E.2.1. Migration to Version 8.3.22</td>
<td>1474</td>
</tr>
<tr>
<td>E.2.2. Changes</td>
<td>1474</td>
</tr>
<tr>
<td>E.3. Release 8.3.21</td>
<td>1476</td>
</tr>
<tr>
<td>E.3.1. Migration to Version 8.3.21</td>
<td>1476</td>
</tr>
<tr>
<td>E.3.2. Changes</td>
<td>1476</td>
</tr>
<tr>
<td>E.4. Release 8.3.20</td>
<td>1477</td>
</tr>
<tr>
<td>E.4.1. Migration to Version 8.3.20</td>
<td>1477</td>
</tr>
<tr>
<td>E.4.2. Changes</td>
<td>1477</td>
</tr>
<tr>
<td>E.5. Release 8.3.19</td>
<td>1479</td>
</tr>
<tr>
<td>E.5.1. Migration to Version 8.3.19</td>
<td>1479</td>
</tr>
<tr>
<td>E.5.2. Changes</td>
<td>1479</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>E.60.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.61.</td>
<td>Release 8.1.11</td>
</tr>
<tr>
<td>E.61.1.</td>
<td>Migration to Version 8.1.11</td>
</tr>
<tr>
<td>E.61.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.62.</td>
<td>Release 8.1.10</td>
</tr>
<tr>
<td>E.62.1.</td>
<td>Migration to Version 8.1.10</td>
</tr>
<tr>
<td>E.62.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.63.</td>
<td>Release 8.1.9</td>
</tr>
<tr>
<td>E.63.1.</td>
<td>Migration to Version 8.1.9</td>
</tr>
<tr>
<td>E.63.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.64.</td>
<td>Release 8.1.8</td>
</tr>
<tr>
<td>E.64.1.</td>
<td>Migration to Version 8.1.8</td>
</tr>
<tr>
<td>E.64.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.65.</td>
<td>Release 8.1.7</td>
</tr>
<tr>
<td>E.65.1.</td>
<td>Migration to Version 8.1.7</td>
</tr>
<tr>
<td>E.65.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.66.</td>
<td>Release 8.1.6</td>
</tr>
<tr>
<td>E.66.1.</td>
<td>Migration to Version 8.1.6</td>
</tr>
<tr>
<td>E.66.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.67.</td>
<td>Release 8.1.5</td>
</tr>
<tr>
<td>E.67.1.</td>
<td>Migration to Version 8.1.5</td>
</tr>
<tr>
<td>E.67.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.68.</td>
<td>Release 8.1.4</td>
</tr>
<tr>
<td>E.68.1.</td>
<td>Migration to Version 8.1.4</td>
</tr>
<tr>
<td>E.68.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.69.</td>
<td>Release 8.1.3</td>
</tr>
<tr>
<td>E.69.1.</td>
<td>Migration to Version 8.1.3</td>
</tr>
<tr>
<td>E.69.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.70.</td>
<td>Release 8.1.2</td>
</tr>
<tr>
<td>E.70.1.</td>
<td>Migration to Version 8.1.2</td>
</tr>
<tr>
<td>E.70.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.71.</td>
<td>Release 8.1.1</td>
</tr>
<tr>
<td>E.71.1.</td>
<td>Migration to Version 8.1.1</td>
</tr>
<tr>
<td>E.71.2.</td>
<td>Changes</td>
</tr>
<tr>
<td>E.72.</td>
<td>Release 8.1</td>
</tr>
<tr>
<td>E.72.1.</td>
<td>Overview</td>
</tr>
<tr>
<td>E.72.2.</td>
<td>Migration to Version 8.1</td>
</tr>
<tr>
<td>E.72.3.</td>
<td>Additional Changes</td>
</tr>
<tr>
<td>E.72.3.1.</td>
<td>Performance Improvements</td>
</tr>
<tr>
<td>E.72.3.2.</td>
<td>Server Changes</td>
</tr>
<tr>
<td>E.72.3.3.</td>
<td>Query Changes</td>
</tr>
<tr>
<td>E.72.3.4.</td>
<td>Object Manipulation Changes</td>
</tr>
<tr>
<td>E.72.3.5.</td>
<td>Utility Command Changes</td>
</tr>
<tr>
<td>E.72.3.6.</td>
<td>Data Type and Function Changes</td>
</tr>
<tr>
<td>E.72.3.7.</td>
<td>Encoding and Locale Changes</td>
</tr>
<tr>
<td>E.72.3.8.</td>
<td>General Server-Side Language Changes</td>
</tr>
<tr>
<td>E.72.3.9.</td>
<td>PL/pgSQL Server-Side Language Changes</td>
</tr>
<tr>
<td>E.72.3.10.</td>
<td>PL/Perl Server-Side Language Changes</td>
</tr>
<tr>
<td>E.72.3.11.</td>
<td>psql Changes</td>
</tr>
<tr>
<td>E.72.3.12.</td>
<td>pg_dump Changes</td>
</tr>
<tr>
<td>E.72.3.13.</td>
<td>libpq Changes</td>
</tr>
<tr>
<td>E.72.3.14.</td>
<td>Source Code Changes</td>
</tr>
</tbody>
</table>
E.132. Release 7.3.20 ........................................................................................................ 1701
  E.132.1. Migration to Version 7.3.20................................................................. 1701
  E.132.2. Changes ............................................................................................... 1701
E.133. Release 7.3.19 ................................................................................................. 1701
  E.133.1. Migration to Version 7.3.19................................................................. 1702
  E.133.2. Changes ............................................................................................... 1702
E.134. Release 7.3.18 ................................................................................................. 1702
  E.134.1. Migration to Version 7.3.18................................................................. 1702
  E.134.2. Changes ............................................................................................... 1702
E.135. Release 7.3.17 ................................................................................................. 1703
  E.135.1. Migration to Version 7.3.17................................................................. 1703
  E.135.2. Changes ............................................................................................... 1703
E.136. Release 7.3.16 ................................................................................................. 1703
  E.136.1. Migration to Version 7.3.16................................................................. 1703
  E.136.2. Changes ............................................................................................... 1703
E.137. Release 7.3.15 ................................................................................................. 1704
  E.137.1. Migration to Version 7.3.15................................................................. 1704
  E.137.2. Changes ............................................................................................... 1704
E.138. Release 7.3.14 ................................................................................................. 1705
  E.138.1. Migration to Version 7.3.14................................................................. 1705
  E.138.2. Changes ............................................................................................... 1705
E.139. Release 7.3.13 ................................................................................................. 1705
  E.139.1. Migration to Version 7.3.13................................................................. 1706
  E.139.2. Changes ............................................................................................... 1706
E.140. Release 7.3.12 ................................................................................................. 1706
  E.140.1. Migration to Version 7.3.12................................................................. 1706
  E.140.2. Changes ............................................................................................... 1707
E.141. Release 7.3.11 ................................................................................................. 1707
  E.141.1. Migration to Version 7.3.11................................................................. 1707
  E.141.2. Changes ............................................................................................... 1707
E.142. Release 7.3.10 ................................................................................................. 1708
  E.142.1. Migration to Version 7.3.10................................................................. 1708
  E.142.2. Changes ............................................................................................... 1709
E.143. Release 7.3.9 ................................................................................................. 1709
  E.143.1. Migration to Version 7.3.9................................................................. 1709
  E.143.2. Changes ............................................................................................... 1710
E.144. Release 7.3.8 ................................................................................................. 1710
  E.144.1. Migration to Version 7.3.8................................................................. 1710
  E.144.2. Changes ............................................................................................... 1710
E.145. Release 7.3.7 ................................................................................................. 1711
  E.145.1. Migration to Version 7.3.7................................................................. 1711
  E.145.2. Changes ............................................................................................... 1711
E.146. Release 7.3.6 ................................................................................................. 1711
  E.146.1. Migration to Version 7.3.6................................................................. 1711
  E.146.2. Changes ............................................................................................... 1711
E.147. Release 7.3.5 ................................................................................................. 1712
  E.147.1. Migration to Version 7.3.5................................................................. 1712
  E.147.2. Changes ............................................................................................... 1712
E.148. Release 7.3.4 ................................................................................................. 1713
  E.148.1. Migration to Version 7.3.4................................................................. 1713
  E.148.2. Changes ............................................................................................... 1713
E.149. Release 7.3.3 ................................................................................................. 1714
F.19. pg_buffercache .................................................................................................... 1868
F.19.1. The pg_buffercache view ............................................................................. 1869
F.19.2. Sample output ............................................................................................ 1869
F.19.3. Authors ....................................................................................................... 1870
F.20. pgcrypto ............................................................................................................ 1870
F.20.1. General hashing functions ......................................................................... 1870
F.20.1.1. crypt() ........................................................................................................ 1871
F.20.1.2. hmac() .................................................................................................... 1872
F.20.2. Password hashing functions ..................................................................... 1871
F.20.2.1. crypt() ........................................................................................................ 1871
F.20.2.2. gen_salt() ................................................................................................ 1872
F.20.3. PGP encryption functions ........................................................................ 1873
F.20.3.1. pgp_sym_encrypt() ................................................................................ 1874
F.20.3.2. pgp_sym_decrypt() ................................................................................ 1874
F.20.3.3. pgp_pub_encrypt() ................................................................................ 1874
F.20.3.4. pgp_pub_decrypt() ................................................................................ 1874
F.20.3.5. pgp_key_id() .......................................................................................... 1874
F.20.3.6. armor(), dearmor() ................................................................................. 1875
F.20.3.7. Options for PGP functions .................................................................... 1875
F.20.3.7.1. cipher-algo ......................................................................................... 1875
F.20.3.7.2. compress-algo .................................................................................. 1875
F.20.3.7.3. compress-level .................................................................................. 1876
F.20.3.7.4. convert-crlf ....................................................................................... 1876
F.20.3.7.5. disable-mdc ....................................................................................... 1876
F.20.3.7.6. enable-session-key ............................................................................ 1876
F.20.3.7.7. s2k-mode ............................................................................................ 1876
F.20.3.7.8. s2k-digest-algo .................................................................................. 1877
F.20.3.7.9. s2k-cipher-algo ................................................................................ 1877
F.20.3.7.10. unicode-mode ................................................................................ 1877
F.20.3.8. Generating PGP keys with GnuPG ......................................................... 1877
F.20.3.9. Limitations of PGP code ....................................................................... 1878
F.20.4. Raw encryption functions ......................................................................... 1878
F.20.5. Random-data functions .......................................................................... 1879
F.20.6. Notes .......................................................................................................... 1879
F.20.6.1. Configuration ......................................................................................... 1879
F.20.6.2. NULL handling ...................................................................................... 1880
F.20.6.3. Security limitations ............................................................................... 1880
F.20.6.4. Useful limitations .................................................................................. 1880
F.20.6.5. Technical references ............................................................................ 1881
F.20.7. Author ......................................................................................................... 1881
F.21. pg_freespacemap .............................................................................................. 1882
F.21.1. The pg_freespacemap views ..................................................................... 1882
F.21.2. Sample output ............................................................................................ 1883
F.21.3. Author ....................................................................................................... 1884
F.22. pgrowlocks ....................................................................................................... 1884
F.22.1. Overview ..................................................................................................... 1884
F.22.2. Sample output ............................................................................................ 1885
F.22.3. Author ....................................................................................................... 1885
F.23. pg_standby ......................................................................................................... 1885
F.23.1. Usage .......................................................................................................... 1886
F.23.2. Examples ..................................................................................................... 1888
F.23.3. Supported server versions ........................................................................ 1889
Preface

This book is the official documentation of PostgreSQL. It is being written by the PostgreSQL developers and other volunteers in parallel to the development of the PostgreSQL software. It describes all the functionality that the current version of PostgreSQL officially supports.

To make the large amount of information about PostgreSQL manageable, this book has been organized in several parts. Each part is targeted at a different class of users, or at users in different stages of their PostgreSQL experience:

• Part I is an informal introduction for new users.
• Part II documents the SQL query language environment, including data types and functions, as well as user-level performance tuning. Every PostgreSQL user should read this.
• Part III describes the installation and administration of the server. Everyone who runs a PostgreSQL server, be it for private use or for others, should read this part.
• Part IV describes the programming interfaces for PostgreSQL client programs.
• Part V contains information for advanced users about the extensibility capabilities of the server. Topics are, for instance, user-defined data types and functions.
• Part VI contains reference information about SQL commands, client and server programs. This part supports the other parts with structured information sorted by command or program.
• Part VII contains assorted information that might be of use to PostgreSQL developers.

1. What is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2\(^1\), developed at the University of California at Berkeley Computer Science Department. POSTGRES pioneered many concepts that only became available in some commercial database systems much later.

PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features:

• complex queries
• foreign keys
• triggers
• views
• transactional integrity
• multiversion concurrency control

Also, PostgreSQL can be extended by the user in many ways, for example by adding new

• data types
• functions
• operators
• aggregate functions
• index methods

• procedural languages

And because of the liberal license, PostgreSQL can be used, modified, and distributed by everyone free of charge for any purpose, be it private, commercial, or academic.

2. A Brief History of PostgreSQL

The object-relational database management system now known as PostgreSQL is derived from the POSTGRES package written at the University of California at Berkeley. With over a decade of development behind it, PostgreSQL is now the most advanced open-source database available anywhere.

2.1. The Berkeley POSTGRES Project

The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc. The implementation of POSTGRES began in 1986. The initial concepts for the system were presented in *The design of POSTGRES*, and the definition of the initial data model appeared in *The POSTGRES data model*. The design of the rule system at that time was described in *The design of the POSTGRES rules system*. The rationale and architecture of the storage manager were detailed in *The design of the POSTGRES storage system*.

POSTGRES has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. Version 1, described in *The implementation of POSTGRES*, was released to a few external users in June 1989. In response to a critique of the first rule system (*A commentary on the POSTGRES rules system*), the rule system was redesigned (*On Rules, Procedures, Caching and Views in Database Systems*), and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rule system. For the most part, subsequent releases until Postgres95 (see below) focused on portability and reliability.

POSTGRES has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. POSTGRES has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (later merged into Informix², which is now owned by IBM³) picked up the code and commercialized it. In late 1992, POSTGRES became the primary data manager for the Sequoia 2000 scientific computing project⁴.

The size of the external user community nearly doubled during 1993. It became increasingly obvious that maintenance of the prototype code and support was taking up large amounts of time that should have been devoted to database research. In an effort to reduce this support burden, the Berkeley POSTGRES project officially ended with Version 4.2.

---

² http://www.informix.com/
³ http://www.ibm.com/
⁴ http://meteora.ucsd.edu/s2k/s2k_home.html
2.2. Postgres95

In 1994, Andrew Yu and Jolly Chen added a SQL language interpreter to POSTGRES. Under a new name, Postgres95 was subsequently released to the web to find its own way in the world as an open-source descendant of the original POSTGRES Berkeley code.

Postgres95 code was completely ANSI C and trimmed in size by 25%. Many internal changes improved performance and maintainability. Postgres95 release 1.0.x ran about 30-50% faster on the Wisconsin Benchmark compared to POSTGRES, Version 4.2. Apart from bug fixes, the following were the major enhancements:

- The query language PostQUEL was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregate functions were re-implemented. Support for the GROUP BY query clause was also added.

- A new program (psql) was provided for interactive SQL queries, which used GNU Readline. This largely superseded the old monitor program.

- A new front-end library, libpgtcl, supported Tcl-based clients. A sample shell, pgtclsh, provided new Tcl commands to interface Tcl programs with the Postgres95 server.

- The large-object interface was overhauled. The inversion large objects were the only mechanism for storing large objects. (The inversion file system was removed.)

- The instance-level rule system was removed. Rules were still available as rewrite rules.

- A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code

- GNU make (instead of BSD make) was used for the build. Also, Postgres95 could be compiled with an unpatched GCC (data alignment of doubles was fixed).

2.3. PostgreSQL

By 1996, it became clear that the name “Postgres95” would not stand the test of time. We chose a new name, PostgreSQL, to reflect the relationship between the original POSTGRES and the more recent versions with SQL capability. At the same time, we set the version numbering to start at 6.0, putting the numbers back into the sequence originally begun by the Berkeley POSTGRES project.

Many people continue to refer to PostgreSQL as “Postgres” (now rarely in all capital letters) because of tradition or because it is easier to pronounce. This usage is widely accepted as a nickname or alias.

The emphasis during development of Postgres95 was on identifying and understanding existing problems in the server code. With PostgreSQL, the emphasis has shifted to augmenting features and capabilities, although work continues in all areas.

Details about what has happened in PostgreSQL since then can be found in Appendix E.

3. Conventions

This book uses the following typographical conventions to mark certain portions of text: new terms, foreign phrases, and other important passages are emphasized in italics. Everything that represents
input or output of the computer, in particular commands, program code, and screen output, is shown in a monospaced font (example). Within such passages, italics (example) indicate placeholders; you must insert an actual value instead of the placeholder. On occasion, parts of program code are emphasized in bold face (example), if they have been added or changed since the preceding example.

The following conventions are used in the synopsis of a command: brackets ([ and ]) indicate optional parts. (In the synopsis of a Tcl command, question marks (?) are used instead, as is usual in Tcl.) Braces ({ and } ) and vertical lines (|) indicate that you must choose one alternative. Dots ( . . . ) mean that the preceding element can be repeated.

Where it enhances the clarity, SQL commands are preceded by the prompt =>. and shell commands are preceded by the prompt $. Normally, prompts are not shown, though.

An administrator is generally a person who is in charge of installing and running the server. A user could be anyone who is using, or wants to use, any part of the PostgreSQL system. These terms should not be interpreted too narrowly; this book does not have fixed presumptions about system administration procedures.

4. Further Information

Besides the documentation, that is, this book, there are other resources about PostgreSQL:

FAQs
The FAQ list contains continuously updated answers to frequently asked questions.

Web Site
The PostgreSQL web site carries details on the latest release and other information to make your work or play with PostgreSQL more productive.

Mailing Lists
The mailing lists are a good place to have your questions answered, to share experiences with other users, and to contact the developers. Consult the PostgreSQL web site for details.

Yourself!
PostgreSQL is an open-source project. As such, it depends on the user community for ongoing support. As you begin to use PostgreSQL, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. Read the mailing lists and answer questions. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute them.

5. Bug Reporting Guidelines

When you find a bug in PostgreSQL we want to hear about it. Your bug reports play an important part in making PostgreSQL more reliable because even the utmost care cannot guarantee that every part of PostgreSQL will work on every platform under every circumstance.

The following suggestions are intended to assist you in forming bug reports that can be handled in an effective fashion. No one is required to follow them but doing so tends to be to everyone’s advantage.

We cannot promise to fix every bug right away. If the bug is obvious, critical, or affects a lot of users, chances are good that someone will look into it. It could also happen that we tell you to update to a newer version to see if the bug happens there. Or we might decide that the bug cannot be fixed before some major rewrite we might be planning is done. Or perhaps it is simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

5.1. Identifying Bugs

Before you report a bug, please read and re-read the documentation to verify that you can really do whatever it is you are trying. If it is not clear from the documentation whether you can do something or not, please report that too; it is a bug in the documentation. If it turns out that a program does something different from what the documentation says, that is a bug. That might include, but is not limited to, the following circumstances:

- A program terminates with a fatal signal or an operating system error message that would point to a problem in the program. (A counterexample might be a “disk full” message, since you have to fix that yourself.)
- A program produces the wrong output for any given input.
- A program refuses to accept valid input (as defined in the documentation).
- A program accepts invalid input without a notice or error message. But keep in mind that your idea of invalid input might be our idea of an extension or compatibility with traditional practice.
- PostgreSQL fails to compile, build, or install according to the instructions on supported platforms.

Here “program” refers to any executable, not only the backend server.

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications. Failing to comply to the SQL standard is not necessarily a bug either, unless compliance for the specific feature is explicitly claimed.

5.2. What to report

The most important thing to remember about bug reporting is to state all the facts and only facts. Do not speculate what you think went wrong, what “it seemed to do”, or which part of the program has a fault. If you are not familiar with the implementation you would probably guess wrong and not help us a bit. And even if you are, educated explanations are a great supplement to but no substitute for facts. If we are going to fix the bug we still have to see it happen for ourselves first. Reporting the bare facts is relatively straightforward (you can probably copy and paste them from the screen) but all too often important details are left out because someone thought it does not matter or the report would be understood anyway.

The following items should be contained in every bug report:

- The exact sequence of steps from *program start-up* necessary to reproduce the problem. This should be self-contained; it is not enough to send in a bare `SELECT` statement without the preceding `CREATE TABLE` and `INSERT` statements, if the output should depend on the data in the tables.
We do not have the time to reverse-engineer your database schema, and if we are supposed to make up our own data we would probably miss the problem.

The best format for a test case for SQL-related problems is a file that can be run through the psql frontend that shows the problem. (Be sure to not have anything in your ~/.psqlrc start-up file.) An easy start at this file is to use pg_dump to dump out the table declarations and data needed to set the scene, then add the problem query. You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproducible, we will find it either way.

If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We will probably not set up a web server to reproduce your problem. In any case remember to provide the exact input files; do not guess that the problem happens for “large files” or “midsize databases”, etc. since this information is too inexact to be of use.

- The output you got. Please do not say that it “didn’t work” or “crashed”. If there is an error message, show it, even if you do not understand it. If the program terminates with an operating system error, say which. If nothing at all happens, say so. Even if the result of your test case is a program crash or otherwise obvious it might not happen on our platform. The easiest thing is to copy the output from the terminal, if possible.

Note: If you are reporting an error message, please obtain the most verbose form of the message. In psql, say \set VERBOSITY verbose beforehand. If you are extracting the message from the server log, set the run-time parameter log_error_verbosity to verbose so that all details are logged.

Note: In case of fatal errors, the error message reported by the client might not contain all the information available. Please also look at the log output of the database server. If you do not keep your server’s log output, this would be a good time to start doing so.

- The output you expected is very important to state. If you just write “This command gives me that output.” or “This is not what I expected.”, we might run it ourselves, scan the output, and think it looks OK and is exactly what we expected. We should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that “This is not what SQL says/Oracle does.” Digging out the correct behavior from SQL is not a fun undertaking, nor do we all know how all the other relational databases out there behave. (If your problem is a program crash, you can obviously omit this item.)

- Any command line options and other start-up options, including any relevant environment variables or configuration files that you changed from the default. Again, please provide exact information. If you are using a prepackaged distribution that starts the database server at boot time, you should try to find out how that is done.

- Anything you did at all differently from the installation instructions.

- The PostgreSQL version. You can run the command SELECT version(); to find out the version of the server you are connected to. Most executable programs also support a --version option; at least postgres --version and psql --version should work. If the function or the options do not exist then your version is more than old enough to warrant an upgrade. If you run a prepackaged version, such as RPMs, say so, including any subversion the package might have. If you are talking about a Git snapshot, mention that, including the commit hash.
If your version is older than 8.3.23 we will almost certainly tell you to upgrade. There are many bug fixes and improvements in each new release, so it is quite possible that a bug you have encountered in an older release of PostgreSQL has already been fixed. We can only provide limited support for sites using older releases of PostgreSQL; if you require more than we can provide, consider acquiring a commercial support contract.

- Platform information. This includes the kernel name and version, C library, processor, memory information, and so on. In most cases it is sufficient to report the vendor and version, but do not assume everyone knows what exactly “Debian” contains or that everyone runs on Pentiums. If you have installation problems then information about the toolchain on your machine (compiler, make, and so on) is also necessary.

Do not be afraid if your bug report becomes rather lengthy. That is a fact of life. It is better to report everything the first time than us having to squeeze the facts out of you. On the other hand, if your input files are huge, it is fair to ask first whether somebody is interested in looking into it. Here is an article\(^6\) that outlines some more tips on reporting bugs.

Do not spend all your time to figure out which changes in the input make the problem go away. This will probably not help solving it. If it turns out that the bug cannot be fixed right away, you will still have time to find and share your work-around. Also, once again, do not waste your time guessing why the bug exists. We will find that out soon enough.

When writing a bug report, please avoid confusing terminology. The software package in total is called “PostgreSQL”, sometimes “Postgres” for short. If you are specifically talking about the backend server, mention that, do not just say “PostgreSQL crashes”. A crash of a single backend server process is quite different from crash of the parent “postgres” process; please don’t say “the server crashed” when you mean a single backend process went down, nor vice versa. Also, client programs such as the interactive frontend “psql” are completely separate from the backend. Please try to be specific about whether the problem is on the client or server side.

### 5.3. Where to report bugs

In general, send bug reports to the bug report mailing list at <pgsql-bugs@postgresql.org>. You are requested to use a descriptive subject for your email message, perhaps parts of the error message.

Another method is to fill in the bug report web-form available at the project’s web site\(^7\). Entering a bug report this way causes it to be mailed to the <pgsql-bugs@postgresql.org> mailing list.

If your bug report has security implications and you’d prefer that it not become immediately visible in public archives, don’t send it to psql-bugs. Security issues can be reported privately to <security@postgresql.org>.

Do not send bug reports to any of the user mailing lists, such as <pgsql-sql@postgresql.org> or <pgsql-general@postgresql.org>. These mailing lists are for answering user questions, and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them.

Also, please do not send reports to the developers’ mailing list <pgsql-hackers@postgresql.org>. This list is for discussing the development of PostgreSQL, and it would be nice if we could keep the bug reports separate. We might choose to take up a discussion about your bug report on psql-hackers, if the problem needs more review.

\(^6\) [http://www.chiark.greenend.org.uk/~sgtatham/bugs.html](http://www.chiark.greenend.org.uk/~sgtatham/bugs.html)

\(^7\) [http://www.postgresql.org/](http://www.postgresql.org/)
If you have a problem with the documentation, the best place to report it is the documentation mailing
list <pgsql-docs@postgresql.org>. Please be specific about what part of the documentation you
are unhappy with.

If your bug is a portability problem on a non-supported platform, send mail to
<pgsql-hackers@postgresql.org>, so we (and you) can work on porting PostgreSQL to your
platform.

**Note:** Due to the unfortunate amount of spam going around, all of the above email addresses
are closed mailing lists. That is, you need to be subscribed to a list to be allowed to post on it.
(You need not be subscribed to use the bug-report web form, however.) If you would like to send
mail but do not want to receive list traffic, you can subscribe and set your subscription option to
nomail. For more information send mail to <majordomo@postgresql.org> with the single word
help in the body of the message.
I. Tutorial

Welcome to the PostgreSQL Tutorial. The following few chapters are intended to give a simple introduction to PostgreSQL, relational database concepts, and the SQL language to those who are new to any one of these aspects. We only assume some general knowledge about how to use computers. No particular Unix or programming experience is required. This part is mainly intended to give you some hands-on experience with important aspects of the PostgreSQL system. It makes no attempt to be a complete or thorough treatment of the topics it covers.

After you have worked through this tutorial you might want to move on to reading Part II to gain a more formal knowledge of the SQL language, or Part IV for information about developing applications for PostgreSQL. Those who set up and manage their own server should also read Part III.
Chapter 1. Getting Started

1.1. Installation

Before you can use PostgreSQL you need to install it, of course. It is possible that PostgreSQL is already installed at your site, either because it was included in your operating system distribution or because the system administrator already installed it. If that is the case, you should obtain information from the operating system documentation or your system administrator about how to access PostgreSQL.

If you are not sure whether PostgreSQL is already available or whether you can use it for your experimentation then you can install it yourself. Doing so is not hard and it can be a good exercise. PostgreSQL can be installed by any unprivileged user; no superuser (root) access is required.

If you are installing PostgreSQL yourself, then refer to Chapter 15 for instructions on installation, and return to this guide when the installation is complete. Be sure to follow closely the section about setting up the appropriate environment variables.

If your site administrator has not set things up in the default way, you might have some more work to do. For example, if the database server machine is a remote machine, you will need to set the PGHOST environment variable to the name of the database server machine. The environment variable PGPORT might also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the database, you should consult your site administrator or, if that is you, the documentation to make sure that your environment is properly set up. If you did not understand the preceding paragraph then read the next section.

1.2. Architectural Fundamentals

Before we proceed, you should understand the basic PostgreSQL system architecture. Understanding how the parts of PostgreSQL interact will make this chapter somewhat clearer.

In database jargon, PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

• A server process, which manages the database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the clients. The database server program is called **postgres**.

• The user’s client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection. You should keep this in mind, because the files that can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

The PostgreSQL server can handle multiple concurrent connections from clients. For that purpose it starts (“forks”) a new process for each connection. From that point on, the client and the new
server process communicate without intervention by the original \texttt{postgres} process. Thus, the master server process is always running, waiting for client connections, whereas client and associated server processes come and go. (All of this is of course invisible to the user. We only mention it here for completeness.)

1.3. Creating a Database

The first test to see whether you can access the database server is to try to create a database. A running PostgreSQL server can manage many databases. Typically, a separate database is used for each project or for each user.

Possibly, your site administrator has already created a database for your use. He should have told you what the name of your database is. In that case you can omit this step and skip ahead to the next section.

To create a new database, in this example named \texttt{mydb}, you use the following command:

\$ \texttt{createdb mydb}

If this produces no response then this step was successful and you can skip over the remainder of this section.

If you see a message similar to

\texttt{createdb: command not found}

then PostgreSQL was not installed properly. Either it was not installed at all or the search path was not set correctly. Try calling the command with an absolute path instead:

\$ \texttt{/usr/local/pgsql/bin/createdb mydb}

The path at your site might be different. Contact your site administrator or check back in the installation instructions to correct the situation.

Another response could be this:

\texttt{createdb: could not connect to database postgres: could not connect to server: No such file or directory}

Is the server running locally and accepting connections on Unix domain socket "/tmp/.s.PGSQL.5432"?

This means that the server was not started, or it was not started where \texttt{createdb} expected it. Again, check the installation instructions or consult the administrator.

Another response could be this:

\texttt{createdb: could not connect to database postgres: FATAL: role "joe" does not exist}

where your own login name is mentioned. This will happen if the administrator has not created a PostgreSQL user account for you. (PostgreSQL user accounts are distinct from operating system user accounts.) If you are the administrator, see Chapter 19 for help creating accounts. You will need to become the operating system user under which PostgreSQL was installed (usually \texttt{postgres}) to create the first user account. It could also be that you were assigned a PostgreSQL user name that is different from your operating system user name; in that case you need to use the \texttt{-U} switch or set the \texttt{PGUSER} environment variable to specify your PostgreSQL user name.

If you have a user account but it does not have the privileges required to create a database, you will see the following:
createdb: database creation failed: ERROR: permission denied to create database

Not every user has authorization to create new databases. If PostgreSQL refuses to create databases for you then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs. If you installed PostgreSQL yourself then you should log in for the purposes of this tutorial under the user account that you started the server as. ¹

You can also create databases with other names. PostgreSQL allows you to create any number of databases at a given site. Database names must have an alphabetic first character and are limited to 63 bytes in length. A convenient choice is to create a database with the same name as your current user name. Many tools assume that database name as the default, so it can save you some typing. To create that database, simply type

$ createdb

If you do not want to use your database anymore you can remove it. For example, if you are the owner (creator) of the database mydb, you can destroy it using the following command:

$ dropdb mydb

(For this command, the database name does not default to the user account name. You always need to specify it.) This action physically removes all files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

More about createdb and dropdb can be found in createdb and dropdb respectively.

1.4. Accessing a Database

Once you have created a database, you can access it by:

- Running the PostgreSQL interactive terminal program, called psql, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like pgAdmin or an office suite with ODBC support to create and manipulate a database. These possibilities are not covered in this tutorial.
- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in Part IV.

You probably want to start up psql, to try out the examples in this tutorial. It can be activated for the mydb database by typing the command:

$ psql mydb

If you leave off the database name then it will default to your user account name. You already discovered this scheme in the previous section.

In psql, you will be greeted with the following message:

1. As an explanation for why this works: PostgreSQL user names are separate from operating system user accounts. When you connect to a database, you can choose what PostgreSQL user name to connect as; if you don’t, it will default to the same name as your current operating system account. As it happens, there will always be a PostgreSQL user account that has the same name as the operating system user that started the server, and it also happens that that user always has permission to create databases. Instead of logging in as that user you can also specify the –U option everywhere to select a PostgreSQL user name to connect as.
Welcome to psql 8.3.23, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
\h for help with SQL commands
? for help with psql commands
\g or terminate with semicolon to execute query
\q to quit

mydb=>

The last line could also be

mydb=#

That would mean you are a database superuser, which is most likely the case if you installed PostgreSQL yourself. Being a superuser means that you are not subject to access controls. For the purposes of this tutorial that is not of importance.

If you encounter problems starting psql then go back to the previous section. The diagnostics of createdb and psql are similar, and if the former worked the latter should work as well.

The last line printed out by psql is the prompt, and it indicates that psql is listening to you and that you can type SQL queries into a work space maintained by psql. Try out these commands:

mydb=> SELECT version();

```
version
-----------------------------------------------------
PostgreSQL 8.3.23 on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)
```

mydb=> SELECT current_date;

date
------
2002-08-31
(1 row)

mydb=> SELECT 2 + 2;?

```
?column?
-------
4
(1 row)
```

The psql program has a number of internal commands that are not SQL commands. They begin with the backslash character, "\". Some of these commands were listed in the welcome message. For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

mydb=> \h

To get out of psql, type

mydb=> \q

and psql will quit and return you to your command shell. (For more internal commands, type \? at the psql prompt.) The full capabilities of psql are documented in psql. If PostgreSQL is installed correctly you can also type man psql at the operating system shell prompt to see the documentation.
In this tutorial we will not use these features explicitly, but you can use them yourself when you see fit.
Chapter 2. The SQL Language

2.1. Introduction

This chapter provides an overview of how to use SQL to perform simple operations. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. Numerous books have been written on SQL, including *Understanding the New SQL* and *A Guide to the SQL Standard*. You should be aware that some PostgreSQL language features are extensions to the standard.

In the examples that follow, we assume that you have created a database named `mydb`, as described in the previous chapter, and have been able to start `psql`.

Examples in this manual can also be found in the PostgreSQL source distribution in the directory `src/tutorial/`. To use those files, first change to that directory and run make:

```bash
$ cd ....~/src/tutorial
$ make
```

This creates the scripts and compiles the C files containing user-defined functions and types. (If you installed a pre-packaged version of PostgreSQL rather than building from source, look for a directory named `tutorial` within the PostgreSQL documentation. The “make” part should already have been done for you.) Then, to start the tutorial, do the following:

```bash
$ cd ....~/tutorial
$ psql -s mydb
...

mydb=> \i basics.sql
```

The `\i` command reads in commands from the specified file. The `-s` option puts you in single step mode which pauses before sending each statement to the server. The commands used in this section are in the file `basics.sql`.

2.2. Concepts

PostgreSQL is a *relational database management system* (RDBMS). That means it is a system for managing data stored in *relations*. Relation is essentially a mathematical term for *table*. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

Each table is a named collection of *rows*. Each row of a given table has the same set of named *columns*, and each column is of a specific data type. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Tables are grouped into databases, and a collection of databases managed by a single PostgreSQL server instance constitutes a database *cluster*. 
2.3. Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (  
city varchar(80), 
temp_lo int, -- low temperature 
temp_hi int, -- high temperature 
prcp real, -- precipitation 
date date
);
```

You can enter this into `psql` with the line breaks. `psql` will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) can be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dashes ("--") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

`varchar(80)` specifies a data type that can store arbitrary character strings up to 80 characters in length. `int` is the normal integer type. `real` is a type for storing single precision floating-point numbers. `date` should be self-explanatory. (Yes, the column of type `date` is also named `date`. This might be convenient or confusing — you choose.)

PostgreSQL supports the standard SQL types `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp`, and `interval`, as well as other types of general utility and a rich set of geometric types. PostgreSQL can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical key words, except where required to support special cases in the SQL standard.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (  
  name varchar(80),  
  location point
);
```

The `point` type is an example of a PostgreSQL-specific data type.

Finally, it should be mentioned that if you don’t need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

2.4. Populating a Table With Rows

The `INSERT` statement is used to populate a table with rows:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```
Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes (‘’), as in the example. The date type is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here.

The point type requires a coordinate pair as input, as shown here:

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the precipitation is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

You could also have used COPY to load large amounts of data from flat-text files. This is usually faster because the COPY command is optimized for this application while allowing less flexibility than INSERT. An example would be:

```
COPY weather FROM '/home/user/weather.txt';
```

where the file name for the source file must be available to the backend server machine, not the client, since the backend server reads the file directly. You can read more about the COPY command in COPY.

### 2.5. Querying a Table

To retrieve data from a table, the table is queried. An SQL SELECT statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT * FROM weather;
```

Here * is a shorthand for “all columns”. So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

The output should be:

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
</table>

1. While SELECT * is useful for off-the-cuff queries, it is widely considered bad style in production code, since adding a column to the table would change the results.
You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

This should give:

<table>
<thead>
<tr>
<th>city</th>
<th>temp_avg</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>48</td>
<td>1994-11-27</td>
</tr>
<tr>
<td>San Francisco</td>
<td>50</td>
<td>1994-11-29</td>
</tr>
<tr>
<td>Hayward</td>
<td>45</td>
<td>1994-11-29</td>
</tr>
</tbody>
</table>

(3 rows)

Notice how the AS clause is used to relabel the output column. (The AS clause is optional.)

A query can be "qualified" by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT * FROM weather
WHERE city = 'San Francisco' AND prcp > 0.0;
```

Result:

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
</tr>
</tbody>
</table>

(1 row)

You can request that the results of a query be returned in sorted order:

```
SELECT * FROM weather
ORDER BY city;
```

```
<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayward</td>
<td>37</td>
<td>54</td>
<td></td>
<td>1994-11-29</td>
</tr>
<tr>
<td>San Francisco</td>
<td>43</td>
<td>57</td>
<td>0</td>
<td>1994-11-29</td>
</tr>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
</tr>
</tbody>
</table>
```

In this example, the sort order isn’t fully specified, and so you might get the San Francisco rows in either order. But you’d always get the results shown above if you do:

```
SELECT * FROM weather
ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:
SELECT DISTINCT city
FROM weather;

city
---------------
Hayward
San Francisco
(2 rows)

Here again, the result row ordering might vary. You can ensure consistent results by using `DISTINCT` and `ORDER BY` together:

```sql
SELECT DISTINCT city
FROM weather
ORDER BY city;
```

### 2.6. Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a *join* query. As an example, say you wish to list all the weather records together with the location of the associated city. To do that, we need to compare the city column of each row of the weather table with the name column of all rows in the cities table, and select the pairs of rows where these values match.

**Note:** This is only a conceptual model. The join is usually performed in a more efficient manner than actually comparing each possible pair of rows, but this is invisible to the user.

This would be accomplished by the following query:

```sql
SELECT *
FROM weather, cities
WHERE city = name;
```

city | temp_lo | temp_hi | prcp | date | name | location
---------------+---------+---------+------|------------+---------------+-----------
San Francisco | 46 | 50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
San Francisco | 43 | 57 | 0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the `cities` table for Hayward, so the join ignores the unmatched rows in the weather table. We will see shortly how this can be fixed.

---

2. In some database systems, including older versions of PostgreSQL, the implementation of `DISTINCT` automatically orders the rows and so `ORDER BY` is unnecessary. But this is not required by the SQL standard, and current PostgreSQL doesn’t guarantee that `DISTINCT` causes the rows to be ordered.
There are two columns containing the city name. This is correct because the lists of columns of the `weather` and the `cities` table are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using `*`

```sql
SELECT city, temp_lo, temp_hi, prcp, date, location
FROM weather, cities
WHERE city = name;
```

**Exercise:** Attempt to find out the semantics of this query when the `WHERE` clause is omitted.

Since the columns all had different names, the parser automatically found out which table they belong to. If there were duplicate column names in the two tables you’d need to qualify the column names to show which one you meant, as in:

```sql
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
FROM weather, cities
WHERE cities.name = weather.city;
```

It is widely considered good style to qualify all column names in a join query, so that the query won’t fail if a duplicate column name is later added to one of the tables.

Join queries of the kind seen thus far can also be written in this alternative form:

```sql
SELECT *
FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the `weather` table and for each row to find the matching `cities` row(s). If no matching row is found we want some “empty values” to be substituted for the `cities` table’s columns. This kind of query is called an outer join. (The joins we have seen so far are inner joins.) The command looks like this:

```sql
SELECT *
FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);
```

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
<th>name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayward</td>
<td>37</td>
<td>54</td>
<td></td>
<td>1994-11-29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
<td>San Francisco</td>
<td>(-194,53)</td>
</tr>
<tr>
<td>San Francisco</td>
<td>43</td>
<td>57</td>
<td>0</td>
<td>1994-11-29</td>
<td>San Francisco</td>
<td>(-194,53)</td>
</tr>
</tbody>
</table>

(3 rows)

This query is called a left outer join because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (null) values are substituted for the right-table columns.

**Exercise:** There are also right outer joins and full outer joins. Try to find out what those do.

We can also join a table against itself. This is called a self join. As an example, suppose we wish to find all the weather records that are in the temperature range of other weather records. So we need to compare the `temp_lo` and `temp_hi` columns of each `weather` row to the `temp_lo` and `temp_hi` columns of all other `weather` rows. We can do this with the following query:

```sql
11
```
Chapter 2. The SQL Language

SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
     W2.city, W2.temp_lo AS low, W2.temp_hi AS high
FROM weather W1, weather W2
WHERE W1.temp_lo < W2.temp_lo
     AND W1.temp_hi > W2.temp_hi;

<table>
<thead>
<tr>
<th>city</th>
<th>low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>43</td>
<td>57</td>
</tr>
<tr>
<td>Hayward</td>
<td>37</td>
<td>54</td>
</tr>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
</tr>
</tbody>
</table>

(2 rows)

Here we have relabeled the weather table as W1 and W2 to be able to distinguish the left and right side of the join. You can also use these kinds of aliases in other queries to save some typing, e.g.:

SELECT *
     FROM weather w, cities c
     WHERE w.city = c.name;

You will encounter this style of abbreviating quite frequently.

2.7. Aggregate Functions

Like most other relational database products, PostgreSQL supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the count, sum, avg (average), max (maximum) and min (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with:

SELECT max(temp_lo) FROM weather;

<table>
<thead>
<tr>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
</tr>
</tbody>
</table>

(1 row)

If we wanted to know what city (or cities) that reading occurred in, we might try:

SELECT city FROM weather WHERE temp_lo = max(temp_lo);  WRONG

but this will not work since the aggregate max cannot be used in the WHERE clause. (This restriction exists because the WHERE clause determines which rows will be included in the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed.) However, as is often the case the query can be restated to accomplish the desired result, here by using a subquery:

SELECT city FROM weather
     WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

<table>
<thead>
<tr>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
</tr>
</tbody>
</table>

(1 row)
This is OK because the subquery is an independent computation that computes its own aggregate separately from what is happening in the outer query.

Aggregates are also very useful in combination with GROUP BY clauses. For example, we can get the maximum low temperature observed in each city with:

```sql
SELECT city, max(temp_lo)  
FROM weather  
GROUP BY city;
```

<table>
<thead>
<tr>
<th>city</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayward</td>
<td>37</td>
</tr>
<tr>
<td>San Francisco</td>
<td>46</td>
</tr>
</tbody>
</table>

(2 rows)

which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using HAVING:

```sql
SELECT city, max(temp_lo)  
FROM weather  
GROUP BY city  
HAVING max(temp_lo) < 40;
```

<table>
<thead>
<tr>
<th>city</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayward</td>
<td>37</td>
</tr>
</tbody>
</table>

(1 row)

which gives us the same results for only the cities that have all temp_lo values below 40. Finally, if we only care about cities whose names begin with “S”, we might do:

```sql
SELECT city, max(temp_lo)  
FROM weather  
WHERE city LIKE 'S%'  
GROUP BY city  
HAVING max(temp_lo) < 40;
```

The LIKE operator does pattern matching and is explained in Section 9.7.

It is important to understand the interaction between aggregates and SQL’s WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is this: WHERE selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects group rows after groups and aggregates are computed. Thus, the WHERE clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the HAVING clause always contains aggregate functions. (Strictly speaking, you are allowed to write a HAVING clause that doesn’t use aggregates, but it’s seldom useful. The same condition could be used more efficiently at the WHERE stage.)

In the previous example, we can apply the city name restriction in WHERE, since it needs no aggregate. This is more efficient than adding the restriction to HAVING, because we avoid doing the grouping and aggregate calculations for all rows that fail the WHERE check.
2.8. Updates

You can update existing rows using the UPDATE command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You can correct the data as follows:

```
UPDATE weather
    SET temp_hi = temp_hi - 2, temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT * FROM weather;
```

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
</tr>
<tr>
<td>San Francisco</td>
<td>41</td>
<td>55</td>
<td>0</td>
<td>1994-11-29</td>
</tr>
<tr>
<td>Hayward</td>
<td>35</td>
<td>52</td>
<td></td>
<td>1994-11-29</td>
</tr>
</tbody>
</table>

(3 rows)

2.9. Deletions

Rows can be removed from a table using the DELETE command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

```
DELETE FROM weather WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT * FROM weather;
```

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
</tr>
<tr>
<td>San Francisco</td>
<td>41</td>
<td>55</td>
<td>0</td>
<td>1994-11-29</td>
</tr>
</tbody>
</table>

(2 rows)

One should be wary of statements of the form

```
DELETE FROM tablename;
```

Without a qualification, DELETE will remove all rows from the given table, leaving it empty. The system will not request confirmation before doing this!
Chapter 3. Advanced Features

3.1. Introduction

In the previous chapter we have covered the basics of using SQL to store and access your data in PostgreSQL. We will now discuss some more advanced features of SQL that simplify management and prevent loss or corruption of your data. Finally, we will look at some PostgreSQL extensions.

This chapter will on occasion refer to examples found in Chapter 2 to change or improve them, so it will be of advantage if you have read that chapter. Some examples from this chapter can also be found in advanced.sql in the tutorial directory. This file also contains some example data to load, which is not repeated here. (Refer to Section 2.1 for how to use the file.)

3.2. Views

Refer back to the queries in Section 2.6. Suppose the combined listing of weather records and city location is of particular interest to your application, but you do not want to type the query each time you need it. You can create a view over the query, which gives a name to the query that you can refer to like an ordinary table:

```sql
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
    FROM weather, cities
    WHERE city = name;

SELECT * FROM myview;
```

Making liberal use of views is a key aspect of good SQL database design. Views allow you to encapsulate the details of the structure of your tables, which might change as your application evolves, behind consistent interfaces.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.

3.3. Foreign Keys

Recall the weather and cities tables from Chapter 2. Consider the following problem: You want to make sure that no one can insert rows in the weather table that do not have a matching entry in the cities table. This is called maintaining the referential integrity of your data. In simplistic database systems this would be implemented (if at all) by first looking at the cities table to check if a matching record exists, and then inserting or rejecting the new weather records. This approach has a number of problems and is very inconvenient, so PostgreSQL can do this for you.

The new declaration of the tables would look like this:

```sql
CREATE TABLE cities {
    city varchar(80) primary key,
    location point
};
```
CREATE TABLE weather (  
city   varchar(80) references cities(city),
temp_lo int,
temp_hi int,
prcp   real,
date   date
);

Now try inserting an invalid record:

INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR: insert or update on table "weather" violates foreign key constraint "weather_city_fkey"
DETAIL: Key (city)=(Berkeley) is not present in table "cities".

The behavior of foreign keys can be finely tuned to your application. We will not go beyond this simple example in this tutorial, but just refer you to Chapter 5 for more information. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn about them.

3.4. Transactions

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice’s account to Bob’s account. Simplifying outrageously, the SQL commands for this might look like:

```
UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
   WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
   WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

The details of these commands are not important here; the important point is that there are several separate updates involved to accomplish this rather simple operation. Our bank’s officers will want to be assured that either all these updates happen, or none of them happen. It would certainly not do for a system failure to result in Bob receiving $100.00 that was not debited from Alice. Nor would Alice long remain a happy customer if she was debited without Bob being credited. We need a guarantee that if something goes wrong partway through the operation, none of the steps executed so far will take effect. Grouping the updates into a transaction gives us this guarantee. A transaction is said to be atomic: from the point of view of other transactions, it either happens completely or not at all.
Chapter 3. Advanced Features

We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won’t be lost even if a crash ensues shortly thereafter. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just after he walks out the bank door. A transactional database guarantees that all the updates made by a transaction are logged in permanent storage (i.e., on disk) before the transaction is reported complete.

Another important property of transactional databases is closely related to the notion of atomic updates: when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others. For example, if one transaction is busy totalling all the branch balances, it would not do for it to include the debit from Alice’s branch but not the credit to Bob’s branch, nor vice versa. So transactions must be all-or-nothing not only in terms of their permanent effect on the database, but also in terms of their visibility as they happen. The updates made so far by an open transaction are invisible to other transactions until the transaction completes, whereupon all the updates become visible simultaneously.

In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with \texttt{BEGIN} and \texttt{COMMIT} commands. So our banking transaction would actually look like:

\begin{verbatim}
BEGIN;
UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';
-- etc etc
COMMIT;
\end{verbatim}

If, partway through the transaction, we decide we do not want to commit (perhaps we just noticed that Alice’s balance went negative), we can issue the command \texttt{ROLLBACK} instead of \texttt{COMMIT}, and all our updates so far will be canceled.

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a \texttt{BEGIN} command, then each individual statement has an implicit \texttt{BEGIN} and (if successful) \texttt{COMMIT} wrapped around it. A group of statements surrounded by \texttt{BEGIN} and \texttt{COMMIT} is sometimes called a transaction block.

\textbf{Note:} Some client libraries issue \texttt{BEGIN} and \texttt{COMMIT} commands automatically, so that you might get the effect of transaction blocks without asking. Check the documentation for the interface you are using.

It’s possible to control the statements in a transaction in a more granular fashion through the use of \texttt{savepoints}. Savepoints allow you to selectively discard parts of the transaction, while committing the rest. After defining a savepoint with \texttt{SAVEPOINT}, you can if needed roll back to the savepoint with \texttt{ROLLBACK TO}. All the transaction’s database changes between defining the savepoint and rolling back to it are discarded, but changes earlier than the savepoint are kept.

After rolling back to a savepoint, it continues to be defined, so you can roll back to it several times. Conversely, if you are sure you won’t need to roll back to a particular savepoint again, it can be released, so the system can free some resources. Keep in mind that either releasing or rolling back to a savepoint will automatically release all savepoints that were defined after it.

All this is happening within the transaction block, so none of it is visible to other database sessions. When and if you commit the transaction block, the committed actions become visible as a unit to other sessions, while the rolled-back actions never become visible at all.
Remembering the bank database, suppose we debit $100.00 from Alice’s account, and credit Bob’s account, only to find later that we should have credited Wally’s account. We could do it using savepoints like this:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Bob';
-- oops ... forget that and use Wally’s account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Wally';
COMMIT;
```

This example is, of course, oversimplified, but there’s a lot of control to be had over a transaction block through the use of savepoints. Moreover, ROLLBACK TO is the only way to regain control of a transaction block that was put in aborted state by the system due to an error, short of rolling it back completely and starting again.

### 3.5. Inheritance

Inheritance is a concept from object-oriented databases. It opens up interesting new possibilities of database design.

Let’s create two tables: A table `cities` and a table `capitals`. Naturally, capitals are also cities, so you want some way to show the capitals implicitly when you list all cities. If you’re really clever you might invent some scheme like this:

```
CREATE TABLE capitals (
   name text,
   population real,
   altitude int,     -- (in ft)
   state char(2)
);

CREATE TABLE non_capitals (
   name text,
   population real,
   altitude int -- (in ft)
);

CREATE VIEW cities AS
   SELECT name, population, altitude FROM capitals
UNION
   SELECT name, population, altitude FROM non_capitals;
```

This works OK as far as querying goes, but it gets ugly when you need to update several rows, for one thing.

A better solution is this:

```
CREATE TABLE cities (
   name text,
   population real,
   altitude int -- (in ft)
);
```
In this case, a row of capitals inherits all columns (name, population, and altitude) from its parent, cities. The type of the column name is text, a native PostgreSQL type for variable length character strings. State capitals have an extra column, state, that shows their state. In PostgreSQL, a table can inherit from zero or more other tables.

For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500 feet:

```sql
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

which returns:

<table>
<thead>
<tr>
<th>name</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Las Vegas</td>
<td>2174</td>
</tr>
<tr>
<td>Mariposa</td>
<td>1953</td>
</tr>
<tr>
<td>Madison</td>
<td>845</td>
</tr>
</tbody>
</table>

(3 rows)

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude of 500 feet or higher:

```sql
SELECT name, altitude
FROM ONLY cities
WHERE altitude > 500;
```

which returns:

<table>
<thead>
<tr>
<th>name</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Las Vegas</td>
<td>2174</td>
</tr>
<tr>
<td>Mariposa</td>
<td>1953</td>
</tr>
</tbody>
</table>

(2 rows)

Here the ONLY before cities indicates that the query should be run over only the cities table, and not tables below cities in the inheritance hierarchy. Many of the commands that we have already discussed — SELECT, UPDATE, and DELETE — support this ONLY notation.

**Note:** Although inheritance is frequently useful, it has not been integrated with unique constraints or foreign keys, which limits its usefulness. See Section 5.8 for more detail.
3.6. Conclusion

PostgreSQL has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These features are discussed in more detail in the remainder of this book. If you feel you need more introductory material, please visit the PostgreSQL web site\(^1\) for links to more resources.

\(^1\) http://www.postgresql.org
II. The SQL Language

This part describes the use of the SQL language in PostgreSQL. We start with describing the general syntax of SQL, then explain how to create the structures to hold data, how to populate the database, and how to query it. The middle part lists the available data types and functions for use in SQL commands. The rest treats several aspects that are important for tuning a database for optimal performance.

The information in this part is arranged so that a novice user can follow it start to end to gain a full understanding of the topics without having to refer forward too many times. The chapters are intended to be self-contained, so that advanced users can read the chapters individually as they choose. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should look into Part VI.

Readers of this part should know how to connect to a PostgreSQL database and issue SQL commands. Readers that are unfamiliar with these issues are encouraged to read Part I first. SQL commands are typically entered using the PostgreSQL interactive terminal psql, but other programs that have similar functionality can be used as well.
Chapter 4. SQL Syntax

This chapter describes the syntax of SQL. It forms the foundation for understanding the following chapters which will go into detail about how the SQL commands are applied to define and modify data.

We also advise users who are already familiar with SQL to read this chapter carefully because there are several rules and concepts that are implemented inconsistently among SQL databases or that are specific to PostgreSQL.

4.1. Lexical Structure

SQL input consists of a sequence of commands. A command is composed of a sequence of tokens, terminated by a semicolon ";". The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a key word, an identifier, a quoted identifier, a literal (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

Additionally, comments can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace.

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usefully be split across lines).

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a “SELECT”, an “UPDATE”, and an “INSERT” command. But for instance the UPDATE command always requires a SET token to appear in a certain position, and this particular variation of INSERT also requires a VALUES in order to be complete. The precise syntax rules for each command are described in Part VI.

4.1.1. Identifiers and Key Words

Tokens such as SELECT, UPDATE, or VALUES in the example above are examples of key words, that is, words that have a fixed meaning in the SQL language. The tokens MY_TABLE and A are examples of identifiers. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called “names”. Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language. A complete list of key words can be found in Appendix C.

SQL identifiers and key words must begin with a letter (a-z, but also letters with diacritical marks and non-Latin letters) or an underscore (_). Subsequent characters in an identifier or key word can be letters, underscores, digits (0-9), or dollar signs ($). Note that dollar signs are not allowed in identifiers according to the letter of the SQL standard, so their use might render applications less portable. The
Chapter 4. SQL Syntax

SQL standard will not define a key word that contains digits or starts or ends with an underscore, so identifiers of this form are safe against possible conflict with future extensions of the standard.

The system uses no more than NAMEDATALEN-1 bytes of an identifier; longer names can be written in commands, but they will be truncated. By default, NAMEDATALEN is 64 so the maximum identifier length is 63 bytes. If this limit is problematic, it can be raised by changing the NAMEDATALEN constant in src/include/pg_config_manual.h.

Identifier and key word names are case insensitive. Therefore:

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as:

```
uPDaTE my_TabLE SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.:

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the delimited identifier or quoted identifier. It is formed by enclosing an arbitrary sequence of characters in double-quotes ("). A delimited identifier is always an identifier, never a key word. So *select* could be used to refer to a column or table named "select", whereas an unquoted select would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

```
UPDATE "my_table" SET "a" = 5;
```

Quoted identifiers can contain any character, except the character with code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers FOO, foo, and "foo" are considered the same by PostgreSQL, but "Foo" and "FOO" are different from these three and each other. (The folding of unquoted names to lower case in PostgreSQL is incompatible with the SQL standard, which says that unquoted names should be folded to upper case. Thus, foo should be equivalent to "FOO" not "foo" according to the standard. If you want to write portable applications you are advised to always quote a particular name or never quote it.)

4.1.2. Constants

There are three kinds of implicitly-typed constants in PostgreSQL: strings, bit strings, and numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

4.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes (’), for example ‘This is a string’. To include a single-quote character within a string constant, write
two adjacent single quotes, e.g. ‘Dianne’s horse’. Note that this is not the same as a double-quote character (").

Two string constants that are only separated by whitespace with at least one newline are concatenated and effectively treated as if the string had been written as one constant. For example:

```sql
SELECT 'foo'
 'bar';
```

is equivalent to:

```sql
SELECT 'foobar';
```

but:

```sql
SELECT 'foo' 'bar';
```

is not valid syntax. (This slightly bizarre behavior is specified by SQL; PostgreSQL is following the standard.)

PostgreSQL also accepts “escape” string constants, which are an extension to the SQL standard. An escape string constant is specified by writing the letter E (upper or lower case) just before the opening single quote, e.g. E’foo’. (When continuing an escape string constant across lines, write E only before the first opening quote.) Within an escape string, a backslash character (\) begins a C-like \_backslash escape sequence, in which the combination of backslash and following character(s) represents a special byte value. \b is a backspace, \f is a form feed, \n is a newline, \r is a carriage return, \t is a tab. Also supported are \_digits, where \_digits represents an octal byte value, and \_hexdigits, where \_hexdigits represents a hexadecimal byte value. (It is your responsibility that the byte sequences you create are valid characters in the server character set encoding.) Any other character following a backslash is taken literally. Thus, to include a backslash character, write two backslashes (\\). Also, a single quote can be included in an escape string by writing \\\', in addition to the normal way of ‘.

<table>
<thead>
<tr>
<th>Caution</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the configuration parameter standard_conforming_strings is off, then PostgreSQL recognizes backslash escapes in both regular and escape string constants. This is for backward compatibility with the historical behavior, in which backslash escapes were always recognized. Although standard_conforming_strings currently defaults to off, the default will change to on in a future release for improved standards compliance. Applications are therefore encouraged to migrate away from using backslash escapes. If you need to use a backslash escape to represent a special character, write the constant with an E to be sure it will be handled the same way in future releases. In addition to standard_conforming_strings, the configuration parameters escape_string_warning and backslash_quote govern treatment of backslashes in string constants.</td>
</tr>
</tbody>
</table>

The character with the code zero cannot be in a string constant.

### 4.1.2.2. Dollar-Quoted String Constants

While the standard syntax for specifying string constants is usually convenient, it can be difficult to understand when the desired string contains many single quotes or backslashes, since each of those
must be doubled. To allow more readable queries in such situations, PostgreSQL provides another
way, called “dollar quoting”, to write string constants. A dollar-quoted string constant consists of a
dollar sign ($), an optional “tag” of zero or more characters, another dollar sign, an arbitrary sequence
of characters that makes up the string content, a dollar sign, the same tag that began this dollar quote,
and a dollar sign. For example, here are two different ways to specify the string “Dianne’s horse”
using dollar quoting:

$$Dianne’s horse$$
$$SomeTag$$Dianne’s horse$$SomeTag$$

Notice that inside the dollar-quoted string, single quotes can be used without needing to be escaped.
Indeed, no characters inside a dollar-quoted string are ever escaped: the string content is always writ-
ten literally. Backslashes are not special, and neither are dollar signs, unless they are part of a sequence
matching the opening tag.

It is possible to nest dollar-quoted string constants by choosing different tags at each nesting level.
This is most commonly used in writing function definitions. For example:

$function$
BEGIN
   RETURN ($1 ~ $q$
   \$
   \$
   )
END;
$function$

Here, the sequence $q$

\$
\$
represents a dollar-quoted literal string \$
\$
, which will be recognized when the function body is executed by PostgreSQL. But since the sequence
does not match the outer dollar quoting delimiter $function$, it is just some more characters within
the constant so far as the outer string is concerned.

The tag, if any, of a dollar-quoted string follows the same rules as an unquoted identifier, except that
it cannot contain a dollar sign. Tags are case sensitive, so $tag$String content$tag$ is correct,
but $TAG$String content$tag$ is not.

A dollar-quoted string that follows a keyword or identifier must be separated from it by whitespace;
otherwise the dollar quoting delimiter would be taken as part of the preceding identifier.

Dollar quoting is not part of the SQL standard, but it is often a more convenient way to write com-
licated string literals than the standard-compliant single quote syntax. It is particularly useful when
representing string constants inside other constants, as is often needed in procedural function defini-
tions. With single-quote syntax, each backslash in the above example would have to be written as four
backslashes, which would be reduced to two backslashes in parsing the original string constant, and
then to one when the inner string constant is re-parsed during function execution.

### 4.1.2.3. Bit-String Constants

Bit-string constants look like regular string constants with a B (upper or lower case) immediately
before the opening quote (no intervening whitespace), e.g., B’1001’. The only characters allowed
within bit-string constants are 0 and 1.

Alternatively, bit-string constants can be specified in hexadecimal notation, using a leading X (upper
or lower case), e.g., X’1FF’. This notation is equivalent to a bit-string constant with four binary digits
for each hexadecimal digit.

Both forms of bit-string constant can be continued across lines in the same way as regular string
constants. Dollar quoting cannot be used in a bit-string constant.
4.1.2.4. Numeric Constants

Numeric constants are accepted in these general forms:

\[ \text{digits} \]
\[ \text{digits.}[\text{digits}][e[+-]\text{digits}] \]
\[ [\text{digits}].\text{digits}[e[+-]\text{digits}] \]
\[ \text{digitse[+-]}\text{digits} \]

where \( \text{digits} \) is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (\( e \)), if one is present. There cannot be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

42
3.5
4.
.001
5e2
1.925e-3

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type \text{integer} if its value fits in type \text{integer} (32 bits); otherwise it is presumed to be type \text{bigint} if its value fits in type \text{bigint} (64 bits); otherwise it is taken to be type \text{numeric}. Constants that contain decimal points and/or exponents are always initially presumed to be type \text{numeric}.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it. For example, you can force a numeric value to be treated as type \text{real} (\text{float}4) by writing:

\text{REAL} ‘1.23’ -- string style
1.23::REAL -- PostgreSQL (historical) style

These are actually just special cases of the general casting notations discussed next.

4.1.2.5. Constants of Other Types

A constant of an \textit{arbitrary} type can be entered using any one of the following notations:

\text{type} ‘\text{string}'
\text{‘string’::type}
\text{CAST} (‘\text{string}' AS \text{type})

The string constant’s text is passed to the input conversion routine for the type called \text{type}. The result is a constant of the indicated type. The explicit type cast can be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

The string constant can be written using either regular SQL notation or dollar-quoting.
It is also possible to specify a type coercion using a function-like syntax:

```sql
typename ( 'string' )
```

but not all type names can be used in this way; see Section 4.2.8 for details.

The `::`, `CAST()`, and function-call syntaxes can also be used to specify run-time type conversions of arbitrary expressions, as discussed in Section 4.2.8. To avoid syntactic ambiguity, the `type 'string'` syntax can only be used to specify the type of a simple literal constant. Another restriction on the `type 'string'` syntax is that it does not work for array types; use `::` or `CAST()` to specify the type of an array constant.

The `CAST()` syntax conforms to SQL. The `type 'string'` syntax is a generalization of the standard: SQL specifies this syntax only for a few data types, but PostgreSQL allows it for all types. The syntax with `::` is historical PostgreSQL usage, as is the function-call syntax.

### 4.1.3. Operators

An operator name is a sequence of up to `NAMEDATALEN`-1 (63 by default) characters from the following list:

```sql
+-*/<>=~-!@#$%^&|?'\n```

There are a few restrictions on operator names, however:

- `- -` and `/ *` cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multiple-character operator name cannot end in `+` or `-`, unless the name also contains at least one of these characters:
  ```sql
  ~ ! @ # % ^ & | ?
  ```
  For example, `@ -` is an allowed operator name, but `* -` is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left unary operator named `@`, you cannot write `X @ Y`; you must write `X @ Y` to ensure that PostgreSQL reads it as two operator names not one.

### 4.1.4. Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

- A dollar sign (`$`) followed by digits is used to represent a positional parameter in the body of a function definition or a prepared statement. In other contexts the dollar sign can be part of an
identifier or a dollar-quoted string constant.

- Parentheses (()) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.
- Brackets ([])) are used to select the elements of an array. See Section 8.14 for more information on arrays.
- Commas (,) are used in some syntactical constructs to separate the elements of a list.
- The semicolon (;) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
- The colon (:) is used to select “slices” from arrays. (See Section 8.14.) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
- The asterisk (*) is used in some contexts to denote all the fields of a table row or composite value. It also has a special meaning when used as the argument of an aggregate function, namely that the aggregate does not require any explicit parameter.
- The period (.) is used in numeric constants, and to separate schema, table, and column names.

4.1.5. Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```sql
-- This is a standard SQL comment
```

Alternatively, C-style block comments can be used:

```sql
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

where the comment begins with /* and extends to the matching occurrence of */. These block comments nest, as specified in the SQL standard but unlike C, so that one can comment out larger blocks of code that might contain existing block comments.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

4.1.6. Lexical Precedence

Table 4-1 shows the precedence and associativity of the operators in PostgreSQL. Most operators have the same precedence and are left-associative. The precedence and associativity of the operators is hard-wired into the parser. This can lead to non-intuitive behavior; for example the Boolean operators < and > have a different precedence than the Boolean operators <= and >=. Also, you will sometimes need to add parentheses when using combinations of binary and unary operators. For instance:

```sql
SELECT 5 ! - 6;
```

will be parsed as:
SELECT 5 ! (- 6);

because the parser has no idea — until it is too late — that ! is defined as a postfix operator, not an infix one. To get the desired behavior in this case, you must write:

SELECT (5 !) - 6;

This is the price one pays for extensibility.

Table 4-1. Operator Precedence (decreasing)

<table>
<thead>
<tr>
<th>Operator/Element</th>
<th>Associativity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>left</td>
<td>table/column name separator</td>
</tr>
<tr>
<td>::</td>
<td>left</td>
<td>PostgreSQL-style typecast</td>
</tr>
<tr>
<td>[ ]</td>
<td>left</td>
<td>array element selection</td>
</tr>
<tr>
<td>-</td>
<td>right</td>
<td>unary minus</td>
</tr>
<tr>
<td>^</td>
<td>left</td>
<td>exponentiation</td>
</tr>
<tr>
<td>* / %</td>
<td>left</td>
<td>multiplication, division, modulo</td>
</tr>
<tr>
<td>+ -</td>
<td>left</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>IS</td>
<td></td>
<td>IS TRUE, IS FALSE, IS UNKNOWN, IS NULL</td>
</tr>
<tr>
<td>ISNULL</td>
<td></td>
<td>test for null</td>
</tr>
<tr>
<td>NOTNULL</td>
<td></td>
<td>test for not null</td>
</tr>
<tr>
<td>(any other)</td>
<td>left</td>
<td>all other native and user-defined operators</td>
</tr>
<tr>
<td>IN</td>
<td></td>
<td>set membership</td>
</tr>
<tr>
<td>BETWEEN</td>
<td></td>
<td>range containment</td>
</tr>
<tr>
<td>OVERLAPS</td>
<td></td>
<td>time interval overlap</td>
</tr>
<tr>
<td>LIKE ILIKE SIMILAR</td>
<td></td>
<td>string pattern matching</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>right</td>
<td>less than, greater than</td>
</tr>
<tr>
<td>=</td>
<td>right</td>
<td>equality, assignment</td>
</tr>
<tr>
<td>NOT</td>
<td>right</td>
<td>logical negation</td>
</tr>
<tr>
<td>AND</td>
<td>left</td>
<td>logical conjunction</td>
</tr>
<tr>
<td>OR</td>
<td>left</td>
<td>logical disjunction</td>
</tr>
</tbody>
</table>

Note that the operator precedence rules also apply to user-defined operators that have the same names as the built-in operators mentioned above. For example, if you define a “+” operator for some custom data type it will have the same precedence as the built-in “+” operator, no matter what yours does.

When a schema-qualified operator name is used in the `OPERATOR` syntax, as for example in:

SELECT 3 OPERATOR(pg_catalog.+ 4);

the `OPERATOR` construct is taken to have the default precedence shown in Table 4-1 for “any other” operator. This is true no matter which specific operator name appears inside `OPERATOR()`.
4.2. Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the `SELECT` command, as new column values in `INSERT` or `UPDATE`, or in search conditions in a number of commands. The result of a value expression is sometimes called a *scalar*, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called *scalar expressions* (or even simply *expressions*). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value.
- A column reference.
- A positional parameter reference, in the body of a function definition or prepared statement.
- A subscripted expression.
- A field selection expression.
- An operator invocation.
- A function call.
- An aggregate expression.
- A type cast.
- A scalar subquery.
- An array constructor.
- A row constructor.
- Another value expression in parentheses, useful to group subexpressions and override precedence.

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in the appropriate location in Chapter 9. An example is the `IS NULL` clause.

We have already discussed constants in Section 4.1.2. The following sections discuss the remaining options.

4.2.1. Column References

A column can be referenced in the form

```
correlation.columnname
```

`correlation` is the name of a table (possibly qualified with a schema name), or an alias for a table defined by means of a `FROM` clause, or one of the key words `NEW` or `OLD`, (`NEW` and `OLD` can only appear in rewrite rules, while other correlation names can be used in any SQL statement.) The correlation name and separating dot can be omitted if the column name is unique across all the tables being used in the current query. (See also Chapter 7.)
4.2.2. Positional Parameters

A positional parameter reference is used to indicate a value that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. Some client libraries also support specifying data values separately from the SQL command string, in which case parameters are used to refer to the out-of-line data values. The form of a parameter reference is:

$number$

For example, consider the definition of a function, `dept`, as:

```sql
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Here the `$1` references the value of the first function argument whenever the function is invoked.

4.2.3. Subscripts

If an expression yields a value of an array type, then a specific element of the array value can be extracted by writing:

`expression[subscript]`

or multiple adjacent elements (an “array slice”) can be extracted by writing:

`expression[lower_subscript:upper_subscript]`

(Here, the brackets `[ ]` are meant to appear literally.) Each `subscript` is itself an expression, which must yield an integer value.

In general the array `expression` must be parenthesized, but the parentheses can be omitted when the expression to be subscripted is just a column reference or positional parameter. Also, multiple subscripts can be concatenated when the original array is multidimensional. For example:

```sql
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

The parentheses in the last example are required. See Section 8.14 for more about arrays.

4.2.4. Field Selection

If an expression yields a value of a composite type (row type), then a specific field of the row can be extracted by writing:

`expression.fieldname`

In general the row `expression` must be parenthesized, but the parentheses can be omitted when the expression to be selected from is just a table reference or positional parameter. For example:
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3

(Thus, a qualified column reference is actually just a special case of the field selection syntax.)

### 4.2.5. Operator Invocations

There are three possible syntaxes for an operator invocation:

- `expression operator expression` (binary infix operator)
- `operator expression` (unary prefix operator)
- `expression operator` (unary postfix operator)

where the *operator* token follows the syntax rules of Section 4.1.3, or is one of the key words **AND**, **OR**, and **NOT**, or is a qualified operator name in the form

```
OPERATOR(schema.operatorname)
```

Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. Chapter 9 describes the built-in operators.

### 4.2.6. Function Calls

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

```
function ([expression [ , expression ... ]] )
```

For example, the following computes the square root of 2:

```
sqrt(2)
```

The list of built-in functions is in Chapter 9. Other functions can be added by the user.

### 4.2.7. Aggregate Expressions

An *aggregate expression* represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

- `aggregate_name (expression [ , ... ] )`
- `aggregate_name (ALL expression [ , ... ] )`
- `aggregate_name (DISTINCT expression [ , ... ] )`
- `aggregate_name ( * )`

where `aggregate_name` is a previously defined aggregate (possibly qualified with a schema name), and `expression` is any value expression that does not itself contain an aggregate expression.
The first form of aggregate expression invokes the aggregate across all input rows for which the given expression(s) yield non-null values. (Actually, it is up to the aggregate function whether to ignore null values or not — but all the standard ones do.) The second form is the same as the first, since \texttt{ALL} is the default. The third form invokes the aggregate for all distinct non-null values of the expressions found in the input rows. The last form invokes the aggregate once for each input row regardless of null or non-null values; since no particular input value is specified, it is generally only useful for the \texttt{count(*)} aggregate function.

For example, \texttt{count(*)} yields the total number of input rows; \texttt{count(f1)} yields the number of input rows in which \texttt{f1} is non-null; \texttt{count(distinct f1)} yields the number of distinct non-null values of \texttt{f1}.

The predefined aggregate functions are described in Section 9.18. Other aggregate functions can be added by the user.

An aggregate expression can only appear in the result list or \texttt{HAVING} clause of a \texttt{SELECT} command. It is forbidden in other clauses, such as \texttt{WHERE}, because those clauses are logically evaluated before the results of aggregates are formed.

When an aggregate expression appears in a subquery (see Section 4.2.9 and Section 9.19), the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate’s arguments contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or \texttt{HAVING} clause applies with respect to the query level that the aggregate belongs to.

\textbf{Note:} PostgreSQL currently \textbf{does not} support \texttt{DISTINCT} with more than one input expression.

### 4.2.8. Type Casts

A type cast specifies a conversion from one data type to another. PostgreSQL accepts two equivalent syntaxes for type casts:

\begin{verbatim}
CAST ( expression AS type )
expression::type
\end{verbatim}

The \texttt{CAST} syntax conforms to SQL; the syntax with \texttt{::} is historical PostgreSQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined. Notice that this is subtly different from the use of casts with constants, as shown in Section 4.1.2.5. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast can usually be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases. However, automatic casting is only done for casts that are marked “OK to apply implicitly” in the system catalogs. Other casts must be invoked with explicit casting syntax. This restriction is intended to prevent surprising conversions from being applied silently.

It is also possible to specify a type cast using a function-like syntax:
However, this only works for types whose names are also valid as function names. For example, \texttt{double precision} cannot be used this way, but the equivalent \texttt{float8} can. Also, the names \texttt{interval}, \texttt{time}, and \texttt{timestamp} can only be used in this fashion if they are double-quoted, because of syntactic conflicts. Therefore, the use of the function-like cast syntax leads to inconsistencies and should probably be avoided in new applications.

\textbf{Note:} The function-like syntax is in fact just a function call. When one of the two standard cast syntaxes is used to do a run-time conversion, it will internally invoke a registered function to perform the conversion. By convention, these conversion functions have the same name as their output type, and thus the “function-like syntax” is nothing more than a direct invocation of the underlying conversion function. Obviously, this is not something that a portable application should rely on. For further details see \texttt{CREATE CAST}.

4.2.9. Scalar Subqueries

A scalar subquery is an ordinary \texttt{SELECT} query in parentheses that returns exactly one row with one column. (See Chapter 7 for information about writing queries.) The \texttt{SELECT} query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is taken to be null.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery. See also Section 9.19 for other expressions involving subqueries.

For example, the following finds the largest city population in each state:

\begin{verbatim}
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
FROM states;
\end{verbatim}

4.2.10. Array Constructors

An array constructor is an expression that builds an array value from values for its member elements. A simple array constructor consists of the key word \texttt{ARRAY}, a left square bracket \texttt{[}, one or more expressions (separated by commas) for the array element values, and finally a right square bracket \texttt{]}. For example:

\begin{verbatim}
SELECT ARRAY[1,2,3+4];
array
----------
{1,2,7}
(1 row)
\end{verbatim}

The array element type is the common type of the member expressions, determined using the same rules as for \texttt{UNION} or \texttt{CASE} constructs (see Section 10.5).

Multidimensional array values can be built by nesting array constructors. In the inner constructors, the key word \texttt{ARRAY} can be omitted. For example, these produce the same result:

\begin{verbatim}
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
\end{verbatim}
array
-------------------
{{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
array
-------------------
{{1,2},{3,4}}
(1 row)

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions.

Multidimensional array constructor elements can be anything yielding an array of the proper kind, not only a sub-ARRAY construct. For example:

CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
array
----------------------------------------------
{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}
(1 row)

It is also possible to construct an array from the results of a subquery. In this form, the array constructor is written with the key word ARRAY followed by a parenthesized (not bracketed) subquery. For example:

SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
?column?
-------------------------------------------------------------
(1 row)

The subquery must return a single column. The resulting one-dimensional array will have an element for each row in the subquery result, with an element type matching that of the subquery’s output column.

The subscripts of an array value built with ARRAY always begin with one. For more information about arrays, see Section 8.14.

### 4.2.11. Row Constructors

A row constructor is an expression that builds a row value (also called a composite value) from values for its member fields. A row constructor consists of the key word ROW, a left parenthesis, zero or more expressions (separated by commas) for the row field values, and finally a right parenthesis. For example:

SELECT ROW(1,2.5,'this is a test');

The key word ROW is optional when there is more than one expression in the list.
A row constructor can include the syntax `rowvalue.*`, which will be expanded to a list of the elements of the row value, just as occurs when the `.*` syntax is used at the top level of a `SELECT` list. For example, if table `t` has columns `f1` and `f2`, these are the same:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

**Note:** Before PostgreSQL 8.2, the `.*` syntax was not expanded, so that writing `ROW(t.*, 42)` created a two-field row whose first field was another row value. The new behavior is usually more useful. If you need the old behavior of nested row values, write the inner row value without `.*`, for instance `ROW(t, 42).

By default, the value created by a `ROW` expression is of an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with `CREATE TYPE AS`. An explicit cast might be needed to avoid ambiguity. For example:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);
CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;
-- No cast needed since only one getf1() exists
SELECT getf1(ROW(1,2.5,'this is a test'));
getf1
-----
1
(1 row)
```

```
CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);
CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;
-- Now we need a cast to indicate which function to call:
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR: function getf1(record) is not unique
SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
getf1
-----
1
(1 row)
```

Row constructors can be used to build composite values to be stored in a composite-type table column, or to be passed to a function that accepts a composite parameter. Also, it is possible to compare two row values or test a row with `IS NULL` or `IS NOT NULL`, for example:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');
```
Chapter 4. SQL Syntax

SELECT ROW(table.*) IS NULL FROM table; -- detect all-null rows

For more detail see Section 9.20. Row constructors can also be used in connection with subqueries, as discussed in Section 9.19.

4.2.12. Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote:

SELECT true OR somefunc();

then somefunc() would (probably) not be called at all. The same would be the case if one wrote:

SELECT somefunc() OR true;

Note that this is not the same as the left-to-right "short-circuiting" of Boolean operators that is found in some programming languages.

As a consequence, it is unwise to use functions with side effects as part of complex expressions. It is particularly dangerous to rely on side effects or evaluation order in WHERE and HAVING clauses, since those clauses are extensively reprocessed as part of developing an execution plan. Boolean expressions (AND/OR/NOT combinations) in those clauses can be reorganized in any manner allowed by the laws of Boolean algebra.

When it is essential to force evaluation order, a CASE construct (see Section 9.16) can be used. For example, this is an untrustworthy way of trying to avoid division by zero in a WHERE clause:

SELECT ... WHERE x > 0 AND y/x > 1.5;

But this is safe:

SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;

A CASE construct used in this fashion will defeat optimization attempts, so it should only be done when necessary. (In this particular example, it would be better to sidestep the problem by writing \( y > 1.5 \times x \) instead.)
Chapter 5. Data Definition

This chapter covers how one creates the database structures that will hold one’s data. In a relational database, the raw data is stored in tables, so the majority of this chapter is devoted to explaining how tables are created and modified and what features are available to control what data is stored in the tables. Subsequently, we discuss how tables can be organized into schemas, and how privileges can be assigned to tables. Finally, we will briefly look at other features that affect the data storage, such as inheritance, views, functions, and triggers.

5.1. Table Basics

A table in a relational database is much like a table on paper: It consists of rows and columns. The number and order of the columns is fixed, and each column has a name. The number of rows is variable — it reflects how much data is stored at a given moment. SQL does not make any guarantees about the order of the rows in a table. When a table is read, the rows will appear in random order, unless sorting is explicitly requested. This is covered in Chapter 7. Furthermore, SQL does not assign unique identifiers to rows, so it is possible to have several completely identical rows in a table. This is a consequence of the mathematical model that underlies SQL but is usually not desirable. Later in this chapter we will see how to deal with this issue.

Each column has a data type. The data type constrains the set of possible values that can be assigned to a column and assigns semantics to the data stored in the column so that it can be used for computations. For instance, a column declared to be of a numerical type will not accept arbitrary text strings, and the data stored in such a column can be used for mathematical computations. By contrast, a column declared to be of a character string type will accept almost any kind of data but it does not lend itself to mathematical calculations, although other operations such as string concatenation are available.

PostgreSQL includes a sizable set of built-in data types that fit many applications. Users can also define their own data types. Most built-in data types have obvious names and semantics, so we defer a detailed explanation to Chapter 8. Some of the frequently used data types are integer for whole numbers, numeric for possibly fractional numbers, text for character strings, date for dates, time for time-of-day values, and timestamp for values containing both date and time.

To create a table, you use the aptly named CREATE TABLE command. In this command you specify at least a name for the new table, the names of the columns and the data type of each column. For example:

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

This creates a table named my_first_table with two columns. The first column is named first_column and has a data type of text; the second column has the name second_column and the type integer. The table and column names follow the identifier syntax explained in Section 4.1.1. The type names are usually also identifiers, but there are some exceptions. Note that the column list is comma-separated and surrounded by parentheses.

Of course, the previous example was heavily contrived. Normally, you would give names to your tables and columns that convey what kind of data they store. So let’s look at a more realistic example:

```
CREATE TABLE products {
```
Tip: When you create many interrelated tables it is wise to choose a consistent naming pattern for the tables and columns. For instance, there is a choice of using singular or plural nouns for table names, both of which are favored by some theorist or other.

There is a limit on how many columns a table can contain. Depending on the column types, it is between 250 and 1600. However, defining a table with anywhere near this many columns is highly unusual and often a questionable design.

If you no longer need a table, you can remove it using the `DROP TABLE` command. For example:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Attempting to drop a table that does not exist is an error. Nevertheless, it is common in SQL script files to unconditionally try to drop each table before creating it, ignoring any error messages, so that the script works whether or not the table exists. (If you like, you can use the `DROP TABLE IF EXISTS` variant to avoid the error messages, but this is not standard SQL.)

If you need to modify a table that already exists look into Section 5.5 later in this chapter.

With the tools discussed so far you can create fully functional tables. The remainder of this chapter is concerned with adding features to the table definition to ensure data integrity, security, or convenience. If you are eager to fill your tables with data now you can skip ahead to Chapter 6 and read the rest of this chapter later.

### 5.2. Default Values

A column can be assigned a default value. When a new row is created and no values are specified for some of the columns, those columns will be filled with their respective default values. A data manipulation command can also request explicitly that a column be set to its default value, without having to know what that value is. (Details about data manipulation commands are in Chapter 6.)

If no default value is declared explicitly, the default value is the null value. This usually makes sense because a null value can be considered to represent unknown data.

In a table definition, default values are listed after the column data type. For example:

```
CREATE TABLE products {
  product_no integer,
  name text,
  price numeric DEFAULT 9.99
};
```

The default value can be an expression, which will be evaluated whenever the default value is inserted (not when the table is created). A common example is that a `timestamp` column can have a default
of \texttt{now()}, so that it gets set to the time of row insertion. Another common example is generating a “serial number” for each row. In PostgreSQL this is typically done by something like:

\begin{verbatim}
CREATE TABLE products {
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
};
\end{verbatim}

where the \texttt{nextval()} function supplies successive values from a \textit{sequence object} (see Section 9.15). This arrangement is sufficiently common that there’s a special shorthand for it:

\begin{verbatim}
CREATE TABLE products {
    product_no SERIAL,
    ...
};
\end{verbatim}

The \texttt{SERIAL} shorthand is discussed further in Section 8.1.4.

5.3. Constraints

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no standard data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should be only one row for each product number.

To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

5.3.1. Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For instance, to require positive product prices, you could use:

\begin{verbatim}
CREATE TABLE products {
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
};
\end{verbatim}

As you see, the constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. A check constraint consists of the key word \texttt{CHECK} followed by an expression in parentheses. The check constraint expression should involve the column thus constrained, otherwise the constraint would not make too much sense.

You can also give the constraint a separate name. This clarifies error messages and allows you to refer to the constraint when you need to change it. The syntax is:
CREATE TABLE products {
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
};

So, to specify a named constraint, use the key word CONSTRAINT followed by an identifier followed by the constraint definition. (If you don’t specify a constraint name in this way, the system chooses a name for you.)

A check constraint can also refer to several columns. Say you store a regular price and a discounted price, and you want to ensure that the discounted price is lower than the regular price:

CREATE TABLE products {
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
};

The first two constraints should look familiar. The third one uses a new syntax. It is not attached to a particular column, instead it appears as a separate item in the comma-separated column list. Column definitions and these constraint definitions can be listed in mixed order.

We say that the first two constraints are column constraints, whereas the third one is a table constraint because it is written separately from any one column definition. Column constraints can also be written as table constraints, while the reverse is not necessarily possible, since a column constraint is supposed to refer to only the column it is attached to. (PostgreSQL doesn’t enforce that rule, but you should follow it if you want your table definitions to work with other database systems.) The above example could also be written as:

CREATE TABLE products {
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
};

or even:

CREATE TABLE products {
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
};

It’s a matter of taste.

Names can be assigned to table constraints in just the same way as for column constraints:

CREATE TABLE products {
product_no integer,
name text,
price numeric,
CHECK (price > 0),
discounted_price numeric,
CHECK (discounted_price > 0),
CONSTRAINT valid_discount CHECK (price > discounted_price)
);

It should be noted that a check constraint is satisfied if the check expression evaluates to true or the null value. Since most expressions will evaluate to the null value if any operand is null, they will not prevent null values in the constrained columns. To ensure that a column does not contain null values, the not-null constraint described in the next section can be used.

5.3.2. Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A syntax example:

```
CREATE TABLE products {
  product_no integer NOT NULL,
  name text NOT NULL,
  price numeric
};
```

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint `CHECK (column_name IS NOT NULL)`, but in PostgreSQL creating an explicit not-null constraint is more efficient. The drawback is that you cannot give explicit names to not-null constraints created this way.

Of course, a column can have more than one constraint. Just write the constraints one after another:

```
CREATE TABLE products {
  product_no integer NOT NULL,
  name text NOT NULL,
  price numeric NOT NULL CHECK (price > 0)
};
```

The order doesn’t matter. It does not necessarily determine in which order the constraints are checked. The NOT NULL constraint has an inverse: the NULL constraint. This does not mean that the column must be null, which would surely be useless. Instead, this simply selects the default behavior that the column might be null. The NULL constraint is not present in the SQL standard and should not be used in portable applications. (It was only added to PostgreSQL to be compatible with some other database systems.) Some users, however, like it because it makes it easy to toggle the constraint in a script file. For example, you could start with:

```
CREATE TABLE products {
  product_no integer NULL,
  name text NULL,
  price numeric NULL
};
```

and then insert the NOT key word where desired.
5.3.3. Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table. The syntax is:

```sql
CREATE TABLE products {
    product_no integer UNIQUE,
    name text,
    price numeric
};
```

when written as a column constraint, and:

```sql
CREATE TABLE products {
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
};
```

when written as a table constraint.

If a unique constraint refers to a group of columns, the columns are listed separated by commas:

```sql
CREATE TABLE example {
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
};
```

This specifies that the combination of values in the indicated columns is unique across the whole table, though any one of the columns need not be (and ordinarily isn’t) unique.

You can assign your own name for a unique constraint, in the usual way:

```sql
CREATE TABLE products {
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
};
```

In general, a unique constraint is violated when there are two or more rows in the table where the values of all of the columns included in the constraint are equal. However, two null values are not considered equal in this comparison. That means even in the presence of a unique constraint it is possible to store duplicate rows that contain a null value in at least one of the constrained columns. This behavior conforms to the SQL standard, but we have heard that other SQL databases might not follow this rule. So be careful when developing applications that are intended to be portable.
5.3.4. Primary Keys

Technically, a primary key constraint is simply a combination of a unique constraint and a not-null constraint. So, the following two table definitions accept the same data:

```sql
CREATE TABLE products {
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
};

CREATE TABLE products {
    product_no integer PRIMARY KEY,
    name text,
    price numeric
};
```

Primary keys can also constrain more than one column; the syntax is similar to unique constraints:

```sql
CREATE TABLE example {
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
};
```

A primary key indicates that a column or group of columns can be used as a unique identifier for rows in the table. (This is a direct consequence of the definition of a primary key. Note that a unique constraint does not, by itself, provide a unique identifier because it does not exclude null values.) This is useful both for documentation purposes and for client applications. For example, a GUI application that allows modifying row values probably needs to know the primary key of a table to be able to identify rows uniquely.

A table can have at most one primary key. (There can be any number of unique and not-null constraints, which are functionally the same thing, but only one can be identified as the primary key.) Relational database theory dictates that every table must have a primary key. This rule is not enforced by PostgreSQL, but it is usually best to follow it.

5.3.5. Foreign Keys

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integrity between two related tables.

Say you have the product table that we have used several times already:

```sql
CREATE TABLE products {
    product_no integer PRIMARY KEY,
    name text,
    price numeric
};
```
Let's also assume you have a table storing orders of those products. We want to ensure that the orders table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:

```sql
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products (product_no),  
    quantity integer  
);  
```

Now it is impossible to create orders with `product_no` entries that do not appear in the products table.

We say that in this situation the orders table is the referencing table and the products table is the referenced table. Similarly, there are referencing and referenced columns.

You can also shorten the above command to:

```sql
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products,  
    quantity integer  
);  
```

because in absence of a column list the primary key of the referenced table is used as the referenced column(s).

A foreign key can also constrain and reference a group of columns. As usual, it then needs to be written in table constraint form. Here is a contrived syntax example:

```sql
CREATE TABLE t1 (  
    a integer PRIMARY KEY,  
    b integer,  
    c integer,  
    FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)  
);  
```

Of course, the number and type of the constrained columns need to match the number and type of the referenced columns.

You can assign your own name for a foreign key constraint, in the usual way.

A table can contain more than one foreign key constraint. This is used to implement many-to-many relationships between tables. Say you have tables about products and orders, but now you want to allow one order to contain possibly many products (which the structure above did not allow). You could use this table structure:

```sql
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);  
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    shipping_address text,  
    ...  
);  
```
CREATE TABLE order_items (  
  product_no integer REFERENCES products,  
  order_id integer REFERENCES orders,  
  quantity integer,  
  PRIMARY KEY (product_no, order_id) 
);  

Notice that the primary key overlaps with the foreign keys in the last table.

We know that the foreign keys disallow creation of orders that do not relate to any products. But what if a product is removed after an order is created that references it? SQL allows you to handle that as well. Intuitively, we have a few options:

- Disallow deleting a referenced product
- Delete the orders as well
- Something else?

To illustrate this, let’s implement the following policy on the many-to-many relationship example above: when someone wants to remove a product that is still referenced by an order (via order_items), we disallow it. If someone removes an order, the order items are removed as well:

```
CREATE TABLE products (
  product_no integer PRIMARY KEY,
  name text,
  price numeric
);  
 CREATE TABLE orders (  
  order_id integer PRIMARY KEY,
  shipping_address text,
  ...  
);  
 CREATE TABLE order_items (  
  product_no integer REFERENCES products ON DELETE RESTRICT,
  order_id integer REFERENCES orders ON DELETE CASCADE,
  quantity integer,
  PRIMARY KEY (product_no, order_id)
);  
```

Restricting and cascading deletes are the two most common options. RESTRICT prevents deletion of a referenced row. NO ACTION means that if any referencing rows still exist when the constraint is checked, an error is raised; this is the default behavior if you do not specify anything. (The essential difference between these two choices is that NO ACTION allows the check to be deferred until later in the transaction, whereas RESTRICT does not.) CASCADE specifies that when a referenced row is deleted, row(s) referencing it should be automatically deleted as well. There are two other options: SET NULL and SET DEFAULT. These cause the referencing columns to be set to nulls or default values, respectively, when the referenced row is deleted. Note that these do not excuse you from observing any constraints. For example, if an action specifies SET DEFAULT but the default value would not satisfy the foreign key, the operation will fail.

Analogous to ON DELETE there is also ON UPDATE which is invoked when a referenced column is changed (updated). The possible actions are the same.
More information about updating and deleting data is in Chapter 6.

Finally, we should mention that a foreign key must reference columns that either are a primary key or form a unique constraint. If the foreign key references a unique constraint, there are some additional possibilities regarding how null values are matched. These are explained in the reference documentation for \textit{CREATE TABLE}.

### 5.4. System Columns

Every table has several \textit{system columns} that are implicitly defined by the system. Therefore, these names cannot be used as names of user-defined columns. (Note that these restrictions are separate from whether the name is a key word or not; quoting a name will not allow you to escape these restrictions.) You do not really need to be concerned about these columns, just know they exist.

\textbf{oid}

The object identifier (object ID) of a row. This column is only present if the table was created using \texttt{WITH OIDS}, or if the default\textunderscore with\textunderscore oids configuration variable was set at the time. This column is of type \texttt{oid} (same name as the column); see Section 8.16 for more information about the type.

\textbf{tableoid}

The OID of the table containing this row. This column is particularly handy for queries that select from inheritance hierarchies (see Section 5.8), since without it, it’s difficult to tell which individual table a row came from. The \texttt{tableoid} can be joined against the \texttt{oid} column of \texttt{pg\_class} to obtain the table name.

\textbf{xmin}

The identity (transaction ID) of the inserting transaction for this row version. (A row version is an individual state of a row; each update of a row creates a new row version for the same logical row.)

\textbf{cmin}

The command identifier (starting at zero) within the inserting transaction.

\textbf{xmax}

The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn’t committed yet, or that an attempted deletion was rolled back.

\textbf{cmax}

The command identifier within the deleting transaction, or zero.

\textbf{ctid}

The physical location of the row version within its table. Note that although the \texttt{ctid} can be used to locate the row version very quickly, a row’s \texttt{ctid} will change if it is updated or moved by \texttt{VACUUM FULL}. Therefore \texttt{ctid} is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

OIDs are 32-bit quantities and are assigned from a single cluster-wide counter. In a large or long-lived database, it is possible for the counter to wrap around. Hence, it is bad practice to assume that OIDs are unique, unless you take steps to ensure that this is the case. If you need to identify the rows in
a table, using a sequence generator is strongly recommended. However, OIDs can be used as well, provided that a few additional precautions are taken:

- A unique constraint should be created on the OID column of each table for which the OID will be used to identify rows. When such a unique constraint (or unique index) exists, the system takes care not to generate an OID matching an already-existing row. (Of course, this is only possible if the table contains fewer than $2^{32}$ (4 billion) rows, and in practice the table size had better be much less than that, or performance might suffer.)

- OIDs should never be assumed to be unique across tables; use the combination of tableoid and row OID if you need a database-wide identifier.

- Of course, the tables in question must be created WITH OIDS. As of PostgreSQL 8.1, WITHOUT OIDS is the default.

Transaction identifiers are also 32-bit quantities. In a long-lived database it is possible for transaction IDs to wrap around. This is not a fatal problem given appropriate maintenance procedures; see Chapter 23 for details. It is unwise, however, to depend on the uniqueness of transaction IDs over the long term (more than one billion transactions).

Command identifiers are also 32-bit quantities. This creates a hard limit of $2^{32}$ (4 billion) SQL commands within a single transaction. In practice this limit is not a problem — note that the limit is on number of SQL commands, not number of rows processed. Also, as of PostgreSQL 8.3, only commands that actually modify the database contents will consume a command identifier.

### 5.5. Modifying Tables

When you create a table and you realize that you made a mistake, or the requirements of the application change, then you can drop the table and create it again. But this is not a convenient option if the table is already filled with data, or if the table is referenced by other database objects (for instance a foreign key constraint). Therefore PostgreSQL provides a family of commands to make modifications to existing tables. Note that this is conceptually distinct from altering the data contained in the table: here we are interested in altering the definition, or structure, of the table.

You can

- Add columns,
- Remove columns,
- Add constraints,
- Remove constraints,
- Change default values,
- Change column data types,
- Rename columns,
- Rename tables.

All these actions are performed using the `ALTER TABLE` command, whose reference page contains details beyond those given here.
5.5.1. Adding a Column

To add a column, use a command like this:

```
ALTER TABLE products ADD COLUMN description text;
```

The new column is initially filled with whatever default value is given (null if you don’t specify a `DEFAULT` clause).

You can also define constraints on the column at the same time, using the usual syntax:

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

In fact all the options that can be applied to a column description in `CREATE TABLE` can be used here. Keep in mind however that the default value must satisfy the given constraints, or the `ADD` will fail. Alternatively, you can add constraints later (see below) after you’ve filled in the new column correctly.

**Tip:** Adding a column with a default requires updating each row of the table (to store the new column value). However, if no default is specified, PostgreSQL is able to avoid the physical update. So if you intend to fill the column with mostly nondefault values, it’s best to add the column with no default, insert the correct values using `UPDATE`, and then add any desired default as described below.

5.5.2. Removing a Column

To remove a column, use a command like this:

```
ALTER TABLE products DROP COLUMN description;
```

Whatever data was in the column disappears. Table constraints involving the column are dropped, too. However, if the column is referenced by a foreign key constraint of another table, PostgreSQL will not silently drop that constraint. You can authorize dropping everything that depends on the column by adding `CASCADE`:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

See Section 5.11 for a description of the general mechanism behind this.

5.5.3. Adding a Constraint

To add a constraint, the table constraint syntax is used. For example:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

To add a not-null constraint, which cannot be written as a table constraint, use this syntax:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```
Chapter 5. Data Definition

The constraint will be checked immediately, so the table data must satisfy the constraint before it can be added.

5.5.4. Removing a Constraint

To remove a constraint you need to know its name. If you gave it a name then that’s easy. Otherwise the system assigned a generated name, which you need to find out. The psql command `\d tablename` can be helpful here; other interfaces might also provide a way to inspect table details. Then the command is:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

(If you are dealing with a generated constraint name like `$2`, don’t forget that you’ll need to double-quote it to make it a valid identifier.)

As with dropping a column, you need to add `CASCADE` if you want to drop a constraint that something else depends on. An example is that a foreign key constraint depends on a unique or primary key constraint on the referenced column(s).

This works the same for all constraint types except not-null constraints. To drop a not null constraint use:

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(Recall that not-null constraints do not have names.)

5.5.5. Changing a Column’s Default Value

To set a new default for a column, use a command like this:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Note that this doesn’t affect any existing rows in the table, it just changes the default for future `INSERT` commands.

To remove any default value, use:

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

This is effectively the same as setting the default to null. As a consequence, it is not an error to drop a default where one hadn’t been defined, because the default is implicitly the null value.

5.5.6. Changing a Column’s Data Type

To convert a column to a different data type, use a command like this:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

This will succeed only if each existing entry in the column can be converted to the new type by an implicit cast. If a more complex conversion is needed, you can add a `USING` clause that specifies how to compute the new values from the old.

PostgreSQL will attempt to convert the column’s default value (if any) to the new type, as well as any constraints that involve the column. But these conversions might fail, or might produce surprising
results. It’s often best to drop any constraints on the column before altering its type, and then add back suitably modified constraints afterwards.

5.5.7. Renaming a Column

To rename a column:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. Renaming a Table

To rename a table:

```
ALTER TABLE products RENAME TO items;
```

5.6. Privileges

When you create a database object, you become its owner. By default, only the owner of an object can do anything with the object. In order to allow other users to use it, privileges must be granted. (However, users that have the superuser attribute can always access any object.)

There are several different privileges: SELECT, INSERT, UPDATE, DELETE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, and USAGE. The privileges applicable to a particular object vary depending on the object’s type (table, function, etc). For complete information on the different types of privileges supported by PostgreSQL, refer to the `GRANT` reference page. The following sections and chapters will also show you how those privileges are used.

The right to modify or destroy an object is always the privilege of the owner only.

**Note:** To change the owner of a table, index, sequence, or view, use the `ALTER TABLE` command. There are corresponding `ALTER` commands for other object types.

To assign privileges, the `GRANT` command is used. For example, if `joe` is an existing user, and `accounts` is an existing table, the privilege to update the table can be granted with:

```
GRANT UPDATE ON accounts TO joe;
```

Writing `ALL` in place of a specific privilege grants all privileges that are relevant for the object type.

The special “user” name `PUBLIC` can be used to grant a privilege to every user on the system. Also, “group” roles can be set up to help manage privileges when there are many users of a database — for details see Chapter 19.

To revoke a privilege, use the fittingly named `REVOKE` command:

```
REVOKE ALL ON accounts FROM PUBLIC;
```
The special privileges of the object owner (i.e., the right to do `DROP`, `GRANT`, `REVOKE`, etc.) are always implicit in being the owner, and cannot be granted or revoked. But the object owner can choose to revoke his own ordinary privileges, for example to make a table read-only for himself as well as others.

Ordinarily, only the object’s owner (or a superuser) can grant or revoke privileges on an object. However, it is possible to grant a privilege “with grant option”, which gives the recipient the right to grant it in turn to others. If the grant option is subsequently revoked then all who received the privilege from that recipient (directly or through a chain of grants) will lose the privilege. For details see the `GRANT` and `REVOKE` reference pages.

5.7. Schemas

A PostgreSQL database cluster contains one or more named databases. Users and groups of users are shared across the entire cluster, but no other data is shared across databases. Any given client connection to the server can access only the data in a single database, the one specified in the connection request.

Note: Users of a cluster do not necessarily have the privilege to access every database in the cluster. Sharing of user names means that there cannot be different users named, say, `joe` in two databases in the same cluster; but the system can be configured to allow `joe` access to only some of the databases.

A database contains one or more named schemas, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both `schema1` and `myschema` can contain tables named `mytable`. Unlike databases, schemas are not rigidly separated: a user can access objects in any of the schemas in the database he is connected to, if he has privileges to do so.

There are several reasons why one might want to use schemas:

• To allow many users to use one database without interfering with each other.
• To organize database objects into logical groups to make them more manageable.
• Third-party applications can be put into separate schemas so they cannot collide with the names of other objects.

Schemas are analogous to directories at the operating system level, except that schemas cannot be nested.

5.7.1. Creating a Schema

To create a schema, use the `CREATE SCHEMA` command. Give the schema a name of your choice. For example:

```sql
CREATE SCHEMA myschema;
```

To create or access objects in a schema, write a qualified name consisting of the schema name and table name separated by a dot:
This works anywhere a table name is expected, including the table modification commands and the data access commands discussed in the following chapters. (For brevity we will speak of tables only, but the same ideas apply to other kinds of named objects, such as types and functions.)

Actually, the even more general syntax

\[
\text{database.schema.table}
\]

can be used too, but at present this is just for pro forma compliance with the SQL standard. If you write a database name, it must be the same as the database you are connected to.

So to create a table in the new schema, use:

\[
\text{CREATE TABLE myschema.mytable ( ... );}
\]

To drop a schema if it’s empty (all objects in it have been dropped), use:

\[
\text{DROP SCHEMA myschema;}
\]

To drop a schema including all contained objects, use:

\[
\text{DROP SCHEMA myschema CASCADE;}
\]

See Section 5.11 for a description of the general mechanism behind this.

Often you will want to create a schema owned by someone else (since this is one of the ways to restrict the activities of your users to well-defined namespaces). The syntax for that is:

\[
\text{CREATE SCHEMA schemaname AUTHORIZATION username;}
\]

You can even omit the schema name, in which case the schema name will be the same as the user name. See Section 5.7.6 for how this can be useful.

Schema names beginning with `pg_` are reserved for system purposes and cannot be created by users.

### 5.7.2. The Public Schema

In the previous sections we created tables without specifying any schema names. By default, such tables (and other objects) are automatically put into a schema named “public”. Every new database contains such a schema. Thus, the following are equivalent:

\[
\text{CREATE TABLE products { ... };}
\]

and:

\[
\text{CREATE TABLE public.products { ... };}
\]
5.7.3. The Schema Search Path

Qualified names are tedious to write, and it’s often best not to wire a particular schema name into applications anyway. Therefore tables are often referred to by unqualified names, which consist of just the table name. The system determines which table is meant by following a search path, which is a list of schemas to look in. The first matching table in the search path is taken to be the one wanted. If there is no match in the search path, an error is reported, even if matching table names exist in other schemas in the database.

The first schema named in the search path is called the current schema. Aside from being the first schema searched, it is also the schema in which new tables will be created if the `CREATE TABLE` command does not specify a schema name.

To show the current search path, use the following command:

```
SHOW search_path;
```

In the default setup this returns:

```
  search_path
-------------
  "$user",public
```

The first element specifies that a schema with the same name as the current user is to be searched. If no such schema exists, the entry is ignored. The second element refers to the public schema that we have seen already.

The first schema in the search path that exists is the default location for creating new objects. That is the reason that by default objects are created in the public schema. When objects are referenced in any other context without schema qualification (table modification, data modification, or query commands) the search path is traversed until a matching object is found. Therefore, in the default configuration, any unqualified access again can only refer to the public schema.

To put our new schema in the path, we use:

```
SET search_path TO myschema,public;
```

(We omit the `$user` here because we have no immediate need for it.) And then we can access the table without schema qualification:

```
DROP TABLE mytable;
```

Also, since `myschema` is the first element in the path, new objects would by default be created in it.

We could also have written:

```
SET search_path TO myschema;
```

Then we no longer have access to the public schema without explicit qualification. There is nothing special about the public schema except that it exists by default. It can be dropped, too.

See also Section 9.22 for other ways to manipulate the schema search path.

The search path works in the same way for data type names, function names, and operator names as it does for table names. Data type and function names can be qualified in exactly the same way as table names. If you need to write a qualified operator name in an expression, there is a special provision: you must write

```
OPERATOR(schema,operator)
```
This is needed to avoid syntactic ambiguity. An example is:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

In practice one usually relies on the search path for operators, so as not to have to write anything so ugly as that.

### 5.7.4. Schemas and Privileges

By default, users cannot access any objects in schemas they do not own. To allow that, the owner of the schema needs to grant the `USAGE` privilege on the schema. To allow users to make use of the objects in the schema, additional privileges might need to be granted, as appropriate for the object.

A user can also be allowed to create objects in someone else’s schema. To allow that, the `CREATE` privilege on the schema needs to be granted. Note that by default, everyone has `CREATE` and `USAGE` privileges on the schema `public`. This allows all users that are able to connect to a given database to create objects in its `public` schema. If you do not want to allow that, you can revoke that privilege:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(The first “public” is the schema, the second “public” means “every user”. In the first sense it is an identifier, in the second sense it is a key word, hence the different capitalization; recall the guidelines from Section 4.1.1.)

### 5.7.5. The System Catalog Schema

In addition to `public` and user-created schemas, each database contains a `pg_catalog` schema, which contains the system tables and all the built-in data types, functions, and operators. `pg_catalog` is always effectively part of the search path. If it is not named explicitly in the path then it is implicitly searched before searching the path’s schemas. This ensures that built-in names will always be findable. However, you can explicitly place `pg_catalog` at the end of your search path if you prefer to have user-defined names override built-in names.

In PostgreSQL versions before 7.3, table names beginning with `pg_` were reserved. This is no longer true: you can create such a table name if you wish, in any non-system schema. However, it’s best to continue to avoid such names, to ensure that you won’t suffer a conflict if some future version defines a system table named the same as your table. (With the default search path, an unqualified reference to your table name would be resolved as the system table instead.) System tables will continue to follow the convention of having names beginning with `pg_`, so that they will not conflict with unqualified user-table names so long as users avoid the `pg_` prefix.

### 5.7.6. Usage Patterns

Schemas can be used to organize your data in many ways. There are a few usage patterns that are recommended and are easily supported by the default configuration:

- If you do not create any schemas then all users access the public schema implicitly. This simulates the situation where schemas are not available at all. This setup is mainly recommended when there is only a single user or a few cooperating users in a database. This setup also allows smooth transition from the non-schema-aware world.
Chapter 5. Data Definition

• You can create a schema for each user with the same name as that user. Recall that the default
search path starts with $user, which resolves to the user name. Therefore, if each user has a
separate schema, they access their own schemas by default.

If you use this setup then you might also want to revoke access to the public schema (or drop it
altogether), so users are truly constrained to their own schemas.

• To install shared applications (tables to be used by everyone, additional functions provided by third
parties, etc.), put them into separate schemas. Remember to grant appropriate privileges to allow
the other users to access them. Users can then refer to these additional objects by qualifying the
names with a schema name, or they can put the additional schemas into their search path, as they
choose.

5.7.7. Portability

In the SQL standard, the notion of objects in the same schema being owned by different users does
not exist. Moreover, some implementations do not allow you to create schemas that have a different
name than their owner. In fact, the concepts of schema and user are nearly equivalent in a database
system that implements only the basic schema support specified in the standard. Therefore, many
users consider qualified names to really consist of `username.tablename`. This is how PostgreSQL
will effectively behave if you create a per-user schema for every user.

Also, there is no concept of a `public` schema in the SQL standard. For maximum conformance to
the standard, you should not use (perhaps even remove) the `public` schema.

Of course, some SQL database systems might not implement schemas at all, or provide namespace
support by allowing (possibly limited) cross-database access. If you need to work with those systems,
then maximum portability would be achieved by not using schemas at all.

5.8. Inheritance

PostgreSQL implements table inheritance, which can be a useful tool for database designers.
(SQL:1999 and later define a type inheritance feature, which differs in many respects from the
features described here.)

Let’s start with an example: suppose we are trying to build a data model for cities. Each state has many
cities, but only one capital. We want to be able to quickly retrieve the capital city for any particular
state. This can be done by creating two tables, one for state capitals and one for cities that are not
capitals. However, what happens when we want to ask for data about a city, regardless of whether it
is a capital or not? The inheritance feature can help to resolve this problem. We define the `capitals`
table so that it inherits from `cities`:

```
CREATE TABLE cities {
    name text,
    population float,
    altitude int -- in feet
} ;
```

```
CREATE TABLE capitals {
    state char(2)
) INHERITS (cities);
```
In this case, the capitals table *inherits* all the columns of its parent table, cities. State capitals also have an extra column, state, that shows their state.

In PostgreSQL, a table can inherit from zero or more other tables, and a query can reference either all rows of a table or all rows of a table plus all of its descendant tables. The latter behavior is the default. For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500 feet:

```sql
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

Given the sample data from the PostgreSQL tutorial (see Section 2.1), this returns:

<table>
<thead>
<tr>
<th>name</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Las Vegas</td>
<td>2174</td>
</tr>
<tr>
<td>Mariposa</td>
<td>1953</td>
</tr>
<tr>
<td>Madison</td>
<td>845</td>
</tr>
</tbody>
</table>

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude over 500 feet:

```sql
SELECT name, altitude
FROM ONLY cities
WHERE altitude > 500;
```

Here the `ONLY` keyword indicates that the query should apply only to `cities`, and not any tables below `cities` in the inheritance hierarchy. Many of the commands that we have already discussed — `SELECT`, `UPDATE` and `DELETE` — support the `ONLY` keyword.

You can also write the table name with a trailing `*` to explicitly specify that descendant tables are included:

```sql
SELECT name, altitude
FROM cities*
WHERE altitude > 500;
```

Writing `*` is not necessary, since this behavior is the default (unless you have changed the setting of the `sql_inheritance` configuration option). However writing `*` might be useful to emphasize that additional tables will be searched.

In some cases you might wish to know which table a particular row originated from. There is a system column called `tableoid` in each table which can tell you the originating table:

```sql
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

which returns:
(If you try to reproduce this example, you will probably get different numeric OIDs.) By doing a join with `pg_class` you can see the actual table names:

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 and c.tableoid = p.oid;
```

which returns:

<table>
<thead>
<tr>
<th>relname</th>
<th>name</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>cities</td>
<td>Las Vegas</td>
<td>2174</td>
</tr>
<tr>
<td>cities</td>
<td>Mariposa</td>
<td>1953</td>
</tr>
<tr>
<td>capitals</td>
<td>Madison</td>
<td>845</td>
</tr>
</tbody>
</table>

Inheritance does not automatically propagate data from `INSERT` or `COPY` commands to other tables in the inheritance hierarchy. In our example, the following `INSERT` statement will fail:

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

We might hope that the data would somehow be routed to the `capitals` table, but this does not happen: `INSERT` always inserts into exactly the table specified. In some cases it is possible to redirect the insertion using a rule (see Chapter 36). However that does not help for the above case because the `cities` table does not contain the column `state`, and so the command will be rejected before the rule can be applied.

All check constraints and not-null constraints on a parent table are automatically inherited by its children. Other types of constraints (unique, primary key, and foreign key constraints) are not inherited.

A table can inherit from more than one parent table, in which case it has the union of the columns defined by the parent tables. Any columns declared in the child table’s definition are added to these. If the same column name appears in multiple parent tables, or in both a parent table and the child’s definition, then these columns are “merged” so that there is only one such column in the child table. To be merged, columns must have the same data types, else an error is raised. The merged column will have copies of all the check constraints coming from any one of the column definitions it came from, and will be marked not-null if any of them are.

Table inheritance is typically established when the child table is created, using the `INHERITS` clause of the `CREATE TABLE` statement. Alternatively, a table which is already defined in a compatible way can have a new parent relationship added, using the `INHERIT` variant of `ALTER TABLE`. To do this the new child table must already include columns with the same names and types as the columns of the parent. It must also include check constraints with the same names and check expressions as those of the parent. Similarly an inheritance link can be removed from a child using the `NO INHERIT` variant of `ALTER TABLE`. Dynamically adding and removing inheritance links like this can be useful when the inheritance relationship is being used for table partitioning (see Section 5.9).

One convenient way to create a compatible table that will later be made a new child is to use the `LIKE` clause in `CREATE TABLE`. This creates a new table with the same columns as the source table. If there are any check constraints defined on the source table, the `INCLUDING CONSTRAINTS` option to
LIKE should be specified, as the new child must have constraints matching the parent to be considered compatible.

A parent table cannot be dropped while any of its children remain. Neither can columns of child tables be dropped or altered if they are inherited from any parent tables. If you wish to remove a table and all of its descendants, one easy way is to drop the parent table with the \texttt{CASCADE} option. \texttt{ALTER TABLE} will propagate any changes in column data definitions and check constraints down the inheritance hierarchy. Again, dropping columns or constraints on parent tables is only possible when using the \texttt{CASCADE} option. \texttt{ALTER TABLE} follows the same rules for duplicate column merging and rejection that apply during \texttt{CREATE TABLE}.

\section*{5.8.1. Caveats}

Table access permissions are not automatically inherited. Therefore, a user attempting to access a parent table must either have permissions to do the operation on all its child tables as well, or must use the \texttt{ONLY} notation. When adding a new child table to an existing inheritance hierarchy, be careful to grant all the needed permissions on it.

A serious limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. This is true on both the referencing and referenced sides of a foreign key constraint. Thus, in the terms of the above example:

- If we declared \texttt{cities.name} to be \texttt{UNIQUE} or a \texttt{PRIMARY KEY}, this would not stop the \texttt{capitals} table from having rows with names duplicating rows in \texttt{cities}. And those duplicate rows would by default show up in queries from \texttt{cities}. In fact, by default \texttt{capitals} would have no unique constraint at all, and so could contain multiple rows with the same name. You could add a unique constraint to \texttt{capitals}, but this would not prevent duplication compared to \texttt{cities}.

- Similarly, if we were to specify that \texttt{cities.name REFERENCES} some other table, this constraint would not automatically propagate to \texttt{capitals}. In this case you could work around it by manually adding the same \texttt{REFERENCES} constraint to \texttt{capitals}.

- Specifying that another table’s column \texttt{REFERENCES cities(name)} would allow the other table to contain city names, but not capital names. There is no good workaround for this case.

These deficiencies will probably be fixed in some future release, but in the meantime considerable care is needed in deciding whether inheritance is useful for your problem.

\section*{5.9. Partitioning}

PostgreSQL supports basic table partitioning. This section describes why and how to implement partitioning as part of your database design.

\subsection*{5.9.1. Overview}

Partitioning refers to splitting what is logically one large table into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of
Chapter 5. Data Definition

the heavily accessed rows of the table are in a single partition or a small number of partitions. The
partitioning substitutes for leading columns of indexes, reducing index size and making it more
likely that the heavily-used parts of the indexes fit in memory.

- When queries or updates access a large percentage of a single partition, performance can be im-
proved by taking advantage of sequential scan of that partition instead of using an index and random
access reads scattered across the whole table.

- Bulk loads and deletes can be accomplished by adding or removing partitions, if that requirement
is planned into the partitioning design. ALTER TABLE is far faster than a bulk operation. It also
totally avoids the VACUUM overhead caused by a bulk DELETE.

- Seldom-used data can be migrated to cheaper and slower storage media.

The benefits will normally be worthwhile only when a table would otherwise be very large. The exact
point at which a table will benefit from partitioning depends on the application, although a rule of
thumb is that the size of the table should exceed the physical memory of the database server.

Currently, PostgreSQL supports partitioning via table inheritance. Each partition must be created as a
child table of a single parent table. The parent table itself is normally empty; it exists just to represent
the entire data set. You should be familiar with inheritance (see Section 5.8) before attempting to set
up partitioning.

The following forms of partitioning can be implemented in PostgreSQL:

Range Partitioning

The table is partitioned into “ranges” defined by a key column or set of columns, with no overlap
between the ranges of values assigned to different partitions. For example one might partition by
date ranges, or by ranges of identifiers for particular business objects.

List Partitioning

The table is partitioned by explicitly listing which key values appear in each partition.

5.9.2. Implementing Partitioning

To set up a partitioned table, do the following:

1. Create the “master” table, from which all of the partitions will inherit.

   This table will contain no data. Do not define any check constraints on this table, unless you
   intend them to be applied equally to all partitions. There is no point in defining any indexes or
   unique constraints on it, either.

2. Create several “child” tables that each inherit from the master table. Normally, these tables will
   not add any columns to the set inherited from the master.

   We will refer to the child tables as partitions, though they are in every way normal PostgreSQL
   tables.

3. Add table constraints to the partition tables to define the allowed key values in each partition.

   Typical examples would be:
   CHECK ( x = 1 )
   CHECK ( county IN ('Oxfordshire', 'Buckinghamshire', 'Warwickshire') )
   CHECK ( outletID >= 100 AND outletID < 200 )
Chapter 5. Data Definition

Ensure that the constraints guarantee that there is no overlap between the key values permitted in different partitions. A common mistake is to set up range constraints like this:

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

This is wrong since it is not clear which partition the key value 200 belongs in.

Note that there is no difference in syntax between range and list partitioning; those terms are descriptive only.

4. For each partition, create an index on the key column(s), as well as any other indexes you might want. (The key index is not strictly necessary, but in most scenarios it is helpful. If you intend the key values to be unique then you should always create a unique or primary-key constraint for each partition.)

5. Optionally, define a trigger or rule to redirect data inserted into the master table to the appropriate partition.

6. Ensure that the constraint_exclusion configuration parameter is enabled in `postgresql.conf`. Without this, queries will not be optimized as desired.

For example, suppose we are constructing a database for a large ice cream company. The company measures peak temperatures every day as well as ice cream sales in each region. Conceptually, we want a table like this:

```
CREATE TABLE measurement (  
city_id int not null, 
logdate date not null, 
peaktemp int, 
unitsales int 
);
```

We know that most queries will access just the last week’s, month’s or quarter’s data, since the main use of this table will be to prepare online reports for management. To reduce the amount of old data that needs to be stored, we decide to only keep the most recent 3 years worth of data. At the beginning of each month we will remove the oldest month’s data.

In this situation we can use partitioning to help us meet all of our different requirements for the measurements table. Following the steps outlined above, partitioning can be set up as follows:

1. The master table is the `measurement` table, declared exactly as above.

2. Next we create one partition for each active month:

```
CREATE TABLE measurement_y2006m02 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 ( ) INHERITS (measurement);
...  
CREATE TABLE measurement_y2007m11 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 ( ) INHERITS (measurement);
```

Each of the partitions are complete tables in their own right, but they inherit their definitions from the `measurement` table.

This solves one of our problems: deleting old data. Each month, all we will need to do is perform a `DROP TABLE` on the oldest child table and create a new child table for the new month’s data.

3. We must provide non-overlapping table constraints. Rather than just creating the partition tables as above, the table creation script should really be:

```
CREATE TABLE measurement_y2006m02 (  
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )  
);
```
Chapter 5. Data Definition

) INHERITS (measurement);  
CREATE TABLE measurement_y2006m03 (  
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )  
) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 (  
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )  
) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 (  
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )  
) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 (  
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )  
) INHERITS (measurement);

4. We probably need indexes on the key columns too:

CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
...
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);

We choose not to add further indexes at this time.

5. We want our application to be able to say INSERT INTO measurement ... and have the data be redirected into the appropriate partition table. We can arrange that by attaching a suitable trigger function to the master table. If data will be added only to the latest partition, we can use a very simple trigger function:

CREATE OR REPLACE FUNCTION measurement_insert_trigger()  
RETURNS TRIGGER AS $$_  
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;  
$_
LANGUAGE plpgsql;

 After creating the function, we create a trigger which calls the trigger function:

CREATE TRIGGER insert_measurement_trigger
BEFORE INSERT ON measurement
FOR EACH ROW EXECUTE PROCEDURE measurement_insert_trigger();

We must redefine the trigger function each month so that it always points to the current partition. The trigger definition does not need to be updated, however.

We might want to insert data and have the server automatically locate the partition into which the row should be added. We could do this with a more complex trigger function, for example:

CREATE OR REPLACE FUNCTION measurement_insert_trigger()  
RETURNS TRIGGER AS $$_  
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...  
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range. Fix the measurement_insert_trigger() function.';
    END IF;
END;  
$_
LANGUAGE plpgsql;

6. We might want to insert data and have the server automatically locate the partition into which the row should be added. We could do this with a more complex trigger function, for example:
Chapter 5. Data Definition

END IF;
RETURN NULL;
END;
$$
LANGUAGE plpgsql;

The trigger definition is the same as before. Note that each IF test must exactly match the CHECK constraint for its partition.

While this function is more complex than the single-month case, it doesn’t need to be updated as often, since branches can be added in advance of being needed.

Note: In practice it might be best to check the newest partition first, if most inserts go into that partition. For simplicity we have shown the trigger’s tests in the same order as in other parts of this example.

As we can see, a complex partitioning scheme could require a substantial amount of DDL. In the above example we would be creating a new partition each month, so it might be wise to write a script that generates the required DDL automatically.

5.9.3. Managing Partitions

Normally the set of partitions established when initially defining the table are not intended to remain static. It is common to want to remove old partitions of data and periodically add new partitions for new data. One of the most important advantages of partitioning is precisely that it allows this otherwise painful task to be executed nearly instantaneously by manipulating the partition structure, rather than physically moving large amounts of data around.

The simplest option for removing old data is simply to drop the partition that is no longer necessary:

DROP TABLE measurement_y2006m02;

This can very quickly delete millions of records because it doesn’t have to individually delete every record.

Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right:

ALTER TABLE measurement_y2006m02 NO INHERIT measurement;

This allows further operations to be performed on the data before it is dropped. For example, this is often a useful time to back up the data using COPY, pg_dump, or similar tools. It might also be a useful time to aggregate data into smaller formats, perform other data manipulations, or run reports.

Similarly we can add a new partition to handle new data. We can create an empty partition in the partitioned table just as the original partitions were created above:

CREATE TABLE measurement_y2008m02 (  
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01')  
) INHERITS (measurement);
Chapter 5. Data Definition

As an alternative, it is sometimes more convenient to create the new table outside the partition structure, and make it a proper partition later. This allows the data to be loaded, checked, and transformed prior to it appearing in the partitioned table:

```sql
CREATE TABLE measurement_y2008m02
    (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
    CHECK (logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01');
\copy measurement_y2008m02 from 'measurement_y2008m02'
    -- possibly some other data preparation work
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.9.4. Partitioning and Constraint Exclusion

Constraint exclusion is a query optimization technique that improves performance for partitioned tables defined in the fashion described above. As an example:

```sql
SET constraint_exclusion = on;
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

Without constraint exclusion, the above query would scan each of the partitions of the `measurement` table. With constraint exclusion enabled, the planner will examine the constraints of each partition and try to prove that the partition need not be scanned because it could not contain any rows meeting the query’s WHERE clause. When the planner can prove this, it excludes the partition from the query plan.

You can use the `EXPLAIN` command to show the difference between a plan with `constraint_exclusion` on and a plan with it off. A typical default plan for this type of table setup is:

```sql
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

```
QUERY PLAN
-----------------------------------------------------------------------------------------------
Aggregate (cost=158.66..158.68 rows=1 width=0)
    -> Append (cost=0.00..151.88 rows=2715 width=0)
        -> Seq Scan on measurement (cost=0.00..30.38 rows=543 width=0)
            Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2006m02 measurement (cost=0.00..30.38 rows=543 width=0)
            Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2006m03 measurement (cost=0.00..30.38 rows=543 width=0)
            Filter: (logdate >= '2008-01-01'::date)
        ... 
        -> Seq Scan on measurement_y2007m12 measurement (cost=0.00..30.38 rows=543 width=0)
            Filter: (logdate >= '2008-01-01'::date)
        -> Seq Scan on measurement_y2008m01 measurement (cost=0.00..30.38 rows=543 width=0)
            Filter: (logdate >= '2008-01-01'::date)
```

Some or all of the partitions might use index scans instead of full-table sequential scans, but the point here is that there is no need to scan the older partitions at all to answer this query. When we enable constraint exclusion, we get a significantly reduced plan that will deliver the same answer:

```sql
SET constraint_exclusion = on;
```
Chapter 5. Data Definition

EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';

QUERY PLAN

Aggregate (cost=63.47..63.48 rows=1 width=0)
   ->  Append (cost=0.00..60.75 rows=1086 width=0)
      ->  Seq Scan on measurement (cost=0.00..30.38 rows=543 width=0)
          Filter: (logdate > '2008-01-01'::date)
      ->  Seq Scan on measurement_y2008m01 measurement (cost=0.00..30.38 rows=543 width=0)
          Filter: (logdate > '2008-01-01'::date)

Note that constraint exclusion is driven only by CHECK constraints, not by the presence of indexes. Therefore it isn’t necessary to define indexes on the key columns. Whether an index needs to be created for a given partition depends on whether you expect that queries that scan the partition will generally scan a large part of the partition or just a small part. An index will be helpful in the latter case but not the former.

5.9.5. Alternative Partitioning Methods

A different approach to redirecting inserts into the appropriate partition table is to set up rules, instead of a trigger, on the master table. For example:

CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
   ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
   INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...

CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
   ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
   INSERT INTO measurement_y2008m01 VALUES (NEW.*);

A rule has significantly more overhead than a trigger, but the overhead is paid once per query rather than once per row, so this method might be advantageous for bulk-insert situations. In most cases, however, the trigger method will offer better performance.

Be aware that COPY ignores rules. If you want to use COPY to insert data, you’ll need to copy into the correct partition table rather than into the master. COPY does fire triggers, so you can use it normally if you use the trigger approach.

Another disadvantage of the rule approach is that there is no simple way to force an error if the set of rules doesn’t cover the insertion date; the data will silently go into the master table instead.

Partitioning can also be arranged using a \texttt{UNION ALL} view, instead of table inheritance. For example,

CREATE VIEW measurement AS
   SELECT * FROM measurement_y2006m02
UNION ALL SELECT * FROM measurement_y2006m03
...
UNION ALL SELECT * FROM measurement_y2007m11
UNION ALL SELECT * FROM measurement_y2007m12
UNION ALL SELECT * FROM measurement_y2008m01;
However, the need to recreate the view adds an extra step to adding and dropping individual partitions of the data set. In practice this method has little to recommend it compared to using inheritance.

5.9.6. Caveats

The following caveats apply to partitioned tables:

- There is no automatic way to verify that all of the \texttt{CHECK} constraints are mutually exclusive. It is safer to create code that generates partitions and creates and/or modifies associated objects than to write each by hand.

- The schemes shown here assume that the partition key column(s) of a row never change, or at least do not change enough to require it to move to another partition. An \texttt{UPDATE} that attempts to do that will fail because of the \texttt{CHECK} constraints. If you need to handle such cases, you can put suitable update triggers on the partition tables, but it makes management of the structure much more complicated.

- If you are using manual \texttt{VACUUM} or \texttt{ANALYZE} commands, don’t forget that you need to run them on each partition individually. A command like

  \texttt{ANALYZE measurement;}

  will only process the master table.

The following caveats apply to constraint exclusion:

- Constraint exclusion only works when the query’s \texttt{WHERE} clause contains constants. A parameterized query will not be optimized, since the planner cannot know which partitions the parameter value might select at run time. For the same reason, “stable” functions such as \texttt{CURRENT_DATE} must be avoided.

- Keep the partitioning constraints simple, else the planner may not be able to prove that partitions don’t need to be visited. Use simple equality conditions for list partitioning, or simple range tests for range partitioning, as illustrated in the preceding examples. A good rule of thumb is that partitioning constraints should contain only comparisons of the partitioning column(s) to constants using B-tree-indexable operators.

- All constraints on all partitions of the master table are examined during constraint exclusion, so large numbers of partitions are likely to increase query planning time considerably. Partitioning using these techniques will work well with up to perhaps a hundred partitions; don’t try to use many thousands of partitions.

5.10. Other Database Objects

Tables are the central objects in a relational database structure, because they hold your data. But they are not the only objects that exist in a database. Many other kinds of objects can be created to make the use and management of the data more efficient or convenient. They are not discussed in this chapter, but we give you a list here so that you are aware of what is possible.
5.11. Dependency Tracking

When you create complex database structures involving many tables with foreign key constraints, views, triggers, functions, etc. you will implicitly create a net of dependencies between the objects. For instance, a table with a foreign key constraint depends on the table it references. To ensure the integrity of the entire database structure, PostgreSQL makes sure that you cannot drop objects that other objects still depend on. For example, attempting to drop the products table we had considered in Section 5.3.5, with the orders table depending on it, would result in an error message such as this:

```
DROP TABLE products;
```

```
NOTICE: constraint orders_product_no_fkey on table orders depends on table products
ERROR: cannot drop table products because other objects depend on it
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

The error message contains a useful hint: if you do not want to bother deleting all the dependent objects individually, you can run

```
DROP TABLE products CASCADE;
```

and all the dependent objects will be removed. In this case, it doesn’t remove the orders table, it only removes the foreign key constraint. (If you want to check what DROP ... CASCADE will do, run DROP without CASCADE and read the NOTICE messages.)

All drop commands in PostgreSQL support specifying CASCADE. Of course, the nature of the possible dependencies varies with the type of the object. You can also write RESTRICT instead of CASCADE to get the default behavior, which is to prevent drops of objects that other objects depend on.

**Note:** According to the SQL standard, specifying either RESTRICT or CASCADE is required. No database system actually enforces that rule, but whether the default behavior is RESTRICT or CASCADE varies across systems.

**Note:** Foreign key constraint dependencies and serial column dependencies from PostgreSQL versions prior to 7.3 are not maintained or created during the upgrade process. All other dependency types will be properly created during an upgrade from a pre-7.3 database.
Chapter 6. Data Manipulation

The previous chapter discussed how to create tables and other structures to hold your data. Now it is time to fill the tables with data. This chapter covers how to insert, update, and delete table data. We also introduce ways to effect automatic data changes when certain events occur: triggers and rewrite rules. The chapter after this will finally explain how to extract your long-lost data back out of the database.

6.1. Inserting Data

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. Data is conceptually inserted one row at a time. Of course you can also insert more than one row, but there is no way to insert less than one row at a time. Even if you know only some column values, a complete row must be created.

To create a new row, use the INSERT command. The command requires the table name and a value for each of the columns of the table. For example, consider the products table from Chapter 5:

```
CREATE TABLE products {
    product_no integer,
    name text,
    price numeric
};
```

An example command to insert a row would be:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

The data values are listed in the order in which the columns appear in the table, separated by commas. Usually, the data values will be literals (constants), but scalar expressions are also allowed.

The above syntax has the drawback that you need to know the order of the columns in the table. To avoid that you can also list the columns explicitly. For example, both of the following commands have the same effect as the one above:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

Many users consider it good practice to always list the column names.

If you don’t have values for all the columns, you can omit some of them. In that case, the columns will be filled with their default values. For example:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

The second form is a PostgreSQL extension. It fills the columns from the left with as many values as are given, and the rest will be defaulted.

For clarity, you can also request default values explicitly, for individual columns or for the entire row:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```
You can insert multiple rows in a single command:

```
INSERT INTO products (product_no, name, price) VALUES
  (1, 'Cheese', 9.99),
  (2, 'Bread', 1.99),
  (3, 'Milk', 2.99);
```

**Tip:** When inserting a lot of data at the same time, considering using the `COPY` command. It is not as flexible as the `INSERT` command, but is more efficient. Refer to Section 14.4 for more information on improving bulk loading performance.

### 6.2. Updating Data

The modification of data that is already in the database is referred to as updating. You can update individual rows, all the rows in a table, or a subset of all rows. Each column can be updated separately; the other columns are not affected.

To perform an update, you need three pieces of information:

1. The name of the table and column to update,
2. The new value of the column,
3. Which row(s) to update.

Recall from Chapter 5 that SQL does not, in general, provide a unique identifier for rows. Therefore it is not necessarily possible to directly specify which row to update. Instead, you specify which conditions a row must meet in order to be updated. Only if you have a primary key in the table (no matter whether you declared it or not) can you reliably address individual rows, by choosing a condition that matches the primary key. Graphical database access tools rely on this fact to allow you to update rows individually.

For example, this command updates all products that have a price of 5 to have a price of 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

This might cause zero, one, or many rows to be updated. It is not an error to attempt an update that does not match any rows.

Let’s look at that command in detail. First is the key word `UPDATE` followed by the table name. As usual, the table name can be schema-qualified, otherwise it is looked up in the path. Next is the key word `SET` followed by the column name, an equals sign and the new column value. The new column value can be any scalar expression, not just a constant. For example, if you want to raise the price of all products by 10% you could use:

```
UPDATE products SET price = price * 1.10;
```

As you see, the expression for the new value can refer to the existing value(s) in the row. We also left out the `WHERE` clause. If it is omitted, it means that all rows in the table are updated. If it is present, only those rows that match the `WHERE` condition are updated. Note that the equals sign in the `SET` clause is an assignment while the one in the `WHERE` clause is a comparison, but this does not
create any ambiguity. Of course, the WHERE condition does not have to be an equality test. Many other operators are available (see Chapter 9). But the expression needs to evaluate to a Boolean result.

You can update more than one column in an UPDATE command by listing more than one assignment in the SET clause. For example:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Deleting Data

So far we have explained how to add data to tables and how to change data. What remains is to discuss how to remove data that is no longer needed. Just as adding data is only possible in whole rows, you can only remove entire rows from a table. In the previous section we explained that SQL does not provide a way to directly address individual rows. Therefore, removing rows can only be done by specifying conditions that the rows to be removed have to match. If you have a primary key in the table then you can specify the exact row. But you can also remove groups of rows matching a condition, or you can remove all rows in the table at once.

You use the DELETE command to remove rows; the syntax is very similar to the UPDATE command. For instance, to remove all rows from the products table that have a price of 10, use:

```
DELETE FROM products WHERE price = 10;
```

If you simply write:

```
DELETE FROM products;
```

then all rows in the table will be deleted! Caveat programmer.
Chapter 7. Queries

The previous chapters explained how to create tables, how to fill them with data, and how to manipulate that data. Now we finally discuss how to retrieve the data out of the database.

7.1. Overview

The process of retrieving or the command to retrieve data from a database is called a query. In SQL the SELECT command is used to specify queries. The general syntax of the SELECT command is

```
SELECT select_list FROM table_expression [sort_specification]
```

The following sections describe the details of the select list, the table expression, and the sort specification.

A simple kind of query has the form:

```
SELECT * FROM table1;
```

Assuming that there is a table called `table1`, this command would retrieve all rows and all columns from `table1`. (The method of retrieval depends on the client application. For example, the psql program will display an ASCII-art table on the screen, while client libraries will offer functions to extract individual values from the query result.) The select list specification `*` means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or make calculations using the columns. For example, if `table1` has columns named `a`, `b`, and `c` (and perhaps others) you can make the following query:

```
SELECT a, b + c FROM table1;
```

(assuming that `b` and `c` are of a numerical data type). See Section 7.3 for more details.

```
FROM table1 is a particularly simple kind of table expression: it reads just one table. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the SELECT command as a calculator:
```

```
SELECT 3 * 4;
```

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way:

```
SELECT random();
```

7.2. Table Expressions

A table expression computes a table. The table expression contains a FROM clause that is optionally followed by WHERE, GROUP BY, and HAVING clauses. Trivial table expressions simply refer to a table on disk, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional WHERE, GROUP BY, and HAVING clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the FROM clause. All these transforma-
tions produce a virtual table that provides the rows that are passed to the select list to compute the output rows of the query.

## 7.2.1. The FROM Clause

The **FROM Clause** derives a table from one or more other tables given in a comma-separated table reference list.

```sql
FROM table_reference [, table_reference [, ...]]
```

A table reference can be a table name (possibly schema-qualified), or a derived table such as a sub-query, a table join, or complex combinations of these. If more than one table reference is listed in the `FROM` clause they are cross-joined (see below) to form the intermediate virtual table that can then be subject to transformations by the `WHERE`, `GROUP BY`, and `HAVING` clauses and is finally the result of the overall table expression.

When a table reference names a table that is the parent of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its descendant tables, unless the key word `ONLY` precedes the table name. However, the reference produces only the columns that appear in the named table — any columns added in subtables are ignored.

Instead of writing `ONLY` before the table name, you can write `*` after the table name to explicitly specify that descendant tables are included. Writing `*` is not necessary since that behavior is the default (unless you have changed the setting of the `sql_inheritance` configuration option). However writing `*` might be useful to emphasize that additional tables will be searched.

### 7.2.1.1. Joined Tables

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. Inner, outer, and cross-joins are available.

#### Join Types

**Cross join**

```sql
T1 CROSS JOIN T2
```

For each combination of rows from `T1` and `T2`, the derived table will contain a row consisting of all columns in `T1` followed by all columns in `T2`. If the tables have N and M rows respectively, the joined table will have N * M rows.

```sql
FROM T1 CROSS JOIN T2
```

is equivalent to `FROM T1, T2`. It is also equivalent to `FROM T1 INNER JOIN T2 ON TRUE` (see below).

**Qualified joins**

```sql
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

The words **INNER** and **OUTER** are optional in all forms. **INNER** is the default; **LEFT**, **RIGHT**, and **FULL** imply an outer join.

The **join condition** is specified in the `ON` or `USING` clause, or implicitly by the word `NATURAL`. The join condition determines which rows from the two source tables are considered to “match”, as explained in detail below.
Chapter 7. Queries

The **ON** clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a **WHERE** clause. A pair of rows from \( T1 \) and \( T2 \) match if the **ON** expression evaluates to true for them.

**USING** is a shorthand notation: it takes a comma-separated list of column names, which the joined tables must have in common, and forms a join condition specifying equality of each of these pairs of columns. Furthermore, the output of a **JOIN USING** has one column for each of the equated pairs of input columns, followed by all of the other columns from each table. Thus, **USING** \((a, b, c)\) is equivalent to **ON** \((t1.a = t2.a \text{ AND } t1.b = t2.b \text{ AND } t1.c = t2.c)\) with the exception that if **ON** is used there will be two columns \(a, b, \text{ and } c\) in the result, whereas with **USING** there will be only one of each.

Finally, **NATURAL** is a shorthand form of **USING**: it forms a **USING** list consisting of exactly those column names that appear in both input tables. As with **USING**, these columns appear only once in the output table.

The possible types of qualified join are:

**INNER JOIN**

For each row \( R1 \) of \( T1 \), the joined table has a row for each row in \( T2 \) that satisfies the join condition with \( R1 \).

**LEFT OUTER JOIN**

First, an inner join is performed. Then, for each row in \( T1 \) that does not satisfy the join condition with any row in \( T2 \), a joined row is added with null values in columns of \( T2 \). Thus, the joined table unconditionally has at least one row for each row in \( T1 \).

**RIGHT OUTER JOIN**

First, an inner join is performed. Then, for each row in \( T2 \) that does not satisfy the join condition with any row in \( T1 \), a joined row is added with null values in columns of \( T1 \). This is the converse of a left join: the result table will unconditionally have a row for each row in \( T2 \).

**FULL OUTER JOIN**

First, an inner join is performed. Then, for each row in \( T1 \) that does not satisfy the join condition with any row in \( T2 \), a joined row is added with null values in columns of \( T2 \). Also, for each row of \( T2 \) that does not satisfy the join condition with any row in \( T1 \), a joined row with null values in the columns of \( T1 \) is added.

Joins of all types can be chained together or nested: either or both of \( T1 \) and \( T2 \) might be joined tables. Parentheses can be used around **JOIN** clauses to control the join order. In the absence of parentheses, **JOIN** clauses nest left-to-right.

To put this together, assume we have tables \( t1 \):

```
<table>
<thead>
<tr>
<th>num</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
</tbody>
</table>
```

and \( t2 \):

```
<table>
<thead>
<tr>
<th>num</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>------</td>
</tr>
</tbody>
</table>
```

74
then we get the following results for the various joins:

```sql
-> SELECT * FROM t1 CROSS JOIN t2;
num | name | num | value
-----+------+-----+-------
 1   | a    | 1   | xxx   
 1   | a    | 3   | yyy   
 1   | a    | 5   | zzz   
 2   | b    | 1   | xxx   
 2   | b    | 3   | yyy   
 2   | b    | 5   | zzz   
 3   | c    | 1   | xxx   
 3   | c    | 3   | yyy   
 3   | c    | 5   | zzz   
(9 rows)
```

```sql
-> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
num | name | num | value
-----+------+-----+-------
 1   | a    | 1   | xxx   
 3   | c    | 3   | yyy   
(2 rows)
```

```sql
-> SELECT * FROM t1 INNER JOIN t2 USING (num);
num | name | value
-----+------+-------
 1   | a    | xxx   
 3   | c    | yyy   
(2 rows)
```

```sql
-> SELECT * FROM t1 NATURAL INNER JOIN t2;
num | name | value
-----+------+-------
 1   | a    | xxx   
 3   | c    | yyy   
(2 rows)
```

```sql
-> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
num | name | num | value
-----+------+-----+-------
 1   | a    | 1   | xxx   
 2   | b    |     |       
 3   | c    | 3   | yyy   
(3 rows)
```

```sql
-> SELECT * FROM t1 LEFT JOIN t2 USING (num);
num | name | value
-----+------+-------
 1   | a    | xxx   
 2   | b    |       
 3   | c    | yyy   
(3 rows)
```
The join condition specified with ON can also contain conditions that do not relate directly to the join. This can prove useful for some queries but needs to be thought out carefully. For example:

```sql
-> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
```

<table>
<thead>
<tr>
<th>num</th>
<th>name</th>
<th>num</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td>xxx</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>3</td>
<td>yyy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>zzz</td>
</tr>
</tbody>
</table>

(3 rows)

7.2.1.2. Table and Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query. This is called a **table alias**.

To create a table alias, write

```
FROM table_reference AS alias
```

or

```
FROM table_reference alias
```

The **AS** key word is noise. **alias** can be any identifier.

A typical application of table aliases is to assign short identifiers to long table names to keep the join clauses readable. For example:

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num
```

The alias becomes the new name of the table reference for the current query — it is no longer possible to refer to the table by the original name. Thus:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;
```
is not valid according to the SQL standard. In PostgreSQL this will draw an error if the add_missing_from configuration variable is off (as it is by default). If it is on, an implicit table reference will be added to the FROM clause, so the query is processed as if it were written as:

```sql
SELECT * FROM my_table AS m, my_table AS my_table WHERE my_table.a > 5;
```

That will result in a cross join, which is usually not what you want.

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.:

```sql
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

Additionally, an alias is required if the table reference is a subquery (see Section 7.2.1.3).

Parentheses are used to resolve ambiguities. In the following example, the first statement assigns the alias b to the second instance of my_table, but the second statement assigns the alias to the result of the join:

```sql
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

Another form of table aliasing gives temporary names to the columns of the table, as well as the table itself:

```sql
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

If fewer column aliases are specified than the actual table has columns, the remaining columns are not renamed. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a JOIN clause, using any of these forms, the alias hides the original names within the JOIN. For example:

```sql
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

is valid SQL, but:

```sql
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

is not valid: the table alias a is not visible outside the alias c.

### 7.2.1.3. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and must be assigned a table alias name. (See Section 7.2.1.2.) For example:

```sql
FROM (SELECT * FROM table1) AS alias_name
```

This example is equivalent to `FROM table1 AS alias_name`. More interesting cases, which cannot be reduced to a plain join, arise when the subquery involves grouping or aggregation.

A subquery can also be a VALUES list:

```sql
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow')) AS names(first, last)
```
Again, a table alias is required. Assigning alias names to the columns of the VALUES list is optional, but is good practice. For more information see Section 7.7.

7.2.1.4. Table Functions

Table functions are functions that produce a set of rows, made up of either base data types (scalar types) or composite data types (table rows). They are used like a table, view, or subquery in the FROM clause of a query. Columns returned by table functions can be included in SELECT, JOIN, or WHERE clauses in the same manner as a table, view, or subquery column.

If a table function returns a base data type, the single result column is named like the function. If the function returns a composite type, the result columns get the same names as the individual attributes of the type.

A table function can be aliased in the FROM clause, but it also can be left unaliased. If a function is used in the FROM clause with no alias, the function name is used as the resulting table name.

Some examples:

```sql
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
WHERE foosubid IN (select foosubid from getfoo(foo.fooid) z
where z.fooid = foo.fooid);

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

In some cases it is useful to define table functions that can return different column sets depending on how they are invoked. To support this, the table function can be declared as returning the pseudotype record. When such a function is used in a query, the expected row structure must be specified in the query itself, so that the system can know how to parse and plan the query. Consider this example:

```sql
SELECT *
FROM dblink('dbname=mydb', 'select proname, prosrc from pg_proc')
AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

The dblink function executes a remote query (see contrib/dblink). It is declared to return record since it might be used for any kind of query. The actual column set must be specified in the calling query so that the parser knows, for example, what * should expand to.
7.2.2. The WHERE Clause

The syntax of the WHERE Clause is

```
WHERE search_condition
```

where `search_condition` is any value expression (see Section 4.2) that returns a value of type boolean.

After the processing of the FROM clause is done, each row of the derived virtual table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (that is, if the result is false or null) it is discarded. The search condition typically references at least some column of the table generated in the FROM clause; this is not required, but otherwise the WHERE clause will be fairly useless.

**Note:** The join condition of an inner join can be written either in the WHERE clause or in the JOIN clause. For example, these table expressions are equivalent:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

and:

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

or perhaps even:

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Which one of these you use is mainly a matter of style. The JOIN syntax in the FROM clause is probably not as portable to other SQL database management systems. For outer joins there is no choice in any case: they must be done in the FROM clause. An ON/USING clause of an outer join is not equivalent to a WHERE condition, because it determines the addition of rows (for unmatched input rows) as well as the removal of rows from the final result.

Here are some examples of WHERE clauses:

```
SELECT ... FROM fdt WHERE c1 > 5
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

`fdt` is the table derived in the FROM clause. Rows that do not meet the search condition of the WHERE clause are eliminated from `fdt`. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice also how `fdt` is referenced in the subqueries. Qualifying `c1` as `fdt.c1` is only necessary if `c1` is also the name of a column in the derived input table of the subquery. But qualifying the column name adds clarity even when it is not needed. This example shows how the column naming scope of an outer query extends into its inner queries.
7.2.3. The \textit{GROUP BY} and \textit{HAVING} Clauses

After passing the \textit{WHERE} filter, the derived input table might be subject to grouping, using the \textit{GROUP BY} clause, and elimination of group rows using the \textit{HAVING} clause.

\begin{verbatim}
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference]...
\end{verbatim}

The \textit{GROUP BY} Clause is used to group together those rows in a table that share the same values in all the columns listed. The order in which the columns are listed does not matter. The effect is to combine each set of rows sharing common values into one group row that is representative of all rows in the group. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups. For instance:

\begin{verbatim}
=> SELECT * FROM test1;
x | y
---+---
a | 3
b | 5
c | 2
a | 1
(4 rows)
=> SELECT x FROM test1 GROUP BY x;
x
---
a b c
(3 rows)
\end{verbatim}

In the second query, we could not have written \texttt{SELECT * FROM test1 GROUP BY x}, because there is no single value for the column \textit{y} that could be associated with each group. The grouped-by columns can be referenced in the select list since they have a single value in each group.

In general, if a table is grouped, columns that are not used in the grouping cannot be referenced except in aggregate expressions. An example with aggregate expressions is:

\begin{verbatim}
=> SELECT x, sum(y) FROM test1 GROUP BY x;
x | sum
---+-----
a | 4   
b | 5   
c | 2   
(3 rows)
\end{verbatim}

Here \texttt{sum} is an aggregate function that computes a single value over the entire group. More information about the available aggregate functions can be found in Section 9.18.

\textbf{Tip:} Grouping without aggregate expressions effectively calculates the set of distinct values in a column. This can also be achieved using the \texttt{DISTINCT} clause (see Section 7.3.3).
Here is another example: it calculates the total sales for each product (rather than the total sales on all products):

```sql
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
FROM products p LEFT JOIN sales s USING (product_id)
GROUP BY product_id, p.name, p.price;
```

In this example, the columns `product_id`, `p.name`, and `p.price` must be in the `GROUP BY` clause since they are referenced in the query select list. (Depending on how exactly the products table is set up, name and price might be fully dependent on the product ID, so the additional groupings could theoretically be unnecessary, but this is not implemented yet.) The column `s.units` does not have to be in the `GROUP BY` list since it is only used in an aggregate expression (`sum(...)`), which represents the sales of a product. For each product, the query returns a summary row about all sales of the product.

In strict SQL, `GROUP BY` can only group by columns of the source table but PostgreSQL extends this to also allow `GROUP BY` to group by columns in the select list. Grouping by value expressions instead of simple column names is also allowed.

If a table has been grouped using a `GROUP BY` clause, but then only certain groups are of interest, the `HAVING` clause can be used, much like a `WHERE` clause, to eliminate groups from a grouped table. The syntax is:

```sql
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

Expressions in the `HAVING` clause can refer both to grouped expressions and to ungrouped expressions (which necessarily involve an aggregate function).

Example:

```sql
--> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum  
---+-----
a |  4   
b |  5   
(2 rows)

--> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum  
---+-----
a |  4   
b |  5   
(2 rows)
```

Again, a more realistic example:

```sql
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
FROM products p LEFT JOIN sales s USING (product_id)
WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
GROUP BY product_id, p.name, p.price, p.cost
HAVING sum(p.price * s.units) > 5000;
```

In the example above, the `WHERE` clause is selecting rows by a column that is not grouped (the expression is only true for sales during the last four weeks), while the `HAVING` clause restricts the output to groups with total gross sales over 5000. Note that the aggregate expressions do not necessarily need to be the same in all parts of the query.
7.3. Select Lists

As shown in the previous section, the table expression in the SELECT command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the select list. The select list determines which columns of the intermediate table are actually output.

7.3.1. Select-List Items

The simplest kind of select list is * which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in Section 4.2). For instance, it could be a list of column names:

```
SELECT a, b, c FROM ...  
```

The columns names a, b, and c are either the actual names of the columns of tables referenced in the FROM clause, or the aliases given to them as explained in Section 7.2.1.2. The name space available in the select list is the same as in the WHERE clause, unless grouping is used, in which case it is the same as in the HAVING clause.

If more than one table has a column of the same name, the table name must also be given, as in:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...  
```

When working with multiple tables, it can also be useful to ask for all the columns of a particular table:

```
SELECT tbl1.*, tbl2.a FROM ...  
```

(See also Section 7.2.2.)

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each result row, with the row’s values substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they could be constant arithmetic expressions as well, for instance.

7.3.2. Column Labels

The entries in the select list can be assigned names for further processing. The “further processing” in this case is an optional sort specification and the client application (e.g., column headers for display). For example:

```
SELECT a AS value, b + c AS sum FROM ...  
```

If no output column name is specified using AS, the system assigns a default name. For simple column references, this is the name of the referenced column. For function calls, this is the name of the function. For complex expressions, the system will generate a generic name.

Note: The naming of output columns here is different from that done in the FROM clause (see Section 7.2.1.2). This pipeline will in fact allow you to rename the same column twice, but the name chosen in the select list is the one that will be passed on.
7.3.3. DISTINCT

After the select list has been processed, the result table can optionally be subject to the elimination of duplicate rows. The **DISTINCT** key word is written directly after **SELECT** to specify this:

```sql
SELECT DISTINCT select_list ...
```

(Instead of **DISTINCT** the key word **ALL** can be used to specify the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. Null values are considered equal in this comparison.

Alternatively, an arbitrary expression can determine what rows are to be considered distinct:

```sql
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

Here **expression** is an arbitrary value expression that is evaluated for all rows. A set of rows for which all the expressions are equal are considered duplicates, and only the first row of the set is kept in the output. Note that the “first row” of a set is unpredictable unless the query is sorted on enough columns to guarantee a unique ordering of the rows arriving at the **DISTINCT** filter. (**DISTINCT ON processing occurs after ORDER BY sorting.**) The **DISTINCT ON** clause is not part of the SQL standard and is sometimes considered bad style because of the potentially indeterminate nature of its results. With judicious use of **GROUP BY** and subqueries in **FROM** the construct can be avoided, but it is often the most convenient alternative.

7.4. Combining Queries

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

```sql
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

**query1** and **query2** are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```sql
query1 UNION query2 UNION query3
```

which really says

```sql
(query1 UNION query2) UNION query3
```

**UNION** effectively appends the result of **query2** to the result of **query1** (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates duplicate rows from its result, in the same way as **DISTINCT**, unless **UNION ALL** is used.
Chapter 7. Queries

INTERSECT returns all rows that are both in the result of \textit{query1} and in the result of \textit{query2}. Duplicate rows are eliminated unless \texttt{INTERSECT ALL} is used.

EXCEPT returns all rows that are in the result of \textit{query1} but not in the result of \textit{query2}. (This is sometimes called the \textit{difference} between two queries.) Again, duplicates are eliminated unless \texttt{EXCEPT ALL} is used.

In order to calculate the union, intersection, or difference of two queries, the two queries must be “union compatible”, which means that they return the same number of columns and the corresponding columns have compatible data types, as described in Section 10.5.

7.5. Sorting Rows

After a query has produced an output table (after the select list has been processed) it can optionally be sorted. If sorting is not chosen, the rows will be returned in an unspecified order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The \texttt{ORDER BY} clause specifies the sort order:

\begin{verbatim}
SELECT select_list
  FROM table_expression
  ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
    [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
\end{verbatim}

The sort expression(s) can be any expression that would be valid in the query’s select list. An example is:

\begin{verbatim}
SELECT a, b FROM table1 ORDER BY a + b, c;
\end{verbatim}

When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values. Each expression can be followed by an optional \texttt{ASC} or \texttt{DESC} keyword to set the sort direction to ascending or descending. \texttt{ASC} order is the default. Ascending order puts smaller values first, where “smaller” is defined in terms of the \texttt{<} operator. Similarly, descending order is determined with the \texttt{>} operator.\footnote{Actually, PostgreSQL uses the \textit{default B-tree operator class} for the expression’s data type to determine the sort ordering for \texttt{ASC} and \texttt{DESC}. Conventionally, data types will be set up so that the \texttt{<} and \texttt{>} operators correspond to this sort ordering, but a user-defined data type’s designer could choose to do something different.}

The \texttt{NULLS FIRST} and \texttt{NULLS LAST} options can be used to determine whether nulls appear before or after non-null values in the sort ordering. By default, null values sort as if larger than any non-null value; that is, \texttt{NULLS FIRST} is the default for \texttt{DESC} order, and \texttt{NULLS LAST} otherwise.

Note that the ordering options are considered independently for each sort column. For example \texttt{ORDER BY x, y DESC} means \texttt{ORDER BY x ASC, y DESC}, which is not the same as \texttt{ORDER BY x DESC, y DESC}.

For backwards compatibility with the SQL92 version of the standard, a sort expression can instead be the name or number of an output column, as in:

\begin{verbatim}
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
\end{verbatim}

both of which sort by the first output column. Note that an output column name has to stand alone, it’s not allowed as part of an expression — for example, this is \textit{not} correct:
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c; -- wrong

This restriction is made to reduce ambiguity. There is still ambiguity if an ORDER BY item is a simple name that could match either an output column name or a column from the table expression. The output column is used in such cases. This would only cause confusion if you use AS to rename an output column to match some other table column's name.

ORDER BY can be applied to the result of a UNION, INTERSECT, or EXCEPT combination, but in this case it is only permitted to sort by output column names or numbers, not by expressions.

7.6. LIMIT and OFFSET

LIMIT and OFFSET allow you to retrieve just a portion of the rows that are generated by the rest of the query:

```
SELECT select_list
FROM table_expression
[ ORDER BY ... ]
[ LIMIT { number | ALL } ] [ OFFSET number ]
```

If a limit count is given, no more than that many rows will be returned (but possibly less, if the query itself yields less rows). LIMIT ALL is the same as omitting the LIMIT clause.

OFFSET says to skip that many rows before beginning to return rows. OFFSET 0 is the same as omitting the OFFSET clause. If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

When using LIMIT, it is important to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query’s rows. You might be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified ORDER BY.

The query optimizer takes LIMIT into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you give for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result will give inconsistent results unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

The rows skipped by an OFFSET clause still have to be computed inside the server; therefore a large OFFSET might be inefficient.

7.7. VALUES Lists

VALUES provides a way to generate a “constant table” that can be used in a query without having to actually create and populate a table on-disk. The syntax is

```
VALUES ( expression [, ...] ) [, ...]
```

Each parenthesized list of expressions generates a row in the table. The lists must all have the same number of elements (i.e., the number of columns in the table), and corresponding entries in each
list must have compatible data types. The actual data type assigned to each column of the result is determined using the same rules as for \texttt{UNION} (see Section 10.5).

As an example:

\texttt{VALUES (1, ’one’), (2, ’two’), (3, ’three’);}

will return a table of two columns and three rows. It’s effectively equivalent to:

\begin{verbatim}
SELECT 1 AS column1, ’one’ AS column2
UNION ALL
SELECT 2, ’two’
UNION ALL
SELECT 3, ’three’;
\end{verbatim}

By default, PostgreSQL assigns the names \texttt{column1}, \texttt{column2}, etc. to the columns of a \texttt{VALUES} table. The column names are not specified by the SQL standard and different database systems do it differently, so it’s usually better to override the default names with a table alias list.

Syntactically, \texttt{VALUES} followed by expression lists is treated as equivalent to

\begin{verbatim}
SELECT select_list FROM expression
\end{verbatim}

and can appear anywhere a \texttt{SELECT} can. For example, you can use it as an arm of a \texttt{UNION}, or attach a \texttt{sort_specification} (\texttt{ORDER BY}, \texttt{LIMIT}, and/or \texttt{OFFSET}) to it. \texttt{VALUES} is most commonly used as the data source in an \texttt{INSERT} command, and next most commonly as a subquery.

For more information see \texttt{VALUES}.
Chapter 8. Data Types

PostgreSQL has a rich set of native data types available to users. Users can add new types to PostgreSQL using the `CREATE TYPE` command.

Table 8-1 shows all the built-in general-purpose data types. Most of the alternative names listed in the “Aliases” column are the names used internally by PostgreSQL for historical reasons. In addition, some internally used or deprecated types are available, but they are not listed here.

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigint</td>
<td>int8</td>
<td>signed eight-byte integer</td>
</tr>
<tr>
<td>bigserial</td>
<td>serial8</td>
<td>autoincrementing eight-byte integer</td>
</tr>
<tr>
<td>bit [ (n) ]</td>
<td></td>
<td>fixed-length bit string</td>
</tr>
<tr>
<td>bit varying [ (n) ]</td>
<td>varbit</td>
<td>variable-length bit string</td>
</tr>
<tr>
<td>boolean</td>
<td>bool</td>
<td>logical Boolean (true/false)</td>
</tr>
<tr>
<td>box</td>
<td></td>
<td>rectangular box in the plane</td>
</tr>
<tr>
<td>bytea</td>
<td></td>
<td>binary data (“byte array”)</td>
</tr>
<tr>
<td>character varying [ (n) ]</td>
<td>varchar [ (n) ]</td>
<td>variable-length character string</td>
</tr>
<tr>
<td>character [ (n) ]</td>
<td>char [ (n) ]</td>
<td>fixed-length character string</td>
</tr>
<tr>
<td>cidr</td>
<td></td>
<td>IPV4 or IPV6 network address</td>
</tr>
<tr>
<td>circle</td>
<td></td>
<td>circle in the plane</td>
</tr>
<tr>
<td>date</td>
<td></td>
<td>calendar date (year, month, day)</td>
</tr>
<tr>
<td>double precision</td>
<td>float8</td>
<td>double precision floating-point number</td>
</tr>
<tr>
<td>inet</td>
<td></td>
<td>IPV4 or IPV6 host address</td>
</tr>
<tr>
<td>integer</td>
<td>int,int4</td>
<td>signed four-byte integer</td>
</tr>
<tr>
<td>interval [ (p) ]</td>
<td></td>
<td>time span</td>
</tr>
<tr>
<td>line</td>
<td></td>
<td>infinite line in the plane</td>
</tr>
<tr>
<td>lseg</td>
<td></td>
<td>line segment in the plane</td>
</tr>
<tr>
<td>macaddr</td>
<td></td>
<td>MAC address</td>
</tr>
<tr>
<td>money</td>
<td></td>
<td>currency amount</td>
</tr>
<tr>
<td>numeric [ (p, s) ]</td>
<td>decimal [ (p, s) ]</td>
<td>exact numeric of selectable precision</td>
</tr>
<tr>
<td>path</td>
<td></td>
<td>geometric path in the plane</td>
</tr>
<tr>
<td>point</td>
<td></td>
<td>geometric point in the plane</td>
</tr>
<tr>
<td>polygon</td>
<td></td>
<td>closed geometric path in the plane</td>
</tr>
<tr>
<td>real</td>
<td>float4</td>
<td>single precision floating-point number</td>
</tr>
<tr>
<td>smallint</td>
<td>int2</td>
<td>signed two-byte integer</td>
</tr>
</tbody>
</table>
Chapter 8. Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>serial4</td>
<td>autoincrementing four-byte integer</td>
</tr>
<tr>
<td>text</td>
<td></td>
<td>variable-length character string</td>
</tr>
<tr>
<td>time</td>
<td></td>
<td>time of day</td>
</tr>
<tr>
<td>time with time zone</td>
<td></td>
<td>time of day, including time zone</td>
</tr>
<tr>
<td>timestamp</td>
<td></td>
<td>date and time</td>
</tr>
<tr>
<td>timestamp with time zone</td>
<td></td>
<td>date and time, including time zone</td>
</tr>
<tr>
<td>tsquery</td>
<td></td>
<td>text search query</td>
</tr>
<tr>
<td>tsvector</td>
<td></td>
<td>text search document</td>
</tr>
<tr>
<td>txid_snapshot</td>
<td></td>
<td>user-level transaction ID snapshot</td>
</tr>
<tr>
<td>uuid</td>
<td></td>
<td>universally unique identifier</td>
</tr>
<tr>
<td>xml</td>
<td></td>
<td>XML data</td>
</tr>
</tbody>
</table>

Compatibility: The following types (or spellings thereof) are specified by SQL: bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (with or without time zone), timestamp (with or without time zone), xml.

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL, such as geometric paths, or have several possibilities for formats, such as the date and time types. Some of the input and output functions are not invertible. That is, the result of an output function might lose accuracy when compared to the original input.

8.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and selectable-precision decimals. Table 8-2 lists the available types.

Table 8-2. Numeric Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallint</td>
<td>2 bytes</td>
<td>small-range integer</td>
<td>-32768 to +32767</td>
</tr>
<tr>
<td>integer</td>
<td>4 bytes</td>
<td>usual choice for integer</td>
<td>-2147483648 to +2147483647</td>
</tr>
<tr>
<td>bigint</td>
<td>8 bytes</td>
<td>large-range integer</td>
<td>-9223372036854775808 to 9223372036854775807</td>
</tr>
</tbody>
</table>
The syntax of constants for the numeric types is described in Section 4.1.2. The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Chapter 9 for more information. The following sections describe the types in detail.

8.1.1. Integer Types

The types `smallint`, `integer`, and `bigint` store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

The type `integer` is the usual choice, as it offers the best balance between range, storage size, and performance. The `smallint` type is generally only used if disk space is at a premium. The `bigint` type should only be used if the `integer` range is not sufficient, because the latter is definitely faster.

The `bigint` type might not function correctly on all platforms, since it relies on compiler support for eight-byte integers. On a machine without such support, `bigint` acts the same as `integer` (but still takes up eight bytes of storage). However, we are not aware of any reasonable platform where this is actually the case.

SQL only specifies the integer types `integer` (or `int`), `smallint`, and `bigint`. The type names `int2`, `int4`, and `int8` are extensions, which are shared with various other SQL database systems.

8.1.2. Arbitrary Precision Numbers

The type `numeric` can store numbers with up to 1000 digits of precision and perform calculations exactly. It is especially recommended for storing monetary amounts and other quantities where exactness is required. However, arithmetic on `numeric` values is very slow compared to the integer types, or to the floating-point types described in the next section.

In what follows we use these terms: The `scale` of a `numeric` is the count of decimal digits in the fractional part, to the right of the decimal point. The `precision` of a `numeric` is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the maximum precision and the maximum scale of a `numeric` column can be configured. To declare a column of type `numeric` use the syntax:

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal</td>
<td>variable</td>
<td>user-specified precision, exact</td>
<td>no limit</td>
</tr>
<tr>
<td>numeric</td>
<td>variable</td>
<td>user-specified precision, exact</td>
<td>no limit</td>
</tr>
<tr>
<td>real</td>
<td>4 bytes</td>
<td>variable-precision, inexact</td>
<td>6 decimal digits</td>
</tr>
<tr>
<td>double precision</td>
<td>8 bytes</td>
<td>variable-precision, inexact</td>
<td>15 decimal digits</td>
</tr>
<tr>
<td>serial</td>
<td>4 bytes</td>
<td>autoincrementing integer</td>
<td>1 to 2147483647</td>
</tr>
<tr>
<td>bigserial</td>
<td>8 bytes</td>
<td>large autoincrementing integer</td>
<td>1 to 9223372036854775807</td>
</tr>
</tbody>
</table>
Chapter 8. Data Types

NUMERIC(\textit{precision, scale})

The precision must be positive, the scale zero or positive. Alternatively:

NUMERIC(\textit{precision})

selects a scale of 0. Specifying:

NUMERIC

without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas numeric columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. We find this a bit useless. If you’re concerned about portability, always specify the precision and scale explicitly.)

If the scale of a value to be stored is greater than the declared scale of the column, the system will round the value to the specified number of fractional digits. Then, if the number of digits to the left of the decimal point exceeds the declared precision minus the declared scale, an error is raised.

Numeric values are physically stored without any extra leading or trailing zeroes. Thus, the declared precision and scale of a column are maximums, not fixed allocations. (In this sense the numeric type is more akin to \textit{varchar(\textit{n})} than to \textit{char(\textit{n})}.) The actual storage requirement is two bytes for each group of four decimal digits, plus five to eight bytes overhead.

In addition to ordinary numeric values, the \textit{numeric} type allows the special value NaN, meaning “not-a-number”. Any operation on NaN yields another NaN. When writing this value as a constant in a SQL command, you must put quotes around it, for example \texttt{UPDATE table SET x = ‘NaN’}. On input, the string \texttt{NaN} is recognized in a case-insensitive manner.

\textbf{Note:} In most implementations of the “not-a-number” concept, NaN is not considered equal to any other numeric value (including NaN). In order to allow numeric values to be sorted and used in tree-based indexes, PostgreSQL treats NaN values as equal, and greater than all non-NaN values.

The types \textit{decimal} and \textit{numeric} are equivalent. Both types are part of the SQL standard.

8.1.3. Floating-Point Types

The data types \textit{real} and \textit{double precision} are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed further here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the \textit{numeric} type instead.
• If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.

• Comparing two floating-point values for equality might or might not work as expected.

On most platforms, the `real` type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The `double precision` type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding might take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

In addition to ordinary numeric values, the floating-point types have several special values:

- `Infinity`
- `-Infinity`
- `NaN`

These represent the IEEE 754 special values “infinity”, “negative infinity”, and “not-a-number”, respectively. (On a machine whose floating-point arithmetic does not follow IEEE 754, these values will probably not work as expected.) When writing these values as constants in a SQL command, you must put quotes around them, for example `UPDATE table SET x = 'Infinity'`. On input, these strings are recognized in a case-insensitive manner.

**Note:** PostgreSQL treats `NaN` values as equal, and greater than all non-`NaN` values.

PostgreSQL also supports the SQL-standard notations `float` and `float(p)` for specifying inexact numeric types. Here, `p` specifies the minimum acceptable precision in binary digits. PostgreSQL accepts `float(1)` to `float(24)` as selecting the `real` type, while `float(25)` to `float(53)` select `double precision`. Values of `p` outside the allowed range draw an error. `float` with no precision specified is taken to mean `double precision`.

**Note:** Prior to PostgreSQL 7.4, the precision in `float(p)` was taken to mean so many decimal digits. This has been corrected to match the SQL standard, which specifies that the precision is measured in binary digits. The assumption that `real` and `double precision` have exactly 24 and 53 bits in the mantissa respectively is correct for IEEE-standard floating point implementations. On non-IEEE platforms it might be off a little, but for simplicity the same ranges of `p` are used on all platforms.

### 8.1.4. Serial Types

The data types `serial` and `bigserial` are not true types, but merely a notational convenience for setting up unique identifier columns (similar to the `AUTO_INCREMENT` property supported by some other databases). In the current implementation, specifying:

```sql
CREATE TABLE tablename (  
    colname SERIAL  
) ;
```
is equivalent to specifying:

```sql
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (  
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

Thus, we have created an integer column and arranged for its default values to be assigned from a sequence generator. A `NOT NULL` constraint is applied to ensure that a null value cannot be explicitly inserted, either. (In most cases you would also want to attach a `UNIQUE` or `PRIMARY KEY` constraint to prevent duplicate values from being inserted by accident, but this is not automatic.) Lastly, the sequence is marked as “owned by” the column, so that it will be dropped if the column or table is dropped.

**Note:** Prior to PostgreSQL 7.3, `serial` implied `UNIQUE`. This is no longer automatic. If you wish a serial column to be in a unique constraint or a primary key, it must now be specified, same as with any other data type.

To insert the next value of the sequence into the `serial` column, specify that the `serial` column should be assigned its default value. This can be done either by excluding the column from the list of columns in the `INSERT` statement, or through the use of the `DEFAULT` key word.

The type names `serial` and `serial4` are equivalent: both create integer columns. The type names `bigserial` and `serial8` work just the same way, except that they create a `bigint` column. `bigserial` should be used if you anticipate the use of more than $2^{31}$ identifiers over the lifetime of the table.

The sequence created for a `serial` column is automatically dropped when the owning column is dropped. You can drop the sequence without dropping the column, but this will force removal of the column default expression.

### 8.2. Monetary Types

The `money` type stores a currency amount with a fixed fractional precision; see Table 8-3. Input is accepted in a variety of formats, including integer and floating-point literals, as well as “typical” currency formatting, such as ‘$1,000.00’. Output is generally in the latter form but depends on the locale. Non-quoted numeric values can be converted to `money` by casting the numeric value to `text` and then `money`:

```sql
SELECT 1234::text::money;
```

There is no simple way of doing the reverse in a locale-independent manner, namely casting a `money` value to a numeric type. If you know the currency symbol and thousands separator you can use `regexp_replace()`:

```sql
SELECT regexp_replace('52093.89':money::text, '[$,]', ',', 'g')::numeric;
```
Since the output of this data type is locale-sensitive, it may not work to load `money` data into a database that has a different setting of `lc_monetary`. To avoid problems, before restoring a dump make sure `lc_monetary` has the same or equivalent value as in the database that was dumped.

**Table 8-3. Monetary Types**

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>money</td>
<td>8 bytes</td>
<td>currency amount</td>
<td>-92233720368547758.08 to +92233720368547758.07</td>
</tr>
</tbody>
</table>

**8.3. Character Types**

**Table 8-4. Character Types**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>character varying((n)), varchar((n))</td>
<td>variable-length with limit</td>
</tr>
<tr>
<td>character((n)), char((n))</td>
<td>fixed-length, blank padded</td>
</tr>
<tr>
<td>text</td>
<td>variable unlimited length</td>
</tr>
</tbody>
</table>

Table 8-4 shows the general-purpose character types available in PostgreSQL.

SQL defines two primary character types: `character varying\((n)\)` and `character\((n)\)`, where \(n\) is a positive integer. Both of these types can store strings up to \(n\) characters in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type `character` will be space-padded; values of type `character varying` will simply store the shorter string.

If one explicitly casts a value to `character varying\((n)\)` or `character\((n)\)`, then an over-length value will be truncated to \(n\) characters without raising an error. (This too is required by the SQL standard.)

The notations `varchar\((n)\)` and `char\((n)\)` are aliases for `character varying\((n)\)` and `character\((n)\)`, respectively. `character` without length specifier is equivalent to `character\(1\)`.

If `character varying` is used without length specifier, the type accepts strings of any size. The latter is a PostgreSQL extension.

In addition, PostgreSQL provides the `text` type, which stores strings of any length. Although the type `text` is not in the SQL standard, several other SQL database management systems have it as well.

Values of type `character` are physically padded with spaces to the specified width \(n\), and are stored and displayed that way. However, the padding spaces are treated as semantically insignificant. Trailing spaces are disregarded when comparing two values of type `character`, and they will be removed when converting a `character` value to one of the other string types. Note that trailing spaces are not semantically significant in `character varying` and `text` values.

The storage requirement for a short string (up to 126 bytes) is 1 byte plus the actual string, which
includes the space padding in the case of character. Longer strings have 4 bytes overhead instead of 1. Long strings are compressed by the system automatically, so the physical requirement on disk might be less. Very long values are also stored in background tables so that they do not interfere with rapid access to shorter column values. In any case, the longest possible character string that can be stored is about 1 GB. (The maximum value that will be allowed for \( n \) in the data type declaration is less than that. It wouldn’t be very useful to change this because with multibyte character encodings the number of characters and bytes can be quite different anyway. If you desire to store long strings with no specific upper limit, use text or character varying without a length specifier, rather than making up an arbitrary length limit.)

**Tip:** There are no performance differences between these three types, apart from increased storage size when using the blank-padded type, and a few extra cycles to check the length when storing into a length-constrained column. While character\((n)\) has performance advantages in some other database systems, it has no such advantages in PostgreSQL. In most situations text or character varying should be used instead.

Refer to Section 4.1.2.1 for information about the syntax of string literals, and to Chapter 9 for information about available operators and functions. The database character set determines the character set used to store textual values; for more information on character set support, refer to Section 22.2.

**Example 8-1. Using the character types**

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1
   a | char_length
 ----+-------------
   ok | 2

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good');
INSERT INTO test2 VALUES ('too long');
ERROR: value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long':varchar(5)); -- explicit truncation
SELECT b, char_length(b) FROM test2;
   b | char_length
 ----+-------------
   ok | 2
   good | 5
   too | 5

1 The `char_length` function is discussed in Section 9.4.
```

There are two other fixed-length character types in PostgreSQL, shown in Table 8-5. The name type exists only for storage of identifiers in the internal system catalogs and is not intended for use by the general user. Its length is currently defined as 64 bytes (63 usable characters plus terminator) but should be referenced using the constant NAMEDATALEN. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length might change in a future release. The type "char" (note the quotes) is different from char\((1)\) in that it only uses one byte of storage. It is internally used in the system catalogs as a poor-man’s enumeration type.
8.4. Binary Data Types

The `bytea` data type allows storage of binary strings; see Table 8-6.

Table 8-6. Binary Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytea</td>
<td>1 or 4 bytes plus the actual binary string</td>
<td>variable-length binary string</td>
</tr>
</tbody>
</table>

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from character strings by two characteristics: First, binary strings specifically allow storing octets of value zero and other “non-printable” octets (usually, octets outside the range 32 to 126). Character strings disallow zero octets, and also disallow any other octet values and sequences of octet values that are invalid according to the database’s selected character set encoding. Second, operations on binary strings process the actual bytes, whereas the processing of character strings depends on locale settings. In short, binary strings are appropriate for storing data that the programmer thinks of as “raw bytes”, whereas character strings are appropriate for storing text.

When entering `bytea` values, octets of certain values must be escaped (but all octet values can be escaped) when used as part of a string literal in an SQL statement. In general, to escape an octet, it is converted into the three-digit octal number equivalent of its decimal octet value, and preceded by two backslashes. Table 8-7 shows the characters that must be escaped, and gives the alternative escape sequences where applicable.

Table 8-7. `bytea` Literal Escaped Octets

<table>
<thead>
<tr>
<th>Decimal Octet Value</th>
<th>Description</th>
<th>Escaped Input Representation</th>
<th>Example</th>
<th>Output Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero octet</td>
<td><code>E'\000'</code></td>
<td><code>SELECT E'\000':bytea;</code></td>
<td>\000</td>
</tr>
<tr>
<td>39</td>
<td>single quote</td>
<td><code>&quot; or E'\047'</code></td>
<td><code>SELECT E&quot;':bytea;</code></td>
<td>'</td>
</tr>
<tr>
<td>92</td>
<td>backslash</td>
<td><code>E''\134'</code> or <code>E'\134'</code></td>
<td><code>SELECT E''':bytea</code></td>
<td>\</td>
</tr>
<tr>
<td>0 to 31 and 127 to 255</td>
<td>“non-printable” octets</td>
<td><code>E'\xxx'</code> (octal value)</td>
<td><code>SELECT E'\001':bytea;</code></td>
<td>\001</td>
</tr>
</tbody>
</table>

The requirement to escape “non-printable” octets actually varies depending on locale settings. In some
instances you can get away with leaving them unescaped. Note that the result in each of the examples in Table 8-7 was exactly one octet in length, even though the output representation of the zero octet and backslash are more than one character.

The reason that you have to write so many backslashes, as shown in Table 8-7, is that an input string written as a string literal must pass through two parse phases in the PostgreSQL server. The first backslash of each pair is interpreted as an escape character by the string-literal parser (assuming escape string syntax is used) and is therefore consumed, leaving the second backslash of the pair. (Dollar-quoted strings can be used to avoid this level of escaping.) The remaining backslash is then recognized by the bytea input function as starting either a three digit octal value or escaping another backslash. For example, a string literal passed to the server as E'\001' becomes \001 after passing through the escape string parser. The \001 is then sent to the bytea input function, where it is converted to a single octet with a decimal value of 1. Note that the single-quote character is not treated specially by bytea, so it follows the normal rules for string literals. (See also Section 4.1.2.1.) Bytea octets are also escaped in the output. In general, each “non-printable” octet is converted into its equivalent three-digit octal value and preceded by one backslash. Most “printable” octets are represented by their standard representation in the client character set. The octet with decimal value 92 (backslash) has a special alternative output representation. Details are in Table 8-8.

Table 8-8. bytea Output Escaped Octets

<table>
<thead>
<tr>
<th>Decimal Octet Value</th>
<th>Description</th>
<th>Escaped Output Representation</th>
<th>Example</th>
<th>Output Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
<td>backslash</td>
<td>\ \</td>
<td>SELECT E'\134'::bytea;</td>
<td>\ \</td>
</tr>
<tr>
<td>0 to 31 and 127 to 255</td>
<td>“non-printable” octets</td>
<td>\xxx (octal value)</td>
<td>SELECT E'\001'::bytea;</td>
<td>\001</td>
</tr>
<tr>
<td>32 to 126</td>
<td>“printable” octets</td>
<td>client character set representation</td>
<td>SELECT E'\176'::bytea;</td>
<td>~</td>
</tr>
</tbody>
</table>

Depending on the front end to PostgreSQL you use, you might have additional work to do in terms of escaping and unescaping bytea strings. For example, you might also have to escape line feeds and carriage returns if your interface automatically translates these.

The SQL standard defines a different binary string type, called BLOB or BINARY LARGE OBJECT. The input format is different from bytea, but the provided functions and operators are mostly the same.

8.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types, shown in Table 8-9. The operations available on these data types are described in Section 9.9.

Table 8-9. Date/Time Types
<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
<th>Low Value</th>
<th>High Value</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp [(p)] [without time zone]</td>
<td>8 bytes</td>
<td>both date and time</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 microsecond / 14 digits</td>
</tr>
<tr>
<td>timestamp [(p)] [with time zone]</td>
<td>8 bytes</td>
<td>both date and time, with time zone</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 microsecond / 14 digits</td>
</tr>
<tr>
<td>interval [(p)]</td>
<td>12 bytes</td>
<td>time intervals</td>
<td>-178000000 years</td>
<td>178000000 years</td>
<td>1 microsecond / 14 digits</td>
</tr>
<tr>
<td>date</td>
<td>4 bytes</td>
<td>dates only</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 day</td>
</tr>
<tr>
<td>time [(p)] [without time zone]</td>
<td>8 bytes</td>
<td>times of day only</td>
<td>00:00:00</td>
<td>24:00:00</td>
<td>1 microsecond / 14 digits</td>
</tr>
<tr>
<td>time [(p)] [with time zone]</td>
<td>12 bytes</td>
<td>times of day only, with time zone</td>
<td>00:00:00+1459</td>
<td>24:00:00-1459</td>
<td>1 microsecond / 14 digits</td>
</tr>
</tbody>
</table>

**Note:** Prior to PostgreSQL 7.3, writing just timestamp was equivalent to timestamp with time zone. This was changed for SQL compliance.

time, timestamp, and interval accept an optional precision value p which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of p is from 0 to 6 for the timestamp and interval types.

**Note:** When timestamp values are stored as double precision floating-point numbers (currently the default), the effective limit of precision might be less than 6. Timestamp values are stored as seconds before or after midnight 2000-01-01. Microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. When timestamp values are stored as eight-byte integers (a compile-time option), microsecond precision is available over the full range of values. However eight-byte integer timestamps have a more limited range of dates than shown above: from 4713 BC up to 294276 AD. The same compile-time option also determines whether time and interval values are stored as floating-point or eight-byte integers. In the floating-point case, large interval values degrade in precision as the size of the interval increases.

For the time types, the allowed range of p is from 0 to 6 when eight-byte integer storage is used, or from 0 to 10 when floating-point storage is used.

The type time with time zone is defined by the SQL standard, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of date, time, timestamp without time zone, and timestamp with time zone should provide a complete range of date/time functionality required by any application.

The types abstime and reltime are lower precision types which are used internally. You are discouraged from using these types in new applications and are encouraged to move any old ones over when appropriate. Any or all of these internal types might disappear in a future release.
8.5.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. For some formats, ordering of month, day, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields. Set the DateStyle parameter to MDY to select month-day-year interpretation, DMY to select day-month-year interpretation, or YMD to select year-month-day interpretation.

PostgreSQL is more flexible in handling date/time input than the SQL standard requires. See Appendix B for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. Refer to Section 4.1.2.5 for more information. SQL requires the following syntax

type [ (p) ] 'value'

where \( p \) in the optional precision specification is an integer corresponding to the number of fractional digits in the seconds field. Precision can be specified for time, timestamp, and interval types. The allowed values are mentioned above. If no precision is specified in a constant specification, it defaults to the precision of the literal value.

8.5.1.1. Dates

Table 8-10 shows some possible inputs for the date type.

Table 8-10. Date Input

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 8, 1999</td>
<td>unambiguous in any datestyle input mode</td>
</tr>
<tr>
<td>1999-01-08</td>
<td>ISO 8601; January 8 in any mode (recommended format)</td>
</tr>
<tr>
<td>1/8/1999</td>
<td>January 8 in MDY mode; August 1 in DMY mode</td>
</tr>
<tr>
<td>1/18/1999</td>
<td>January 18 in MDY mode; rejected in other modes</td>
</tr>
<tr>
<td>01/02/03</td>
<td>January 2, 2003 in MDY mode; February 1, 2003 in DMY mode; February 3, 2001 in YMD mode</td>
</tr>
<tr>
<td>1999-Jan-08</td>
<td>January 8 in any mode</td>
</tr>
<tr>
<td>Jan-08-1999</td>
<td>January 8 in any mode</td>
</tr>
<tr>
<td>08-Jan-1999</td>
<td>January 8 in any mode</td>
</tr>
<tr>
<td>99-Jan-08</td>
<td>January 8 in YMD mode, else error</td>
</tr>
<tr>
<td>08-Jan-99</td>
<td>January 8, except error in YMD mode</td>
</tr>
<tr>
<td>Jan-08-99</td>
<td>January 8, except error in YMD mode</td>
</tr>
<tr>
<td>19990108</td>
<td>ISO 8601; January 8, 1999 in any mode</td>
</tr>
<tr>
<td>990108</td>
<td>ISO 8601; January 8, 1999 in any mode</td>
</tr>
<tr>
<td>1999.008</td>
<td>year and day of year</td>
</tr>
<tr>
<td>12451187</td>
<td>Julian day</td>
</tr>
<tr>
<td>January 8, 99 BC</td>
<td>year 99 before the Common Era</td>
</tr>
</tbody>
</table>
8.5.1.2. Times

The time-of-day types are time [ (p) ] without time zone and time [ (p) ] with time zone. Writing just time is equivalent to time without time zone.

Valid input for these types consists of a time of day followed by an optional time zone. (See Table 8-11 and Table 8-12.) If a time zone is specified in the input for time without time zone, it is silently ignored. You can also specify a date but it will be ignored, except when you use a time zone name that involves a daylight-savings rule, such as America/New_York. In this case specifying the date is required in order to determine whether standard or daylight-savings time applies. The appropriate time zone offset is recorded in the time with time zone value.

Table 8-11. Time Input

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04:05:06.789</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05:06</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>040506</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05 AM</td>
<td>same as 04:05; AM does not affect value</td>
</tr>
<tr>
<td>04:05 PM</td>
<td>same as 16:05; input hour must be &lt;= 12</td>
</tr>
<tr>
<td>04:05:06.789-8</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05:06-08:00</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05-08:00</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>040506-08</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05:06 PST</td>
<td>time zone specified by abbreviation</td>
</tr>
<tr>
<td>2003-04-12 04:05:06 America/New_York</td>
<td>time zone specified by full name</td>
</tr>
</tbody>
</table>

Table 8-12. Time Zone Input

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PST</td>
<td>Abbreviation (for Pacific Standard Time)</td>
</tr>
<tr>
<td>America/New_York</td>
<td>Full time zone name</td>
</tr>
<tr>
<td>PST8PDT</td>
<td>POSIX-style time zone specification</td>
</tr>
<tr>
<td>-8:00</td>
<td>ISO-8601 offset for PST</td>
</tr>
<tr>
<td>-800</td>
<td>ISO-8601 offset for PST</td>
</tr>
<tr>
<td>-8</td>
<td>ISO-8601 offset for PST</td>
</tr>
<tr>
<td>zulu</td>
<td>Military abbreviation for UTC</td>
</tr>
<tr>
<td>z</td>
<td>Short form of zulu</td>
</tr>
</tbody>
</table>

Refer to Section 8.5.3 for more information on how to specify time zones.

8.5.1.3. Time Stamps

Valid input for the time stamp types consists of a concatenation of a date and a time, followed by an optional time zone, followed by an optional AD or BC. (Alternatively, AD/BC can appear before the
time zone, but this is not the preferred ordering.) Thus:

1999-01-08 04:05:06

and:

1999-01-08 04:05:06 -8:00

are valid values, which follow the ISO 8601 standard. In addition, the wide-spread format:

January 8 04:05:06 1999 PST

is supported.

The SQL standard differentiates \texttt{timestamp without time zone} and \texttt{timestamp with time zone} literals by the presence of a “+” or “-”. Hence, according to the standard,

\texttt{TIMESTAMP '2004-10-19 10:23:54'}

is a \texttt{timestamp without time zone}, while

\texttt{TIMESTAMP '2004-10-19 10:23:54+02'}

is a \texttt{timestamp with time zone}. PostgreSQL never examines the content of a literal string before determining its type, and therefore will treat both of the above as \texttt{timestamp without time zone}. To ensure that a literal is treated as \texttt{timestamp with time zone}, give it the correct explicit type:

\texttt{TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'}

In a literal that has been decided to be \texttt{timestamp without time zone}, PostgreSQL will silently ignore any time zone indication. That is, the resulting value is derived from the date/time fields in the input value, and is not adjusted for time zone.

For \texttt{timestamp with time zone}, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system’s timezone parameter, and is converted to UTC using the offset for the timezone zone.

When a \texttt{timestamp with time zone} value is output, it is always converted from UTC to the current timezone zone, and displayed as local time in that zone. To see the time in another time zone, either change timezone or use the AT TIME ZONE construct (see Section 9.9.3).

Conversions between \texttt{timestamp without time zone} and \texttt{timestamp with time zone} normally assume that the \texttt{timestamp without time zone} value should be taken or given as timezone local time. A different zone reference can be specified for the conversion using AT TIME ZONE.

\subsection{8.5.1.4. Intervals}

\texttt{interval} values can be written with the following syntax:

\texttt{[@] quantity unit [quantity unit...] [direction]}

Where: \texttt{quantity} is a number (possibly signed); \texttt{unit} is \texttt{microsecond}, \texttt{millisecond}, \texttt{second}, \texttt{minute}, \texttt{hour}, \texttt{day}, \texttt{week}, \texttt{month}, \texttt{year}, \texttt{decade}, \texttt{century}, \texttt{millennium}, or abbreviations or plu-
Chapter 8. Data Types

...ernals of these units; direction can be ago or empty. The at sign (@) is optional noise. The amounts of different units are implicitly added up with appropriate sign accounting.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, ‘1 12:59:10’ is read the same as ‘1 day 12 hours 59 min 10 sec’.

The optional subsecond precision $p$ should be between 0 and 6, and defaults to the precision of the input literal.

Internally interval values are stored as months, days, and seconds. This is done because the number of days in a month varies, and a day can have 23 or 25 hours if a daylight savings time adjustment is involved. Because intervals are usually created from constant strings or timestamp subtraction, this storage method works well in most cases. Functions justify_days and justify_hours are available for adjusting days and hours that overflow their normal periods.

8.5.1.5. Special Values

PostgreSQL supports several special date/time input values for convenience, as shown in Table 8-13. The values infinity and -infinity are specially represented inside the system and will be displayed the same way; but the others are simply notational shorthands that will be converted to ordinary date/time values when read. (In particular, now and related strings are converted to a specific time value as soon as they are read.) All of these values need to be written in single quotes when used as constants in SQL commands.

Table 8-13. Special Date/Time Inputs

<table>
<thead>
<tr>
<th>Input String</th>
<th>Valid Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>epoch</td>
<td>date, timestamp</td>
<td>1970-01-01 00:00:00+00 (Unix system time zero)</td>
</tr>
<tr>
<td>infinity</td>
<td>timestamp</td>
<td>later than all other time stamps</td>
</tr>
<tr>
<td>-infinity</td>
<td>timestamp</td>
<td>earlier than all other time stamps</td>
</tr>
<tr>
<td>now</td>
<td>date, time, timestamp</td>
<td>current transaction’s start time</td>
</tr>
<tr>
<td>today</td>
<td>date, timestamp</td>
<td>midnight today</td>
</tr>
<tr>
<td>tomorrow</td>
<td>date, timestamp</td>
<td>midnight tomorrow</td>
</tr>
<tr>
<td>yesterday</td>
<td>date, timestamp</td>
<td>midnight yesterday</td>
</tr>
<tr>
<td>allballs</td>
<td>time</td>
<td>00:00:00.00 UTC</td>
</tr>
</tbody>
</table>

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP. The latter four accept an optional subsecond precision specification. (See Section 9.9.4.) Note however that these are SQL functions and are not recognized as data input strings.

8.5.2. Date/Time Output

The output format of the date/time types can be set to one of the four styles ISO 8601, SQL (Ingres), traditional POSTGRES, and German, using the command SET datestyle. The default is the ISO format. (The SQL standard requires the use of the ISO 8601 format. The name of the “SQL” output format is a historical accident.) Table 8-14 shows examples of each output style. The output of the date and time types is of course only the date or time part in accordance with the given examples.
Table 8-14. Date/Time Output Styles

<table>
<thead>
<tr>
<th>Style Specification</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO</td>
<td>ISO 8601/SQL standard</td>
<td>1997-12-17 07:37:16-08</td>
</tr>
<tr>
<td>SQL</td>
<td>traditional style</td>
<td>12/17/1997 07:37:16.00 PST</td>
</tr>
<tr>
<td>POSTGRES</td>
<td>original style</td>
<td>Wed Dec 17 07:37:16 1997 PST</td>
</tr>
<tr>
<td>German</td>
<td>regional style</td>
<td>17.12.1997 07:37:16.00 PST</td>
</tr>
</tbody>
</table>

In the SQL and POSTGRES styles, day appears before month if DMY field ordering has been specified, otherwise month appears before day. (See Section 8.5.1 for how this setting also affects interpretation of input values.) Table 8-15 shows an example.

Table 8-15. Date Order Conventions

<table>
<thead>
<tr>
<th>datestyle Setting</th>
<th>Input Ordering</th>
<th>Example Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL, DMY</td>
<td>day/month/year</td>
<td>17/12/1997 15:37:16.00 CET</td>
</tr>
<tr>
<td>SQL, MDY</td>
<td>month/day/year</td>
<td>12/17/1997 07:37:16.00 PST</td>
</tr>
<tr>
<td>Postgres, DMY</td>
<td>day/month/year</td>
<td>Wed 17 Dec 07:37:16 1997 PST</td>
</tr>
</tbody>
</table>

interval output looks like the input format, except that units like century or week are converted to years and days and ago is converted to an appropriate sign. In ISO mode the output looks like:

```
[ quantity unit [ ... ] ] [ days ] [ hours:minutes:seconds ]
```

The date/time styles can be selected by the user using the SET datestyle command, the DateStyle parameter in the postgresql.conf configuration file, or the PGDATETIME environment variable on the server or client. The formatting function to_char (see Section 9.8) is also available as a more flexible way to format the date/time output.

### 8.5.3. Time Zones

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900’s, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules. PostgreSQL currently supports daylight-savings rules over the time period 1902 through 2038 (corresponding to the full range of conventional Unix system time). Times outside that range are taken to be in “standard time” for the selected time zone, no matter what part of the year they fall in.

PostgreSQL endeavors to be compatible with the SQL standard definitions for typical usage. However, the SQL standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the date type does not have an associated time zone, the time type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
- The default time zone is specified as a constant numeric offset from UTC. It is therefore not possible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.
To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We recommend not using the type time with time zone (though it is supported by PostgreSQL for legacy applications and for compliance with the SQL standard). PostgreSQL assumes your local time zone for any type containing only date or time.

All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the timezone configuration parameter before being displayed to the client.

PostgreSQL allows you to specify time zones in three different forms:

- A full time zone name, for example America/New_York. The recognized time zone names are listed in the pg_timezone_names view (see Section 44.56). PostgreSQL uses the widely-used zic time zone data for this purpose, so the same names are also recognized by much other software.

- A time zone abbreviation, for example PST. Such a specification merely defines a particular offset from UTC, in contrast to full time zone names which might imply a set of daylight savings transition-date rules as well. The recognized abbreviations are listed in the pg_timezone_abbrevs view (see Section 44.55). You cannot set the configuration parameters timezone or log_timezone using a time zone abbreviation, but you can use abbreviations in date/time input values and with the AT TIME ZONE operator.

- In addition to the timezone names and abbreviations, PostgreSQL will accept POSIX-style time zone specifications of the form STDoffset or STDoffsetDST, where STD is a zone abbreviation, offset is a numeric offset in hours west from UTC, and DST is an optional daylight-savings zone abbreviation, assumed to stand for one hour ahead of the given offset. For example, if EST5EDT were not already a recognized zone name, it would be accepted and would be functionally equivalent to USA East Coast time. When a daylight-savings zone name is present, it is assumed to be used according to the same daylight-savings transition rules used in the zic time zone database’s posixrules entry. In a standard PostgreSQL installation, posixrules is the same as US/Eastern, so that POSIX-style time zone specifications follow USA daylight-savings rules. If needed, you can adjust this behavior by replacing the posixrules file.

There is a conceptual and practical difference between the abbreviations and the full names: abbreviations always represent a fixed offset from UTC, whereas most of the full names imply a local daylight-savings time rule and so have two possible UTC offsets.

One should be wary that the POSIX-style time zone feature can lead to silently accepting bogus input, since there is no check on the reasonableness of the zone abbreviations. For example, SET TIMEZONE TO FOOBAR0 will work, leaving the system effectively using a rather peculiar abbreviation for UTC.

Another issue to keep in mind is that in POSIX time zone names, positive offsets are used for locations west of Greenwich. Everywhere else, PostgreSQL follows the ISO-8601 convention that positive timezone offsets are east of Greenwich.

In all cases, timezone names are recognized case-insensitively. (This is a change from PostgreSQL versions prior to 8.2, which were case-sensitive in some contexts and not others.)

Neither full names nor abbreviations are hard-wired into the server; they are obtained from configuration files stored under .../share/timezone/ and .../share/timezonesets/ of the installation directory (see Section B.3).

The timezone configuration parameter can be set in the file postgresql.conf, or in any of the other standard ways described in Chapter 18. There are also several special ways to set it:

- If timezone is not specified in postgresql.conf nor as a server command-line option, the server attempts to use the value of the TZ environment variable as the default time zone. If TZ is
Chapter 8. Data Types

not defined or is not any of the time zone names known to PostgreSQL, the server attempts to
determine the operating system’s default time zone by checking the behavior of the C library func-
tion \texttt{localtime()}. The default time zone is selected as the closest match among PostgreSQL’s
known time zones. (These rules are also used to choose the default value of \texttt{log\_timezone}, if it is
not specified.)

• The SQL command \texttt{SET TIME ZONE} sets the time zone for the session. This is an alternative
   spelling of \texttt{SET TIMEZONE TO} with a more SQL-spec-compatible syntax.

• The \texttt{PGTZ} environment variable, if set at the client, is used by libpq applications to send a \texttt{SET
   TIME ZONE} command to the server upon connection.

8.5.4. Internals

PostgreSQL uses Julian dates for all date/time calculations. They have the nice property of correctly
predicting/calculating any date more recent than 4713 BC to far into the future, using the assumption
that the length of the year is 365.2425 days.

Date conventions before the 19th century make for interesting reading, but are not consistent enough
to warrant coding into a date/time handler.

8.6. Boolean Type

PostgreSQL provides the standard SQL type \texttt{boolean}. \texttt{boolean} can have one of only two states:
“true” or “false”. A third state, “unknown”, is represented by the SQL null value.

Valid literal values for the “true” state are:

\begin{verbatim}
TRUE
    't'
    'true'
    'y'
    'yes'
    '1'
\end{verbatim}

For the “false” state, the following values can be used:

\begin{verbatim}
FALSE
    'f'
    'false'
    'n'
    'no'
    '0'
\end{verbatim}

Leading and trailing whitespace is ignored. Using the key words \texttt{TRUE} and \texttt{FALSE} is preferred (and
SQL-compliant).
Example 8-2. Using the boolean type

CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;

| a | b       |
|---+---------|
| t | sic est |
| f | non est |

SELECT * FROM test1 WHERE a;

| a | b       |
|---+---------|
| t | sic est |

Example 8-2 shows that boolean values are output using the letters t and f.

boolean uses 1 byte of storage.

8.7. Enumerated Types

Enumerated (enum) types are data types that are comprised of a static, predefined set of values with a specific order. They are equivalent to the enum types in a number of programming languages. An example of an enum type might be the days of the week, or a set of status values for a piece of data.

8.7.1. Declaration of Enumerated Types

Enum types are created using the CREATE TYPE command, for example:

CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

Once created, the enum type can be used in table and function definitions much like any other type:

Example 8-3. Basic Enum Usage

CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person {
  name text,
  current_mood mood
};
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';

<table>
<thead>
<tr>
<th>name</th>
<th>current_mood</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moe</td>
<td>happy</td>
</tr>
</tbody>
</table>

(1 row)
8.7.2. Ordering

The ordering of the values in an enum type is the order in which the values were listed when the type was declared. All standard comparison operators and related aggregate functions are supported for enums. For example:

Example 8-4. Enum Ordering

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
  name | current_mood
--------+------------
  Moe   | happy
  Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
  name | current_mood
--------+------------
  Curly | ok
  Moe   | happy
(2 rows)

SELECT name FROM person
  WHERE current_mood = (SELECT MIN(current_mood) FROM person);
  name
--------
  Larry
(1 row)
```

8.7.3. Type Safety

Enumerated types are completely separate data types and may not be compared with each other.

Example 8-5. Lack of Casting

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
  num_weeks int,
  happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR: invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR: operator does not exist: mood = happiness
```

If you really need to do something like that, you can either write a custom operator or add explicit casts to your query:
Chapter 8. Data Types

Example 8-6. Comparing Different Enums by Casting to Text

```sql
SELECT person.name, holidays.num_weeks FROM person, holidays
WHERE person.current_mood::text = holidays.happiness::text;
```

<table>
<thead>
<tr>
<th>name</th>
<th>num_weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moe</td>
<td>4</td>
</tr>
</tbody>
</table>

(1 row)

8.7.4. Implementation Details

An enum value occupies four bytes on disk. The length of an enum value’s textual label is limited by the NAMEDATALEN setting compiled into PostgreSQL; in standard builds this means at most 63 bytes.

Enum labels are case sensitive, so ‘happy’ is not the same as ‘HAPPY’. Spaces in the labels are significant, too.

8.8. Geometric Types

Geometric data types represent two-dimensional spatial objects. Table 8-16 shows the geometric types available in PostgreSQL. The most fundamental type, the point, forms the basis for all of the other types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>16 bytes</td>
<td>Point on the plane</td>
<td>(x,y)</td>
</tr>
<tr>
<td>line</td>
<td>32 bytes</td>
<td>Infinite line (not fully implemented)</td>
<td>((x1,y1),(x2,y2))</td>
</tr>
<tr>
<td>lseg</td>
<td>32 bytes</td>
<td>Finite line segment</td>
<td>((x1,y1),(x2,y2))</td>
</tr>
<tr>
<td>box</td>
<td>32 bytes</td>
<td>Rectangular box</td>
<td>((x1,y1),(x2,y2))</td>
</tr>
<tr>
<td>path</td>
<td>16+16n bytes</td>
<td>Closed path (similar to polygon)</td>
<td>((x1,y1),...)</td>
</tr>
<tr>
<td>path</td>
<td>16+16n bytes</td>
<td>Open path</td>
<td>[(x1,y1),...</td>
</tr>
<tr>
<td>polygon</td>
<td>40+16n bytes</td>
<td>Polygon (similar to closed path)</td>
<td>((x1,y1),...)</td>
</tr>
<tr>
<td>circle</td>
<td>24 bytes</td>
<td>Circle</td>
<td>&lt;(x,y),r&gt; (center and radius)</td>
</tr>
</tbody>
</table>

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections. They are explained in Section 9.11.

8.8.1. Points

Points are the fundamental two-dimensional building block for geometric types. Values of type point are specified using either of the following syntaxes:

( x , y )
where \( x \) and \( y \) are the respective coordinates as floating-point numbers.

Points are output using the first syntax.

### 8.8.2. Line Segments

Line segments (lseg) are represented by pairs of points. Values of type `lseg` are specified using any of the following syntaxes:

\[
\begin{align*}
&[ ( x_1, \ y_1 ), \ ( x_2, \ y_2 ) ] \\
&( ( x_1, \ y_1 ), \ ( x_2, \ y_2 ) ) \\
&( x_1, \ y_1, \ x_2, \ y_2 )
\end{align*}
\]

where \( (x_1,y_1) \) and \( (x_2,y_2) \) are the end points of the line segment.

Line segments are output using the first syntax.

### 8.8.3. Boxes

Boxes are represented by pairs of points that are opposite corners of the box. Values of type `box` are specified using any of the following syntaxes:

\[
\begin{align*}
&( ( x_1, \ y_1 ), \ ( x_2, \ y_2 ) ) \\
&( x_1, \ y_1, \ x_2, \ y_2 )
\end{align*}
\]

where \( (x_1,y_1) \) and \( (x_2,y_2) \) are any two opposite corners of the box.

Boxes are output using the second syntax.

Any two opposite corners can be supplied on input, but the values will be reordered as needed to store the upper right and lower left corners, in that order.

### 8.8.4. Paths

Paths are represented by lists of connected points. Paths can be open, where the first and last points in the list are not considered connected, or closed, where the first and last points are considered connected.

Values of type `path` are specified using any of the following syntaxes:

\[
\begin{align*}
&[ ( x_1, \ y_1 ), \ ... , \ ( x_n, \ y_n ) ] \\
&( ( x_1, \ y_1 ), \ ... , \ ( x_n, \ y_n ) ) \\
&( x_1, \ y_1, \ ... , \ x_n, \ y_n )
\end{align*}
\]

where the points are the end points of the line segments comprising the path. Square brackets ([ ]) indicate an open path, while parentheses (()) indicate a closed path. When the outermost parentheses are omitted, as in the third through fifth syntaxes, a closed path is assumed.

Paths are output using the first syntax.
8.8.5. Polygons

Polygons are represented by lists of points (the vertexes of the polygon). Polygons should probably be considered equivalent to closed paths, but are stored differently and have their own set of support routines.

Values of type `polygon` are specified using any of the following syntaxes:

\[
( (x_1, y_1), \ldots, (x_n, y_n) )
\]

where the points are the end points of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax.

8.8.6. Circles

Circles are represented by a center point and a radius. Values of type `circle` are specified using any of the following syntaxes:

\[
\langle (x, y), r \rangle
\]

where \((x, y)\) is the center and \(r\) is the radius of the circle.

Circles are output using the first syntax.

8.9. Network Address Types

PostgreSQL offers data types to store IPv4, IPv6, and MAC addresses, as shown in Table 8-17. It is preferable to use these types instead of plain text types to store network addresses, because these types offer input error checking and several specialized operators and functions (see Section 9.12).

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cidr</td>
<td>7 or 19 bytes</td>
<td>IPv4 and IPv6 networks</td>
</tr>
<tr>
<td>inet</td>
<td>7 or 19 bytes</td>
<td>IPv4 and IPv6 hosts and networks</td>
</tr>
<tr>
<td>macaddr</td>
<td>6 bytes</td>
<td>MAC addresses</td>
</tr>
</tbody>
</table>

When sorting `inet` or `cidr` data types, IPv4 addresses will always sort before IPv6 addresses, including IPv4 addresses encapsulated or mapped into IPv6 addresses, such as ::10.2.3.4 or ::ffff:10.4.3.2.

8.9.1. `inet`

The `inet` type holds an IPv4 or IPv6 host address, and optionally the identity of the subnet it is in, all
in one field. The subnet identity is represented by stating how many bits of the host address represent the network address (the “netmask”). If the netmask is 32 and the address is IPv4, then the value does not indicate a subnet, only a single host. In IPv6, the address length is 128 bits, so 128 bits specify a unique host address. Note that if you want to accept networks only, you should use the **cidr** type rather than **inet**.

The input format for this type is `address/y` where `address` is an IPv4 or IPv6 address and `y` is the number of bits in the netmask. If the `/y` part is left off, then the netmask is 32 for IPv4 and 128 for IPv6, so the value represents just a single host. On display, the `/y` portion is suppressed if the netmask specifies a single host.

### 8.9.2. cidr

The **cidr** type holds an IPv4 or IPv6 network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying networks is `address/y` where `address` is the network represented as an IPv4 or IPv6 address, and `y` is the number of bits in the netmask. If `y` is omitted, it is calculated using assumptions from the older classful network numbering system, except that it will be at least large enough to include all of the octets written in the input. It is an error to specify a network address that has bits set to the right of the specified netmask.

Table 8-18 shows some examples.

**Table 8-18. cidr Type Input Examples**

<table>
<thead>
<tr>
<th>cidr Input</th>
<th>cidr Output</th>
<th>abbrev(cidr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.100.128/25</td>
<td>192.168.100.128/25</td>
<td>192.168.100.128/25</td>
</tr>
<tr>
<td>192.168/24</td>
<td>192.168.0.0/24</td>
<td>192.168.0/24</td>
</tr>
<tr>
<td>192.168/25</td>
<td>192.168.0.0/25</td>
<td>192.168.0.0/25</td>
</tr>
<tr>
<td>192.168/1</td>
<td>192.168.1.0/24</td>
<td>192.168.1/24</td>
</tr>
<tr>
<td>192.168</td>
<td>192.168.0.0/24</td>
<td>192.168.0/24</td>
</tr>
<tr>
<td>128.1</td>
<td>128.1.0.0/16</td>
<td>128.1/16</td>
</tr>
<tr>
<td>128</td>
<td>128.0.0.0/16</td>
<td>128.0/16</td>
</tr>
<tr>
<td>128.1.2</td>
<td>128.1.2.0/24</td>
<td>128.1.2/24</td>
</tr>
<tr>
<td>10.1.2</td>
<td>10.1.2.0/24</td>
<td>10.1.2/24</td>
</tr>
<tr>
<td>10.1</td>
<td>10.1.0.0/16</td>
<td>10.1/16</td>
</tr>
<tr>
<td>10</td>
<td>10.0.0.0/8</td>
<td>10/8</td>
</tr>
<tr>
<td>10.1.2.3/32</td>
<td>10.1.2.3/32</td>
<td>10.1.2.3/32</td>
</tr>
<tr>
<td>2001:4f8:3:ba:/64</td>
<td>2001:4f8:3:ba:/64</td>
<td>2001:4f8:3:ba:/64</td>
</tr>
<tr>
<td>::ffff:1.2.3.0/120</td>
<td>::ffff:1.2.3.0/120</td>
<td>::ffff:1.2.3/120</td>
</tr>
<tr>
<td>::ffff:1.2.3.0/128</td>
<td>::ffff:1.2.3.0/128</td>
<td>::ffff:1.2.3.0/128</td>
</tr>
</tbody>
</table>

### 8.9.3. inet vs. cidr

The essential difference between **inet** and **cidr** data types is that **inet** accepts values with nonzero bits to the right of the netmask, whereas **cidr** does not.
Tip: If you do not like the output format for inet or cidr values, try the functions host, text, and abbrev.

8.9.4. macaddr

The macaddr type stores MAC addresses, i.e., Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in various customary formats, including

'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08-00-2b-01-02-03'
'08:00:2b:01:02:03'

which would all specify the same address. Upper and lower case is accepted for the digits a through f. Output is always in the last of the forms shown.

8.10. Bit String Types

Bit strings are strings of 1’s and 0’s. They can be used to store or visualize bit masks. There are two SQL bit types: bit(n) and bit varying(n), where n is a positive integer.

Bit type data must match the length n exactly; it is an error to attempt to store shorter or longer bit strings. Bit varying data is of variable length up to the maximum length n; longer strings will be rejected. Writing bit without a length is equivalent to bit(1), while bit varying without a length specification means unlimited length.

Note: If one explicitly casts a bit-string value to bit(n), it will be truncated or zero-padded on the right to be exactly n bits, without raising an error. Similarly, if one explicitly casts a bit-string value to bit varying(n), it will be truncated on the right if it is more than n bits.

Refer to Section 4.1.2.3 for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see Section 9.6.

Example 8-7. Using the bit string types

CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B’101’, B’00’);
INSERT INTO test VALUES (B’10’, B’101’);
ERROR: bit string length 2 does not match type bit(3)
INSERT INTO test VALUES (B’10’::bit(3), B’101’);
SELECT * FROM test;
a | b
-----+-----
101 | 00
100 | 101
Chapter 8. Data Types

A bit string value requires 1 byte for each group of 8 bits, plus 5 or 8 bytes overhead depending on the length of the string (but long values may be compressed or moved out-of-line, as explained in Section 8.3 for character strings).

8.11. Text Search Types

PostgreSQL provides two data types that are designed to support full text search, which is the activity of searching through a collection of natural-language documents to locate those that best match a query. The tsvector type represents a document in a form suited for text search, while the tsquery type similarly represents a query. Chapter 12 provides a detailed explanation of this facility, and Section 9.13 summarizes the related functions and operators.

8.11.1. tsvector

A tsvector value is a sorted list of distinct lexemes, which are words that have been normalized to make different variants of the same word look alike (see Chapter 12 for details). Sorting and duplicate-elimination are done automatically during input, as shown in this example:

```sql
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
```

```
tsvecto
' a' 'on' 'and' 'ate' 'cat' 'fat' 'mat' 'rat' 'sat'
```

(As the example shows, the sorting is first by length and then alphabetically, but that detail is seldom important.) To represent lexemes containing whitespace or punctuation, surround them with quotes:

```sql
SELECT $$the lexeme ' ' contains spaces$$::tsvector;
```

```
tsvecto
' the' ' ' lexeme' 'spaces' 'contains'
```

(We use dollar-quoted string literals in this example and the next one, to avoid confusing matters by having to double quote marks within the literals.) Embedded quotes and backslashes must be doubled:

```sql
SELECT $$the lexeme 'Joe"s' contains a quote$$::tsvector;
```

```
tsvecto
'a' 'the' 'Joe"s' 'quote' 'lexeme' 'contains'
```

Optionally, integer position(s) can be attached to any or all of the lexemes:

```sql
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
```

```
tsvecto
'a':1,6,10 'on':5 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

A position normally indicates the source word’s location in the document. Positional information can be used for proximity ranking. Position values can range from 1 to 16383; larger numbers are silently clamped to 16383. Duplicate positions for the same lexeme are discarded.

Lexemes that have positions can further be labeled with a weight, which can be A, B, C, or D. D is the default and hence is not shown on output:

```sql
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
```
Weights are typically used to reflect document structure, for example by marking title words differently from body words. Text search ranking functions can assign different priorities to the different weight markers.

It is important to understand that the \texttt{tsvector} type itself does not perform any normalization; it assumes that the words it is given are normalized appropriately for the application. For example,

\begin{verbatim}
SELECT 'The Fat Rats'::tsvector;
\end{verbatim}

\begin{verbatim}
\texttt{tsvector}
\end{verbatim}

\begin{verbatim}
'Fat' 'The' 'Rats'
\end{verbatim}

For most English-text-searching applications the above words would be considered non-normalized, but \texttt{tsvector} doesn't care. Raw document text should usually be passed through \texttt{to_tsvector} to normalize the words appropriately for searching:

\begin{verbatim}
SELECT to_tsvector('english', 'The Fat Rats');
\end{verbatim}

\begin{verbatim}
\texttt{to_tsvector}
\end{verbatim}

\begin{verbatim}
'fat':2 'rat':3
\end{verbatim}

Again, see Chapter 12 for more detail.

\section{tsquery}

A \texttt{tsquery} value stores lexemes that are to be searched for, and combines them using the boolean operators $\&$ (AND), $|$ (OR), and $!$ (NOT). Parentheses can be used to enforce grouping of the operators:

\begin{verbatim}
SELECT 'fat & rat'::tsquery;
\end{verbatim}

\begin{verbatim}
\texttt{tsquery}
\end{verbatim}

\begin{verbatim}
'fat' & 'rat'
\end{verbatim}

\begin{verbatim}
SELECT 'fat & (rat | cat)'::tsquery;
\end{verbatim}

\begin{verbatim}
\texttt{tsquery}
\end{verbatim}

\begin{verbatim}
'fat' & ('rat' | 'cat')
\end{verbatim}

\begin{verbatim}
SELECT 'fat & rat & ! cat'::tsquery;
\end{verbatim}

\begin{verbatim}
\texttt{tsquery}
\end{verbatim}

\begin{verbatim}
'fat' & 'rat' & !'cat'
\end{verbatim}

In the absence of parentheses, ! (NOT) binds most tightly, and $\&$ (AND) binds more tightly than $|$ (OR).

Optionally, lexemes in a \texttt{tsquery} can be labeled with one or more weight letters, which restricts them to match only \texttt{tsvector} lexemes with one of those weights:

\begin{verbatim}
SELECT 'fat:ab & cat'::tsquery;
\end{verbatim}

\begin{verbatim}
\texttt{tsquery}
\end{verbatim}
Quoting rules for lexemes are the same as described above for lexemes in tsvector; and, as with tsvector, any required normalization of words must be done before putting them into the tsquery type. The to_tsquery function is convenient for performing such normalization:

```
SELECT to_tsquery('Fat:ab & Cats');
```

Output is always in the standard form.

PostgreSQL provides storage and comparison functions for UUIDs, but the core database does not include any function for generating UUIDs, because no single algorithm is well suited for every application. The contrib module contrib/uuid-ossp provides functions that implement several standard algorithms. Alternatively, UUIDs could be generated by client applications or other libraries invoked through a server-side function.

8.13. XML Type

The data type xml can be used to store XML data. Its advantage over storing XML data in a text field is that it checks the input values for well-formedness, and there are support functions to perform
type-safe operations on it; see Section 9.14. Use of this data type requires the installation to have been built with configure --with-libxml.

The xml type can store well-formed “documents”, as defined by the XML standard, as well as “content” fragments, which are defined by the production XMLDecl? content in the XML standard. Roughly, this means that content fragments can have more than one top-level element or character node. The expression xmlvalue IS DOCUMENT can be used to evaluate whether a particular xml value is a full document or only a content fragment.

### 8.13.1. Creating XML Values

To produce a value of type xml from character data, use the function xmlparse:

\[
\text{XMLPARSE ( \{ DOCUMENT | CONTENT \} value)}
\]

Examples:

XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')

XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')</n

While this is the only way to convert character strings into XML values according to the SQL standard, the PostgreSQL-specific syntaxes:

\[
\text{xml '<foo>bar</foo>''<foo>bar</foo>':'::xml}
\]

can also be used.

The xml type does not validate its input values against a possibly included document type declaration (DTD).

The inverse operation, producing character string type values from xml, uses the function xmlserialize:

\[
\text{XMLSERIALIZE ( \{ DOCUMENT | CONTENT \} value AS type )}
\]

type can be one of character, character varying, or text (or an alias name for those). Again, according to the SQL standard, this is the only way to convert between type xml and character types, but PostgreSQL also allows you to simply cast the value.

When character string values are cast to or from type xml without going through XMLPARSE or XMLSERIALIZE, respectively, the choice of DOCUMENT versus CONTENT is determined by the “XML option” session configuration parameter, which can be set using the standard command

\[
\text{SET XML OPTION \{ DOCUMENT | CONTENT \};}
\]

or the more PostgreSQL-like syntax

\[
\text{SET xmloption TO \{ DOCUMENT | CONTENT \};}
\]

The default is CONTENT, so all forms of XML data are allowed.

### 8.13.2. Encoding Handling

Care must be taken when dealing with multiple character encodings on the client, server, and in the XML data passed through them. When using the text mode to pass queries to the server and query
results to the client (which is the normal mode), PostgreSQL converts all character data passed between the client and the server and vice versa to the character encoding of the respective end; see Section 22.2. This includes string representations of XML values, such as in the above examples. This would ordinarily mean that encoding declarations contained in XML data might become invalid as the character data is converted to other encodings while travelling between client and server, while the embedded encoding declaration is not changed. To cope with this behavior, an encoding declaration contained in a character string presented for input to the xml type is ignored, and the content is always assumed to be in the current server encoding. Consequently, for correct processing, such character strings of XML data must be sent off from the client in the current client encoding. It is the responsibility of the client to either convert the document to the current client encoding before sending it off to the server or to adjust the client encoding appropriately. On output, values of type xml will not have an encoding declaration, and clients must assume that the data is in the current client encoding.

When using the binary mode to pass query parameters to the server and query results back to the client, no character set conversion is performed, so the situation is different. In this case, an encoding declaration in the XML data will be observed, and if it is absent, the data will be assumed to be in UTF-8 (as required by the XML standard; note that PostgreSQL does not support UTF-16 at all). On output, data will have an encoding declaration specifying the client encoding, unless the client encoding is UTF-8, in which case it will be omitted.

Needless to say, processing XML data with PostgreSQL will be less error-prone and more efficient if data encoding, client encoding, and server encoding are the same. Since XML data is internally processed in UTF-8, computations will be most efficient if the server encoding is also UTF-8.

### 8.13.3. Accessing XML Values

The xml data type is unusual in that it does not provide any comparison operators. This is because there is no well-defined and universally useful comparison algorithm for XML data. One consequence of this is that you cannot retrieve rows by comparing an xml column against a search value. XML values should therefore typically be accompanied by a separate key field such as an ID. An alternative solution for comparing XML values is to convert them to character strings first, but note that character string comparison has little to do with a useful XML comparison method.

Since there are no comparison operators for the xml data type, it is not possible to create an index directly on a column of this type. If speedy searches in XML data are desired, possible workarounds would be casting the expression to a character string type and indexing that, or indexing an XPath expression. The actual query would of course have to be adjusted to search by the indexed expression.

The text-search functionality in PostgreSQL could also be used to speed up full-document searches in XML data. The necessary preprocessing support is, however, not available in the PostgreSQL distribution in this release.

### 8.14. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in or user-defined base type, enum type, or composite type can be created. Arrays of domains are not yet supported.
8.14.1. Declaration of Array Types

To illustrate the use of array types, we create this table:

```sql
CREATE TABLE sal_emp (  
    name text,  
    pay_by_quarter integer[],  
    schedule text[][]
);
```

As shown, an array data type is named by appending square brackets ([ ]) to the data type name of the array elements. The above command will create a table named `sal_emp` with a column of type `text` (name), a one-dimensional array of type `integer` (pay_by_quarter), which represents the employee’s salary by quarter, and a two-dimensional array of `text` (schedule), which represents the employee’s weekly schedule.

The syntax for `CREATE TABLE` allows the exact size of arrays to be specified, for example:

```sql
CREATE TABLE tictactoe (  
    squares integer[3][3]
);
```

However, the current implementation does not enforce the array size limits — the behavior is the same as for arrays of unspecified length.

Actually, the current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring number of dimensions or sizes in `CREATE TABLE` is simply documentation, it does not affect run-time behavior.

An alternative syntax, which conforms to the SQL standard, can be used for one-dimensional arrays. `pay_by_quarter` could have been defined as:

```
    pay_by_quarter integer ARRAY[4],
```

This syntax requires an integer constant to denote the array size. As before, however, PostgreSQL does not enforce the size restriction.

8.14.2. Array Value Input

To write an array value as a literal constant, enclose the element values within curly braces and separate them by commas. (If you know C, this is not unlike the C syntax for initializing structures.) You can put double quotes around any element value, and must do so if it contains commas or curly braces. (More details appear below.) Thus, the general format of an array constant is the following:

```
'{ val1 delim val2 delim ... }'
```

where `delim` is the delimiter character for the type, as recorded in its `pg_type` entry. Among the standard data types provided in the PostgreSQL distribution, type `box` uses a semicolon (`;`) but all the others use comma (`,`). Each `val` is either a constant of the array element type, or a subarray. An example of an array constant is:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

This constant is a two-dimensional, 3-by-3 array consisting of three subarrays of integers.
To set an element of an array constant to NULL, write NULL for the element value. (Any upper- or lower-case variant of NULL will do.) If you want an actual string value “NULL”, you must put double quotes around it.

(These kinds of array constants are actually only a special case of the generic type constants discussed in Section 4.1.2.5. The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

Now we can show some INSERT statements:

```sql
INSERT INTO sal_emp
VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

The result of the previous two inserts looks like this:

```
SELECT * FROM sal_emp;
name | pay_by_quarter | schedule
-------+---------------------------+-------------------------------------------
Bill | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

The ARRAY constructor syntax can also be used:

```sql
INSERT INTO sal_emp
VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[{'meeting','lunch'}, {'training','presentation'}]);

INSERT INTO sal_emp
VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[{'breakfast','consulting'}, {'meeting','lunch'}]);
```

Notice that the array elements are ordinary SQL constants or expressions; for instance, string literals are single quoted, instead of double quoted as they would be in an array literal. The ARRAY constructor syntax is discussed in more detail in Section 4.2.10.

Multidimensional arrays must have matching extents for each dimension. A mismatch causes an error report, for example:

```sql
INSERT INTO sal_emp
VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting","lunch"}, {{"meeting"}}'});
ERROR:  multidimensional arrays must have array expressions with matching dimensions
```
8.14.3. Accessing Arrays

Now, we can run some queries on the table. First, we show how to access a single element of an array at a time. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];
```

```
name
-----
Carol
(1 row)
```

The array subscript numbers are written within square brackets. By default PostgreSQL uses the one-based numbering convention for arrays, that is, an array of $n$ elements starts with $array[1]$ and ends with $array[n]$.

This query retrieves the third quarter pay of all employees:

```
SELECT pay_by_quarter[3] FROM sal_emp;
```

```
pay_by_quarter
----------------
 10000
 25000
(2 rows)
```

We can also access arbitrary rectangular slices of an array, or subarrays. An array slice is denoted by writing $lower-bound:upper-bound$ for one or more array dimensions. For example, this query retrieves the first item on Bill’s schedule for the first two days of the week:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';
```

```
schedule
----------
{{meeting},{training}}
(1 row)
```

If any dimension is written as a slice, i.e. contains a colon, then all dimensions are treated as slices. Any dimension that has only a single number (no colon) is treated as being from 1 to the number specified. For example, $[2]$ is treated as $[1:2]$, as in this example:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';
```

```
schedule
------------------------
{{meeting,lunch},{training,presentation}}
(1 row)
```

An array subscript expression will return null if either the array itself or any of the subscript expressions are null. Also, null is returned if a subscript is outside the array bounds (this case does not raise an error). For example, if $schedule$ currently has the dimensions $[1:3][1:2]$ then referencing $schedule[3][3]$ yields NULL. Similarly, an array reference with the wrong number of subscripts yields a null rather than an error.
An array slice expression likewise yields null if the array itself or any of the subscript expressions are null. However, in other corner cases such as selecting an array slice that is completely outside the current array bounds, a slice expression yields an empty (zero-dimensional) array instead of null. If the requested slice partially overlaps the array bounds, then it is silently reduced to just the overlapping region.

The current dimensions of any array value can be retrieved with the `array_dims` function:

```sql
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';
```

```
array_dims
------------
[1:2][1:2]
(1 row)
```

`array_dims` produces a text result, which is convenient for people to read but perhaps not so convenient for programs. Dimensions can also be retrieved with `array_upper` and `array_lower`, which return the upper and lower bound of a specified array dimension, respectively:

```sql
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';
```

```
array_upper
------------
2
(1 row)
```

### 8.14.4. Modifying Arrays

An array value can be replaced completely:

```sql
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
WHERE name = 'Carol';
```

or using the `ARRAY` expression syntax:

```sql
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
WHERE name = 'Carol';
```

An array can also be updated at a single element:

```sql
WHERE name = 'Bill';
```

or updated in a slice:

```sql
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
WHERE name = 'Carol';
```

A stored array value can be enlarged by assigning to element(s) not already present. Any positions between those previously present and the newly assigned element(s) will be filled with nulls. For example, if array `myarray` currently has 4 elements, it will have six elements after an update that assigns to `myarray[6]`, and `myarray[5]` will contain a null. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.
Subscripted assignment allows creation of arrays that do not use one-based subscripts. For example one might assign to \texttt{myarray[-2:7]} to create an array with subscript values running from -2 to 7.

New array values can also be constructed by using the concatenation operator, \texttt{||}:

```sql
SELECT ARRAY[1,2] || ARRAY[3,4];
?column?
-----------
(1,2,3,4)
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
?column?
---------------------
{{5,6},{1,2},{3,4}}
(1 row)
```

The concatenation operator allows a single element to be pushed on to the beginning or end of a one-dimensional array. It also accepts two \(N\)-dimensional arrays, or an \(N\)-dimensional and an \(N+1\)-dimensional array.

When a single element is pushed on to either the beginning or end of a one-dimensional array, the result is an array with the same lower bound subscript as the array operand. For example:

```sql
SELECT array_dims(1 || '1:0={2,3}'::int[]);
array_dims
------------
[0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
array_dims
------------
[1:3]
(1 row)
```

When two arrays with an equal number of dimensions are concatenated, the result retains the lower bound subscript of the left-hand operand’s outer dimension. The result is an array comprising every element of the left-hand operand followed by every element of the right-hand operand. For example:

```sql
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
array_dims
------------
[1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
array_dims
------------
[1:5][1:2]
(1 row)
```
When an $N$-dimensional array is pushed on to the beginning or end of an $N+1$-dimensional array, the result is analogous to the element-array case above. Each $n$-dimensional sub-array is essentially an element of the $N+1$-dimensional array’s outer dimension. For example:

```sql
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
array_dims
------------
[1:3][1:2]
(1 row)
```

An array can also be constructed by using the functions `array_prepend`, `array_append`, or `array_cat`. The first two only support one-dimensional arrays, but `array_cat` supports multidimensional arrays. Note that the concatenation operator discussed above is preferred over direct use of these functions. In fact, the functions exist primarily for use in implementing the concatenation operator. However, they might be directly useful in the creation of user-defined aggregates. Some examples:

```sql
SELECT array_prepend(1, ARRAY[2,3]);
array_prepend
---------------
{1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
array_append
---------------
{1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
array_cat
---------
{1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
array_cat
---------
{{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
array_cat
---------
{{5,6},{1,2},{3,4}}
```

### 8.14.5. Searching in Arrays

To search for a value in an array, you must check each value of the array. This can be done by hand, if you know the size of the array. For example:

```sql
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
```

---

Page 122
However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is uncertain. An alternative method is described in Section 9.20. The above query could be replaced by:

```sql
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

In addition, you could find rows where the array had all values equal to 10000 with:

```sql
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

**Tip:** Arrays are not sets; searching for specific array elements can be a sign of database misdesign. Consider using a separate table with a row for each item that would be an array element. This will be easier to search, and is likely to scale up better to large numbers of elements.

### 8.14.6. Array Input and Output Syntax

The external text representation of an array value consists of items that are interpreted according to the I/O conversion rules for the array’s element type, plus decoration that indicates the array structure. The decoration consists of curly braces (\{ and \}) around the array value plus delimiter characters between adjacent items. The delimiter character is usually a comma (, ) but can be something else: it is determined by the `typdelim` setting for the array’s element type. (Among the standard data types provided in the PostgreSQL distribution, type `box` uses a semicolon (;) but all the others use comma.) In a multidimensional array, each dimension (row, plane, cube, etc.) gets its own level of curly braces, and delimiters must be written between adjacent curly-braced entities of the same level.

The array output routine will put double quotes around element values if they are empty strings, contain curly braces, delimiter characters, double quotes, backslashes, or white space, or match the word `NULL`. Double quotes and backslashes embedded in element values will be backslash-escaped. For numeric data types it is safe to assume that double quotes will never appear, but for textual data types one should be prepared to cope with either presence or absence of quotes.

By default, the lower bound index value of an array’s dimensions is set to one. To represent arrays with other lower bounds, the array subscript ranges can be specified explicitly before writing the array contents. This decoration consists of square brackets ([ ]) around each array dimension’s lower and upper bounds, with a colon (:) delimiter character in between. The array dimension decoration is followed by an equal sign (=). For example:

```sql
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
FROM (SELECT '[[1:1][-2:-1][3:5]=({{1,2,3},{4,5,6}})']::int[] AS f1) AS ss;
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>
```

The array output routine will include explicit dimensions in its result only when there are one or more lower bounds different from one.
If the value written for an element is `NULL` (in any case variant), the element is taken to be `NULL`. The presence of any quotes or backslashes disables this and allows the literal string value “NULL” to be entered. Also, for backwards compatibility with pre-8.2 versions of PostgreSQL, the `array_nulls` configuration parameter might be turned off to suppress recognition of `NULL` as a `NULL`.

As shown previously, when writing an array value you can write double quotes around any individual array element. You must do so if the element value would otherwise confuse the array-value parser. For example, elements containing curly braces, commas (or whatever the delimiter character is), double quotes, backslashes, or leading or trailing whitespace must be double-quoted. Empty strings and strings matching the word `NULL` must be quoted, too. To put a double quote or backslash in a quoted array element value, use escape string syntax and precede it with a backslash. Alternatively, you can use backslash-escaping to protect all data characters that would otherwise be taken as array syntax.

You can write whitespace before a left brace or after a right brace. You can also write whitespace before or after any individual item string. In all of these cases the whitespace will be ignored. However, whitespace within double-quoted elements, or surrounded on both sides by non-whitespace characters of an element, is not ignored.

**Note:** Remember that what you write in an SQL command will first be interpreted as a string literal, and then as an array. This doubles the number of backslashes you need. For example, to insert a `text` array value containing a backslash and a double quote, you'd need to write:

```sql
INSERT ... VALUES (E'"\","\"');
```

The escape string processor removes one level of backslashes, so that what arrives at the array-value parser looks like `{\\","\"}`. In turn, the strings fed to the `text` data type's input routine become \ and " respectively. (If we were working with a data type whose input routine also treated backslashes specially, `bytea`, for example, we might need as many as eight backslashes in the command to get one backslash into the stored array element.) Dollar quoting (see Section 4.1.2.2) can be used to avoid the need to double backslashes.

**Tip:** The `ARRAY` constructor syntax (see Section 4.2.10) is often easier to work with than the array-literal syntax when writing array values in SQL commands. In `ARRAY`, individual element values are written the same way they would be written when not members of an array.

### 8.15. Composite Types

A composite type describes the structure of a row or record; it is in essence just a list of field names and their data types. PostgreSQL allows values of composite types to be used in many of the same ways that simple types can be used. For example, a column of a table can be declared to be of a composite type.

#### 8.15.1. Declaration of Composite Types

Here are two simple examples of defining composite types:

```sql
CREATE TYPE complex AS (  
  r double precision,  
  i double precision
```
CREATE TYPE inventory_item AS (  name text,  supplier_id integer,  price numeric );

The syntax is comparable to CREATE TABLE, except that only field names and types can be specified; no constraints (such as NOT NULL) can presently be included. Note that the AS keyword is essential; without it, the system will think a quite different kind of CREATE TYPE command is meant, and you’ll get odd syntax errors.

Having defined the types, we can use them to create tables:

CREATE TABLE on_hand (  item inventory_item,  count integer );

INSERT INTO on_hand VALUES (ROW(’fuzzy dice’, 42, 1.99), 1000);

or functions:

CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric AS ’SELECT $1.price * $2’ LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;

Whenever you create a table, a composite type is also automatically created, with the same name as the table, to represent the table’s row type. For example, had we said:

CREATE TABLE inventory_item (  name text,  supplier_id integer REFERENCES suppliers,  price numeric CHECK (price > 0) );

then the same inventory_item composite type shown above would come into being as a byproduct, and could be used just as above. Note however an important restriction of the current implementation: since no constraints are associated with a composite type, the constraints shown in the table definition do not apply to values of the composite type outside the table. (A partial workaround is to use domain types as members of composite types.)

8.15.2. Composite Value Input

To write a composite value as a literal constant, enclose the field values within parentheses and separate them by commas. You can put double quotes around any field value, and must do so if it contains commas or parentheses. (More details appear below.) Thus, the general format of a composite constant is the following:

’( val1 , val2 , ... )’

An example is:
Chapter 8. Data Types

\[ \text{"fuzzy dice", 42, 1.99}\]

which would be a valid value of the `inventory_item` type defined above. To make a field be NULL,
write no characters at all in its position in the list. For example, this constant specifies a NULL third
field:

\[ \text{"fuzzy dice", 42,}\]

If you want an empty string rather than NULL, write double quotes:

\[ ", 42,\]

Here the first field is a non-NULL empty string, the third is NULL.

(These constants are actually only a special case of the generic type constants discussed in Section
4.1.2.5. The constant is initially treated as a string and passed to the composite-type input conversion
routine. An explicit type specification might be necessary.)

The `ROW` expression syntax can also be used to construct composite values. In most cases this is
considerably simpler to use than the string-literal syntax, since you don’t have to worry about multiple
layers of quoting. We already used this method above:

\begin{verbatim}
ROW\('fuzzy dice', 42, 1.99)  
ROW\('', 42, NULL)
\end{verbatim}

The `ROW` keyword is actually optional as long as you have more than one field in the expression, so
these can simplify to:

\begin{verbatim}
('fuzzy dice', 42, 1.99) 
('', 42, NULL)
\end{verbatim}

The `ROW` expression syntax is discussed in more detail in Section 4.2.11.

8.15.3. Accessing Composite Types

To access a field of a composite column, one writes a dot and the field name, much like selecting a
field from a table name. In fact, it’s so much like selecting from a table name that you often have to use
parentheses to keep from confusing the parser. For example, you might try to select some subfields
from our `on_hand` example table with something like:

\begin{verbatim}
SELECT item.name FROM on_hand WHERE item.price > 9.99;
\end{verbatim}

This will not work since the name `item` is taken to be a table name, not a field name, per SQL syntax
rules. You must write it like this:

\begin{verbatim}
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
\end{verbatim}

or if you need to use the table name as well (for instance in a multitable query), like this:

\begin{verbatim}
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
\end{verbatim}

Now the parenthesized object is correctly interpreted as a reference to the `item` column, and then the
subfield can be selected from it.

Similar syntactic issues apply whenever you select a field from a composite value. For instance, to
select just one field from the result of a function that returns a composite value, you’d need to write
something like:
SELECT (my_func(...)).field FROM ...

Without the extra parentheses, this will provoke a syntax error.

### 8.15.4. Modifying Composite Types

Here are some examples of the proper syntax for inserting and updating composite columns. First, inserting or updating a whole column:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));
```

```
UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

The first example omits ROW, the second uses it; we could have done it either way.

We can update an individual subfield of a composite column:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

Notice here that we don’t need to (and indeed cannot) put parentheses around the column name appearing just after SET, but we do need parentheses when referencing the same column in the expression to the right of the equal sign.

And we can specify subfields as targets for INSERT, too:

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

Had we not supplied values for all the subfields of the column, the remaining subfields would have been filled with null values.

### 8.15.5. Composite Type Input and Output Syntax

The external text representation of a composite value consists of items that are interpreted according to the I/O conversion rules for the individual field types, plus decoration that indicates the composite structure. The decoration consists of parentheses ( ( and ) ) around the whole value, plus commas (,) between adjacent items. Whitespace outside the parentheses is ignored, but within the parentheses it is considered part of the field value, and might or might not be significant depending on the input conversion rules for the field data type. For example, in:

```
'( 42)'
```

the whitespace will be ignored if the field type is integer, but not if it is text.

As shown previously, when writing a composite value you can write double quotes around any individual field value. You must do so if the field value would otherwise confuse the composite-value parser. In particular, fields containing parentheses, commas, double quotes, or backslashes must be double-quoted. To put a double quote or backslash in a quoted composite field value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted field value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can use backslash-escaping to protect all data characters that would otherwise be taken as composite syntax.

A completely empty field value (no characters at all between the commas or parentheses) represents a NULL. To write a value that is an empty string rather than NULL, write "".
Chapter 8. Data Types

The composite output routine will put double quotes around field values if they are empty strings or contain parentheses, commas, double quotes, backslashes, or white space. (Doing so for white space is not essential, but aids legibility.) Double quotes and backslashes embedded in field values will be doubled.

Note: Remember that what you write in an SQL command will first be interpreted as a string literal, and then as a composite. This doubles the number of backslashes you need (assuming escape string syntax is used). For example, to insert a text field containing a double quote and a backslash in a composite value, you’d need to write:

```
INSERT ... VALUES (E'("\\")');
```

The string-literal processor removes one level of backslashes, so that what arrives at the composite-value parser looks like "\". (If we were working with a data type whose input routine also treated backslashes specially, for example, we might need as many as eight backslashes in the command to get one backslash into the stored composite field.) Dollar quoting (see Section 4.1.2.2) can be used to avoid the need to double backslashes.

Tip: The ROW constructor syntax is usually easier to work with than the composite-literal syntax when writing composite values in SQL commands. In ROW, individual field values are written the same way they would be written when not members of a composite.

8.16. Object Identifier Types

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. OIDs are not added to user-created tables, unless WITH OIDS is specified when the table is created, or the default_with_oids configuration variable is enabled. Type oid represents an object identifier. There are also several alias types for oid: regproc, regprocedure, regoper, regoperator, regclass, regtype, regconfig, and regdictionary. Table 8-19 shows an overview.

The oid type is currently implemented as an unsigned four-byte integer. Therefore, it is not large enough to provide database-wide uniqueness in large databases, or even in large individual tables. So, using a user-created table’s OID column as a primary key is discouraged. OIDs are best used only for references to system tables.

The oid type itself has few operations beyond comparison. It can be cast to integer, however, and then manipulated using the standard integer operators. (Beware of possible signed-versus-unsigned confusion if you do this.)

The OID alias types have no operations of their own except for specialized input and output routines. These routines are able to accept and display symbolic names for system objects, rather than the raw numeric value that type oid would use. The alias types allow simplified lookup of OID values for objects. For example, to examine the pg_attribute rows related to a table mytable, one could write:

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable':regclass;
```

rather than:

128
Chapter 8. Data Types

SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');

While that doesn’t look all that bad by itself, it’s still oversimplified. A far more complicated sub-
select would be needed to select the right OID if there are multiple tables named mytable in differ-
ent schemas. The regclass input converter handles the table lookup according to the schema path
setting, and so it does the “right thing” automatically. Similarly, casting a table’s OID to regclass
is handy for symbolic display of a numeric OID.

Table 8-19. Object Identifier Types

<table>
<thead>
<tr>
<th>Name</th>
<th>References</th>
<th>Description</th>
<th>Value Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>any</td>
<td>numeric object identifier</td>
<td>564182</td>
</tr>
<tr>
<td>regproc</td>
<td>pg_proc</td>
<td>function name</td>
<td>sum</td>
</tr>
<tr>
<td>regprocedure</td>
<td>pg_proc</td>
<td>function with argument types</td>
<td>sum(int4)</td>
</tr>
<tr>
<td>regoper</td>
<td>pg_operator</td>
<td>operator name</td>
<td>+</td>
</tr>
<tr>
<td>regoperator</td>
<td>pg_operator</td>
<td>operator with argument types</td>
<td>*(integer,integer) or -(NONE,integer)</td>
</tr>
<tr>
<td>regclass</td>
<td>pg_class</td>
<td>relation name</td>
<td>pg_type</td>
</tr>
<tr>
<td>regtype</td>
<td>pg_type</td>
<td>data type name</td>
<td>integer</td>
</tr>
<tr>
<td>regconfig</td>
<td>pg_ts_config</td>
<td>text search configuration</td>
<td>english</td>
</tr>
<tr>
<td>regdictionary</td>
<td>pg_ts_dict</td>
<td>text search dictionary</td>
<td>simple</td>
</tr>
</tbody>
</table>

All of the OID alias types accept schema-qualified names, and will display schema-qualified names
on output if the object would not be found in the current search path without being qualified. The regproc
and regoper alias types will only accept input names that are unique (not overloaded),
so they are of limited use; for most uses regprocedure or regoperator is more appropriate. For regoperator, unary operators are identified by writing NONE for the unused operand.

An additional property of the OID alias types is that if a constant of one of these types appears in a
stored expression (such as a column default expression or view), it creates a dependency on the refer-
enced object. For example, if a column has a default expression nextval('my_seq'::regclass),
PostgreSQL understands that the default expression depends on the sequence my_seq; the system
will not let the sequence be dropped without first removing the default expression.

Another identifier type used by the system is xid, or transaction (abbreviated xact) identifier. This is
the data type of the system columns xmin and xmax. Transaction identifiers are 32-bit quantities.

A third identifier type used by the system is cid, or command identifier. This is the data type of the
system columns cmin and cmax. Command identifiers are also 32-bit quantities.

A final identifier type used by the system is tid, or tuple identifier (row identifier). This is the data
type of the system column ctid. A tuple ID is a pair (block number, tuple index within block) that
identifies the physical location of the row within its table.

(The system columns are further explained in Section 5.4.)
8.17. Pseudo-Types

The PostgreSQL type system contains a number of special-purpose entries that are collectively called pseudo-types. A pseudo-type cannot be used as a column data type, but it can be used to declare a function’s argument or result type. Each of the available pseudo-types is useful in situations where a function’s behavior does not correspond to simply taking or returning a value of a specific SQL data type. Table 8-20 lists the existing pseudo-types.

Table 8-20. Pseudo-Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>Indicates that a function accepts any input data type whatever.</td>
</tr>
<tr>
<td>anyarray</td>
<td>Indicates that a function accepts any array data type (see Section 34.2.5).</td>
</tr>
<tr>
<td>anyelement</td>
<td>Indicates that a function accepts any data type (see Section 34.2.5).</td>
</tr>
<tr>
<td>anyenum</td>
<td>Indicates that a function accepts any enum data type (see Section 34.2.5 and Section 8.7).</td>
</tr>
<tr>
<td>anynonarray</td>
<td>Indicates that a function accepts any non-array data type (see Section 34.2.5).</td>
</tr>
<tr>
<td>cstring</td>
<td>Indicates that a function accepts or returns a null-terminated C string.</td>
</tr>
<tr>
<td>internal</td>
<td>Indicates that a function accepts or returns a server-internal data type.</td>
</tr>
<tr>
<td>language_handler</td>
<td>A procedural language call handler is declared to return language_handler.</td>
</tr>
<tr>
<td>record</td>
<td>Identifies a function returning an unspecified row type.</td>
</tr>
<tr>
<td>trigger</td>
<td>A trigger function is declared to return trigger.</td>
</tr>
<tr>
<td>void</td>
<td>Indicates that a function returns no value.</td>
</tr>
<tr>
<td>opaque</td>
<td>An obsolete type name that formerly served all the above purposes.</td>
</tr>
</tbody>
</table>

Functions coded in C (whether built-in or dynamically loaded) can be declared to accept or return any of these pseudo data types. It is up to the function author to ensure that the function will behave safely when a pseudo-type is used as an argument type.

Functions coded in procedural languages can use pseudo-types only as allowed by their implementation languages. At present the procedural languages all forbid use of a pseudo-type as argument type, and allow only void and record as a result type (plus trigger when the function is used as a trigger). Some also support polymorphic functions using the types anyarray, anyelement, anyenum, and anynonarray.

The internal pseudo-type is used to declare functions that are meant only to be called internally by the database system, and not by direct invocation in a SQL query. If a function has at least one internal-type argument then it cannot be called from SQL. To preserve the type safety of this restriction it is important to follow this coding rule: do not create any function that is declared to return internal unless it has at least one internal argument.
Chapter 9. Functions and Operators

PostgreSQL provides a large number of functions and operators for the built-in data types. Users can also define their own functions and operators, as described in Part V. The psql commands \df and \do can be used to show the list of all actually available functions and operators, respectively.

If you are concerned about portability then take note that most of the functions and operators described in this chapter, with the exception of the most trivial arithmetic and comparison operators and some explicitly marked functions, are not specified by the SQL standard. Some of the extended functionality is present in other SQL database management systems, and in many cases this functionality is compatible and consistent between the various implementations. This chapter is also not exhaustive; additional functions appear in relevant sections of the manual.

9.1. Logical Operators

The usual logical operators are available:

```
AND
OR
NOT
```

SQL uses a three-valued Boolean logic where the null value represents “unknown”. Observe the following truth tables:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a AND b</th>
<th>a OR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>NULL</td>
<td>NULL</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>NULL</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result. But see Section 4.2.12 for more information about the order of evaluation of subexpressions.

9.2. Comparison Operators

The usual comparison operators are available, shown in Table 9-1.
Table 9-1. Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>&lt;&gt; or !=</td>
<td>not equal</td>
</tr>
</tbody>
</table>

Note: The != operator is converted to <> in the parser stage. It is not possible to implement != and <> operators that do different things.

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type boolean: expressions like \(1 < 2 < 3\) are not valid (because there is no < operator to compare a Boolean value with 3).

In addition to the comparison operators, the special BETWEEN construct is available.

\[ a \text{ BETWEEN } x \text{ AND } y \]

is equivalent to

\[ a \geq x \text{ AND } a \leq y \]

Similarly,

\[ a \text{ NOT BETWEEN } x \text{ AND } y \]

is equivalent to

\[ a < x \text{ OR } a > y \]

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally. BETWEEN SYMMETRIC is the same as BETWEEN except there is no requirement that the argument to the left of AND be less than or equal to the argument on the right; the proper range is automatically determined.

To check whether a value is or is not null, use the constructs

\[ \text{expression IS NULL} \]
\[ \text{expression IS NOT NULL} \]

or the equivalent, but nonstandard, constructs

\[ \text{expression ISNULL} \]
\[ \text{expression NOTNULL} \]

Do not write \[ \text{expression = NULL} \] because NULL is not “equal to” NULL. (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.
Tip: Some applications might expect that \texttt{expression} = \texttt{NULL} returns true if \texttt{expression} evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard. However, if that cannot be done the transform\_null\_equals configuration variable is available. If it is enabled, PostgreSQL will convert \texttt{x = NULL} clauses to \texttt{x IS NULL}. This was the default behavior in PostgreSQL releases 6.5 through 7.1.

Note: If the \textit{expression} is row-valued, then \texttt{IS NULL} is true when the row expression itself is null or when all the row's fields are null, while \texttt{IS NOT NULL} is true when the row expression itself is non-null and all the row's fields are non-null. This definition conforms to the SQL standard, and is a change from the inconsistent behavior exhibited by PostgreSQL versions prior to 8.2.

The ordinary comparison operators yield null (signifying “unknown”) when either input is null. Another way to do comparisons is with the \texttt{IS \ [ NOT \ ] DISTINCT FROM} construct:

\begin{align*}
\texttt{expression IS DISTINCT FROM expression} \\
\texttt{expression IS NOT DISTINCT FROM expression}
\end{align*}

For non-null inputs, \texttt{IS DISTINCT FROM} is the same as the \texttt{<>} operator. However, when both inputs are null it will return false, and when just one input is null it will return true. Similarly, \texttt{IS NOT DISTINCT FROM} is identical to \texttt{=} for non-null inputs, but it returns true when both inputs are null, and false when only one input is null. Thus, these constructs effectively act as though null were a normal data value, rather than “unknown”.

Boolean values can also be tested using the constructs

\begin{align*}
\texttt{expression IS TRUE} \\
\texttt{expression IS NOT TRUE} \\
\texttt{expression IS FALSE} \\
\texttt{expression IS NOT FALSE} \\
\texttt{expression IS UNKNOWN} \\
\texttt{expression IS NOT UNKNOWN}
\end{align*}

These will always return true or false, never a null value, even when the operand is null. A null input is treated as the logical value “unknown”. Notice that \texttt{IS UNKNOWN} and \texttt{IS NOT UNKNOWN} are effectively the same as \texttt{IS NULL} and \texttt{IS NOT NULL}, respectively, except that the input expression must be of Boolean type.

### 9.3. Mathematical Functions and Operators

Mathematical operators are provided for many PostgreSQL types. For types without common mathematical conventions for all possible permutations (e.g., date/time types) we describe the actual behavior in subsequent sections.

Table 9-2 shows the available mathematical operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>2 + 3</td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>2 - 3</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiplication</td>
<td>$2 * 3$</td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>division (integer division truncates results)</td>
<td>$4 / 2$</td>
<td>2</td>
</tr>
<tr>
<td>%</td>
<td>modulo (remainder)</td>
<td>$5 % 4$</td>
<td>1</td>
</tr>
<tr>
<td>^</td>
<td>exponentiation</td>
<td>$2.0 ^ 3.0$</td>
<td>8</td>
</tr>
<tr>
<td>//</td>
<td>square root</td>
<td>$\sqrt{25.0}$</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cube root</td>
</tr>
<tr>
<td>!</td>
<td>factorial</td>
<td>$5 !$</td>
<td>120</td>
</tr>
<tr>
<td>!!</td>
<td>factorial (prefix operator)</td>
<td>$!! 5$</td>
<td>120</td>
</tr>
<tr>
<td>@</td>
<td>absolute value</td>
<td>$@ -5.0$</td>
<td>5</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>$91 &amp; 15$</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td>$32</td>
</tr>
<tr>
<td>#</td>
<td>bitwise XOR</td>
<td>$17 # 5$</td>
<td>20</td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>$\neg 1$</td>
<td>-2</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bitwise shift left</td>
<td>$1 &lt;&lt; 4$</td>
<td>16</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bitwise shift right</td>
<td>$8 &gt;&gt; 2$</td>
<td>2</td>
</tr>
</tbody>
</table>

The bitwise operators work only on integral data types, whereas the others are available for all numeric data types. The bitwise operators are also available for the bit string types `bit` and `bit varying`, as shown in Table 9-10.

Table 9-3 shows the available mathematical functions. In the table, `dp` indicates `double precision`. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with `double precision` data are mostly implemented on top of the host system’s `C` library; accuracy and behavior in boundary cases can therefore vary depending on the host system.

### Table 9-3. Mathematical Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(x)</code></td>
<td>(same as <code>x</code>)</td>
<td>absolute value</td>
<td><code>abs(-17.4)</code></td>
<td>17.4</td>
</tr>
<tr>
<td><code>cbrt(dp)</code></td>
<td><code>dp</code></td>
<td>cube root</td>
<td><code>cbrt(27.0)</code></td>
<td>3</td>
</tr>
<tr>
<td><code>ceil(dp or numeric)</code></td>
<td>(same as input)</td>
<td>smallest integer not less than argument</td>
<td><code>ceil(-42.8)</code></td>
<td>-42</td>
</tr>
<tr>
<td><code>ceiling(dp or numeric)</code></td>
<td>(same as input)</td>
<td>smallest integer not less than argument (alias for <code>ceil</code>)</td>
<td><code>ceiling(-95.3)</code></td>
<td>-95</td>
</tr>
<tr>
<td><code>degrees(dp)</code></td>
<td><code>dp</code></td>
<td>radians to degrees</td>
<td><code>degrees(0.5)</code></td>
<td>28.6478897565412</td>
</tr>
<tr>
<td><code>exp(dp or numeric)</code></td>
<td>(same as input)</td>
<td>exponential</td>
<td><code>exp(1.0)</code></td>
<td>2.71828182845905</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
<td>------------------------------------</td>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td>floor(dp or numeric)</td>
<td>(same as input)</td>
<td>largest integer not greater than argument</td>
<td>floor(-42.8)</td>
<td>-43</td>
</tr>
<tr>
<td>ln(dp or numeric)</td>
<td>(same as input)</td>
<td>natural logarithm</td>
<td>ln(2.0)</td>
<td>0.693147180559945</td>
</tr>
<tr>
<td>log(dp or numeric)</td>
<td>(same as input)</td>
<td>base 10 logarithm</td>
<td>log(100.0)</td>
<td>2</td>
</tr>
<tr>
<td>log(b numeric, x numeric)</td>
<td>numeric</td>
<td>logarithm to base b</td>
<td>log(2.0, 64.0)</td>
<td>6.00000000000</td>
</tr>
<tr>
<td>mod(y, x)</td>
<td>(same as argument types)</td>
<td>remainder of y/x</td>
<td>mod(9,4)</td>
<td>1</td>
</tr>
<tr>
<td>pi()</td>
<td>dp</td>
<td>“π” constant</td>
<td>pi()</td>
<td>3.14159265358979</td>
</tr>
<tr>
<td>power(a dp, b dp)</td>
<td>dp</td>
<td>a raised to the power of b</td>
<td>power(9.0, 3.0)</td>
<td>729</td>
</tr>
<tr>
<td>power(a numeric, b numeric)</td>
<td>numeric</td>
<td>a raised to the power of b</td>
<td>power(9.0, 3.0)</td>
<td>729</td>
</tr>
<tr>
<td>radians(dp)</td>
<td>dp</td>
<td>degrees to radians</td>
<td>radians(45.0)</td>
<td>0.78539816339748</td>
</tr>
<tr>
<td>random()</td>
<td>dp</td>
<td>random value in the range 0.0 &lt;= x &lt; 1.0</td>
<td>random()</td>
<td></td>
</tr>
<tr>
<td>round(dp or numeric)</td>
<td>(same as input)</td>
<td>round to nearest integer</td>
<td>round(42.4)</td>
<td>42</td>
</tr>
<tr>
<td>round(v numeric, s int)</td>
<td>numeric</td>
<td>round to s decimal places</td>
<td>round(42.4382, 2)</td>
<td>42.44</td>
</tr>
<tr>
<td>setseed(dp)</td>
<td>void</td>
<td>set seed for subsequent random() calls (value between 0 and 1.0)</td>
<td>setseed(0.54823)</td>
<td></td>
</tr>
<tr>
<td>sign(dp or numeric)</td>
<td>(same as input)</td>
<td>sign of the argument (-1, 0, +1)</td>
<td>sign(-8.4)</td>
<td>-1</td>
</tr>
<tr>
<td>sqrt(dp or numeric)</td>
<td>(same as input)</td>
<td>square root</td>
<td>sqrt(2.0)</td>
<td>1.4142135623731</td>
</tr>
<tr>
<td>trunc(dp or numeric)</td>
<td>(same as input)</td>
<td>truncate toward zero</td>
<td>trunc(42.8)</td>
<td>42</td>
</tr>
<tr>
<td>trunc(v numeric, s int)</td>
<td>numeric</td>
<td>truncate to s decimal places</td>
<td>trunc(42.4382, 2)</td>
<td>42.43</td>
</tr>
</tbody>
</table>
Finally, Table 9-4 shows the available trigonometric functions. All trigonometric functions take arguments and return values of type `double precision`.

Table 9-4. Trigonometric Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos(x)</td>
<td>inverse cosine</td>
</tr>
<tr>
<td>asin(x)</td>
<td>inverse sine</td>
</tr>
<tr>
<td>atan(x)</td>
<td>inverse tangent</td>
</tr>
<tr>
<td>atan2(y, x)</td>
<td>inverse tangent of y/x</td>
</tr>
<tr>
<td>cos(x)</td>
<td>cosine</td>
</tr>
<tr>
<td>cot(x)</td>
<td>cotangent</td>
</tr>
<tr>
<td>sin(x)</td>
<td>sine</td>
</tr>
<tr>
<td>tan(x)</td>
<td>tangent</td>
</tr>
</tbody>
</table>

9.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types `character`, `character varying`, and `text`. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic space-padding when using the `character` type. Some functions also exist natively for the bit-string types.

SQL defines some string functions with a special syntax wherein certain key words rather than commas are used to separate the arguments. Details are in Table 9-5. These functions are also implemented using the regular syntax for function invocation. (See Table 9-6.)

**Note:** Before PostgreSQL 8.3, these functions would silently accept values of several non-string data types as well, due to the presence of implicit coercions from those data types to `text`. Those coercions have been removed because they frequently caused surprising behaviors. However,
the string concatenation operator (||) still accepts non-string input, so long as at least one input is of a string type, as shown in Table 9-5. For other cases, insert an explicit coercion to text if you need to duplicate the previous behavior.

Table 9-5. SQL String Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td></td>
<td>string</td>
<td>text</td>
<td>String concatenation</td>
</tr>
<tr>
<td>string</td>
<td></td>
<td>non-string</td>
<td>text</td>
<td>String concatenation with one non-string input</td>
</tr>
<tr>
<td>bit_length(string)</td>
<td>int</td>
<td>Number of bits in string</td>
<td>bit_length('jose')</td>
<td>32</td>
</tr>
<tr>
<td>char_length(string)</td>
<td>int</td>
<td>Number of characters in string</td>
<td>char_length('jose')</td>
<td>4</td>
</tr>
<tr>
<td>lower(string)</td>
<td>text</td>
<td>Convert string to lower case</td>
<td>lower('TOM')</td>
<td>tom</td>
</tr>
<tr>
<td>octet_length(string)</td>
<td>int</td>
<td>Number of bytes in string</td>
<td>octet_length('jose')</td>
<td>4</td>
</tr>
<tr>
<td>overlay(string placing string from int [for int])</td>
<td>text</td>
<td>Replace substring</td>
<td>overlay('Txxxxas' placing 'hom' from 2 for 4)</td>
<td>Thomas</td>
</tr>
<tr>
<td>position(substring in string)</td>
<td>int</td>
<td>Location of specified substring</td>
<td>position('om' in 'Thomas')</td>
<td>3</td>
</tr>
<tr>
<td>substring(string [from int] [for int])</td>
<td>text</td>
<td>Extract substring</td>
<td>substring('Thomas' from 2 for 3)</td>
<td>hom</td>
</tr>
<tr>
<td>substring(string from pattern)</td>
<td>text</td>
<td>Extract substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.</td>
<td>substring('Thomas' from '...$')</td>
<td>Thomas</td>
</tr>
</tbody>
</table>
### Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>substring(string</td>
<td>text</td>
<td>Extract substring matching SQL regular expression. See Section 9.7 for</td>
<td>substring('Thomas' from '%'</td>
<td>oma</td>
</tr>
<tr>
<td>from pattern</td>
<td></td>
<td>more information on pattern matching.</td>
<td>'%&quot;o_a&quot;%' for '#'</td>
<td></td>
</tr>
<tr>
<td>for escape)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trim([leading</td>
<td>text</td>
<td>Remove the longest string containing only the characters (a space by</td>
<td>trim(both 'x' from 'xTomxx')</td>
<td>Tom</td>
</tr>
<tr>
<td></td>
<td>trailing</td>
<td></td>
<td>default) from the start/end/both ends of the string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>both]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[characters]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>from string)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>upper(string)</td>
<td>text</td>
<td>Convert string to uppercase</td>
<td>upper('tom')</td>
<td>TOM</td>
</tr>
</tbody>
</table>

Additional string manipulation functions are available and are listed in Table 9-6. Some of them are used internally to implement the SQL-standard string functions listed in Table 9-5.

### Table 9-6. Other String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii(string)</td>
<td>int</td>
<td>ASCII code of the first character of the argument. For UTF8 returns the</td>
<td>ascii('x')</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unicode code point of the character. For other multibyte encodings, the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>argument must be a strictly ASCII character.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>btrim(string text</td>
<td>text</td>
<td>Remove the longest string consisting only of characters in characters</td>
<td>btrim('xyxtrim', 'xy')</td>
<td>xyx</td>
</tr>
<tr>
<td>[, characters</td>
<td></td>
<td>(a space by default) from the start and end of string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>text])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td><code>chr(int)</code></td>
<td><code>text</code></td>
<td>Character with the given code. For UTF8 the argument is treated as a Unicode code point. For other multibyte encodings the argument must designate a strictly ASCII character. The NULL (0) character is not allowed because text data types cannot store such bytes.</td>
<td><code>chr(65)</code></td>
<td>A</td>
</tr>
<tr>
<td><code>convert(string bytea, src_encoding name, dest_encoding name)</code></td>
<td><code>bytea</code></td>
<td>Convert string to <code>dest_encoding</code>. The original encoding is specified by <code>src_encoding</code>. The string must be valid in this encoding. Conversions can be defined by <code>CREATE CONVERSION</code>. Also there are some predefined conversions. See Table 9-7 for available conversions.</td>
<td><code>convert('text_in_utf8', 'UTF8', 'LATIN1')</code></td>
<td>Represented in ISO 8859-1 encoding</td>
</tr>
<tr>
<td><code>convert_from(string bytea, src_encoding name)</code></td>
<td><code>text</code></td>
<td>Convert string to the database encoding. The original encoding is specified by <code>src_encoding</code>. The string must be valid in this encoding.</td>
<td><code>convert_from('text_in_utf8', 'UTF8')</code></td>
<td>Represented in the current database encoding</td>
</tr>
</tbody>
</table>
## Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convert_to(string text, dest_encoding name)</code></td>
<td><code>bytea</code></td>
<td>Convert string to <code>dest_encoding</code>.</td>
<td><code>convert_to('some text', 'UTF8')</code></td>
<td><code>some text represented in the UTF8 encoding</code></td>
</tr>
<tr>
<td><code>decode(string text, type text)</code></td>
<td><code>bytea</code></td>
<td>Decode binary data from string previously encoded with <code>encode</code>. Parameter type is same as in <code>encode</code>.</td>
<td><code>decode('MTIzAAE=','base64')</code></td>
<td><code>123\000\001</code></td>
</tr>
<tr>
<td><code>encode(data bytea, type text)</code></td>
<td><code>text</code></td>
<td>Encode binary data to different representation. Supported types are: <code>base64</code>, <code>hex</code>, <code>escape</code>. Escape merely outputs null bytes as <code>\000</code> and doubles backslashes.</td>
<td><code>encode(E'123\000\001','base64')</code></td>
<td><code>MTIzAAE4</code></td>
</tr>
<tr>
<td><code>initcap(string)</code></td>
<td><code>text</code></td>
<td>Convert the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.</td>
<td><code>initcap('hi THOMAS')</code></td>
<td>Hi Thomas</td>
</tr>
<tr>
<td><code>length(string)</code></td>
<td><code>int</code></td>
<td>Number of characters in string</td>
<td><code>length('jose')</code></td>
<td>4</td>
</tr>
<tr>
<td><code>length(string bytea, encoding name)</code></td>
<td><code>int</code></td>
<td>Number of characters in string in the given encoding. The string must be valid in this encoding.</td>
<td><code>length('jose', 'UTF8')</code></td>
<td>4</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
<td>-----------</td>
</tr>
<tr>
<td><code>lpad()</code></td>
<td><code>text</code></td>
<td>Fill up the string to length length by prepending the characters fill (a space by default). If the string is already longer than length then it is truncated (on the right).</td>
<td><code>lpad('hi', 5, 'xy')</code></td>
<td><code>xyxhi</code></td>
</tr>
<tr>
<td><code>ltrim()</code></td>
<td><code>text</code></td>
<td>Remove the longest string containing only characters from characters (a space by default) from the start of string</td>
<td><code>ltrim('zzzytrim', 'xyz')</code></td>
<td><code>trim</code></td>
</tr>
<tr>
<td><code>md5()</code></td>
<td><code>text</code></td>
<td>Calculates the MD5 hash of string, returning the result in hexadecimal</td>
<td><code>md5('abc')</code></td>
<td><code>900150983cd24fb0 d6963f7d28e17f72</code></td>
</tr>
<tr>
<td><code>pg_client_encoding()</code></td>
<td><code>text</code></td>
<td>Current client encoding name</td>
<td><code>pg_client_encoding()</code></td>
<td><code>SQL_ASCII</code></td>
</tr>
<tr>
<td><code>quote_ident()</code></td>
<td><code>text</code></td>
<td>Return the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled.</td>
<td><code>quote_ident('Foo bar')</code></td>
<td><code>&quot;Foo bar&quot;</code></td>
</tr>
</tbody>
</table>
## Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>quote_literal</code> (string)</td>
<td>text</td>
<td>Return the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled.</td>
<td><code>quote_literal('O’Reilly')</code></td>
<td><code>'O”Reilly'</code></td>
</tr>
<tr>
<td><code>quote_literal</code> (anyelement)</td>
<td>text</td>
<td>Coerce the given value to text and then quote it as a literal. Embedded single-quotes and backslashes are properly doubled.</td>
<td><code>quote_literal(42.5)</code></td>
<td><code>'42.5'</code></td>
</tr>
<tr>
<td><code>regexp_matches</code> (string, pattern text [, flags text])</td>
<td>setof text[]</td>
<td>Return all captured substrings resulting from matching a POSIX regular expression against the string. See Section 9.7.3 for more information.</td>
<td><code>regexp_matches('foobarbequebaz', '(bar)(beque)')</code></td>
<td><code>{bar,beque}</code></td>
</tr>
<tr>
<td><code>regexp_replace</code> (string, pattern text, replacement text [, flags text])</td>
<td>text</td>
<td>Replace substring(s) matching a POSIX regular expression. See Section 9.7.3 for more information.</td>
<td><code>regexp_replace('Thomas', '.[mN]a.', 'M')</code></td>
<td><code>ThMomas'</code></td>
</tr>
<tr>
<td><code>regexp_split_to_array</code> (string, pattern text [, flags text])</td>
<td>text[]</td>
<td>Split string using a POSIX regular expression as the delimiter. See Section 9.7.3 for more information.</td>
<td><code>regexp_split_to_array('hello world', E'\s+')</code></td>
<td><code>{hello,world}</code></td>
</tr>
</tbody>
</table>
## Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>regexp_split_to_table</code></td>
<td><code>setof text</code></td>
<td>Split string using a POSIX regular expression as the delimiter. See Section 9.7.3 for more information.</td>
<td><code>regexp_split_to_table('hello world', E'\s+')</code></td>
<td><code>helloworld (2 rows)</code></td>
</tr>
<tr>
<td><code>repeat</code></td>
<td><code>text</code></td>
<td>Repeat string the specified number of times</td>
<td><code>repeat('Pg', 4)</code></td>
<td><code>PgPgPgPg</code></td>
</tr>
<tr>
<td><code>replace</code></td>
<td><code>text</code></td>
<td>Replace all occurrences in string of substring from with substring to</td>
<td><code>replace('abcdefabcdef', 'cd', 'XX')</code></td>
<td><code>abXXefabXXef</code></td>
</tr>
<tr>
<td><code>rpad</code></td>
<td><code>text</code></td>
<td>Fill up the string to length length by appending the characters fill (a space by default). If the string is already longer than length then it is truncated.</td>
<td><code>rpad('hi', 5, 'xy')</code></td>
<td><code>hixyx</code></td>
</tr>
<tr>
<td><code>rtrim</code></td>
<td><code>text</code></td>
<td>Remove the longest string containing only characters from characters (a space by default) from the end of string</td>
<td><code>rtrim('trimxxxx', 'x')</code></td>
<td><code>trim</code></td>
</tr>
<tr>
<td><code>split_part</code></td>
<td><code>text</code></td>
<td>Split string on delimiter and return the given field (counting from one)</td>
<td><code>split_part('abdef~0~ghi', '~0~', 2)</code></td>
<td><code>ab0de~0~ghi</code></td>
</tr>
<tr>
<td><code>strpos</code></td>
<td><code>int</code></td>
<td>Location of specified substring (same as position(substring in string), but note the reversed argument order)</td>
<td><code>strpos('high', 'ig')</code></td>
<td>2</td>
</tr>
</tbody>
</table>

143
Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>substr(string,</td>
<td>text</td>
<td>Extract substring (same as substring(string from from for count))</td>
<td>substr('alphabet',</td>
<td>ph,</td>
</tr>
<tr>
<td>from [,</td>
<td></td>
<td></td>
<td>3, 2)</td>
<td></td>
</tr>
<tr>
<td>count])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to_ascii(string</td>
<td>text</td>
<td>Convert string to ASCII from another encoding (only supports conversion from</td>
<td>to_ascii('Karel')</td>
<td>Karel</td>
</tr>
<tr>
<td>text [ ,</td>
<td></td>
<td>LATIN1, LATIN2, LATIN9, and WIN1250 encodings)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>encoding text])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to_hex(number int</td>
<td>text</td>
<td>Convert number to its equivalent hexadecimal representation</td>
<td>to_hex(2147483647)</td>
<td>ffffff</td>
</tr>
<tr>
<td>or bigint)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>translate(string</td>
<td>text</td>
<td>Any character in string that matches a character in the from set is</td>
<td>translate('12345',</td>
<td>a23x5</td>
</tr>
<tr>
<td>text, from text,</td>
<td></td>
<td>replaced by the corresponding character in the to set</td>
<td>'14', 'ax')</td>
<td></td>
</tr>
<tr>
<td>to text)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9-7. Built-in Conversions

<table>
<thead>
<tr>
<th>Conversion Name</th>
<th>Source Encoding</th>
<th>Destination Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii_to_mic</td>
<td>SQL_ASCII</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>ascii_to_utf8</td>
<td>SQL_ASCII</td>
<td>UTF8</td>
</tr>
<tr>
<td>big5_to_euc_tw</td>
<td>BIG5</td>
<td>EUC_TW</td>
</tr>
<tr>
<td>big5_to_mic</td>
<td>BIG5</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>big5_to_utf8</td>
<td>BIG5</td>
<td>UTF8</td>
</tr>
<tr>
<td>euc_cn_to_mic</td>
<td>EUC_CN</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>euc_cn_to_utf8</td>
<td>EUC_CN</td>
<td>UTF8</td>
</tr>
<tr>
<td>euc_jp_to_mic</td>
<td>EUC_JP</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>euc_jp_to_sjis</td>
<td>EUC_JP</td>
<td>SJIS</td>
</tr>
<tr>
<td>euc_jp_to_utf8</td>
<td>EUC_JP</td>
<td>UTF8</td>
</tr>
<tr>
<td>euc_kr_to_mic</td>
<td>EUC_KR</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>euc_kr_to_utf8</td>
<td>EUC_KR</td>
<td>UTF8</td>
</tr>
<tr>
<td>euc_tw_to_big5</td>
<td>EUC_TW</td>
<td>BIG5</td>
</tr>
<tr>
<td>euc_tw_to_mic</td>
<td>EUC_TW</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>euc_tw_to_utf8</td>
<td>EUC_TW</td>
<td>UTF8</td>
</tr>
<tr>
<td>Conversion Name</td>
<td>Source Encoding</td>
<td>Destination Encoding</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>gb18030_to_utf8</td>
<td>GB18030</td>
<td>UTF8</td>
</tr>
<tr>
<td>gbk_to_utf8</td>
<td>GBK</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_10_to_utf8</td>
<td>LATIN6</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_13_to_utf8</td>
<td>LATIN7</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_14_to_utf8</td>
<td>LATIN8</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_15_to_utf8</td>
<td>LATIN9</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_16_to_utf8</td>
<td>LATIN10</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_1_to_mic</td>
<td>LATIN1</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>iso_8859_1_to_utf8</td>
<td>LATIN1</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_2_to_mic</td>
<td>LATIN2</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>iso_8859_2_to_utf8</td>
<td>LATIN2</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_3_to_mic</td>
<td>LATIN3</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>iso_8859_3_to_utf8</td>
<td>LATIN3</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_4_to_mic</td>
<td>LATIN4</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>iso_8859_4_to_utf8</td>
<td>LATIN4</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_5_to_koi8_r</td>
<td>ISO_8859_5</td>
<td>KOI8</td>
</tr>
<tr>
<td>iso_8859_5_to_mic</td>
<td>ISO_8859_5</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>iso_8859_5_to_utf8</td>
<td>ISO_8859_5</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_5_to_windows_1251</td>
<td>ISO_8859_5</td>
<td>WIN1251</td>
</tr>
<tr>
<td>iso_8859_5_to_windows_866</td>
<td>ISO_8859_5</td>
<td>WIN866</td>
</tr>
<tr>
<td>iso_8859_6_to_utf8</td>
<td>ISO_8859_6</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_7_to_utf8</td>
<td>ISO_8859_7</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_8_toUtf8</td>
<td>ISO_8859_8</td>
<td>UTF8</td>
</tr>
<tr>
<td>iso_8859_9_to_utf8</td>
<td>LATIN5</td>
<td>UTF8</td>
</tr>
<tr>
<td>johab_to_utf8</td>
<td>JOHAB</td>
<td>UTF8</td>
</tr>
<tr>
<td>koi8_r_to_iso_8859_5</td>
<td>KOI8</td>
<td>ISO_8859_5</td>
</tr>
<tr>
<td>koi8_r_to_mic</td>
<td>KOI8</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>koi8_r_to_utf8</td>
<td>KOI8</td>
<td>UTF8</td>
</tr>
<tr>
<td>koi8_r_to_windows_1251</td>
<td>KOI8</td>
<td>WIN1251</td>
</tr>
<tr>
<td>koi8_r_to_windows_866</td>
<td>KOI8</td>
<td>WIN866</td>
</tr>
<tr>
<td>mic_to_ascii</td>
<td>MULE_INTERNAL</td>
<td>SQL_ASCII</td>
</tr>
<tr>
<td>mic_to_big5</td>
<td>MULE_INTERNAL</td>
<td>BIG5</td>
</tr>
<tr>
<td>mic_to_euc_cn</td>
<td>MULE_INTERNAL</td>
<td>EUC_CN</td>
</tr>
<tr>
<td>mic_to_euc_jp</td>
<td>MULE_INTERNAL</td>
<td>EUC_JP</td>
</tr>
<tr>
<td>mic_to_euc_kr</td>
<td>MULE_INTERNAL</td>
<td>EUC_KR</td>
</tr>
<tr>
<td>mic_to_euc_tw</td>
<td>MULE_INTERNAL</td>
<td>EUC_TW</td>
</tr>
<tr>
<td>mic_to_iso_8859_1</td>
<td>MULE_INTERNAL</td>
<td>LATIN1</td>
</tr>
<tr>
<td>mic_to_iso_8859_2</td>
<td>MULE_INTERNAL</td>
<td>LATIN2</td>
</tr>
<tr>
<td>Conversion Name</td>
<td>Source Encoding</td>
<td>Destination Encoding</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>mic_to_iso_8859_3</td>
<td>MULE_INTERNAL</td>
<td>LATIN3</td>
</tr>
<tr>
<td>mic_to_iso_8859_4</td>
<td>MULE_INTERNAL</td>
<td>LATIN4</td>
</tr>
<tr>
<td>mic_to_iso_8859_5</td>
<td>MULE_INTERNAL</td>
<td>ISO_8859_5</td>
</tr>
<tr>
<td>mic_to_koi8_r</td>
<td>MULE_INTERNAL</td>
<td>KOI8</td>
</tr>
<tr>
<td>mic_to_sjis</td>
<td>MULE_INTERNAL</td>
<td>SJIS</td>
</tr>
<tr>
<td>mic_to_windows_1250</td>
<td>MULE_INTERNAL</td>
<td>WIN1250</td>
</tr>
<tr>
<td>mic_to_windows_1251</td>
<td>MULE_INTERNAL</td>
<td>WIN1251</td>
</tr>
<tr>
<td>mic_to_windows_866</td>
<td>MULE_INTERNAL</td>
<td>WIN866</td>
</tr>
<tr>
<td>sjis_to_eucjp</td>
<td>SJIS</td>
<td>EUC_JP</td>
</tr>
<tr>
<td>sjis_to_mic</td>
<td>SJIS</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>sjis_to_utf8</td>
<td>SJIS</td>
<td>UTF8</td>
</tr>
<tr>
<td>tcvn_to_utf8</td>
<td>WIN1258</td>
<td>UTF8</td>
</tr>
<tr>
<td>uhc_to_utf8</td>
<td>UHC</td>
<td>UTF8</td>
</tr>
<tr>
<td>utf8_to_ascii</td>
<td>UTF8</td>
<td>SQL_ASCII</td>
</tr>
<tr>
<td>utf8_to_big5</td>
<td>UTF8</td>
<td>BIG5</td>
</tr>
<tr>
<td>utf8_to_euc_cn</td>
<td>UTF8</td>
<td>EUC_CN</td>
</tr>
<tr>
<td>utf8_to_eucjp</td>
<td>UTF8</td>
<td>EUC_JP</td>
</tr>
<tr>
<td>utf8_to_euckr</td>
<td>UTF8</td>
<td>EUC_KR</td>
</tr>
<tr>
<td>utf8_to_euctw</td>
<td>UTF8</td>
<td>EUC_TW</td>
</tr>
<tr>
<td>utf8_to_gb18030</td>
<td>UTF8</td>
<td>GB18030</td>
</tr>
<tr>
<td>utf8_to_gbk</td>
<td>UTF8</td>
<td>GBK</td>
</tr>
<tr>
<td>utf8_to_iso_8859_1</td>
<td>UTF8</td>
<td>LATIN1</td>
</tr>
<tr>
<td>utf8_to_iso_8859_10</td>
<td>UTF8</td>
<td>LATIN6</td>
</tr>
<tr>
<td>utf8_to_iso_8859_13</td>
<td>UTF8</td>
<td>LATIN7</td>
</tr>
<tr>
<td>utf8_to_iso_8859_14</td>
<td>UTF8</td>
<td>LATIN8</td>
</tr>
<tr>
<td>utf8_to_iso_8859_15</td>
<td>UTF8</td>
<td>LATIN9</td>
</tr>
<tr>
<td>utf8_to_iso_8859_16</td>
<td>UTF8</td>
<td>LATIN10</td>
</tr>
<tr>
<td>utf8_to_iso_8859_2</td>
<td>UTF8</td>
<td>LATIN2</td>
</tr>
<tr>
<td>utf8_to_iso_8859_3</td>
<td>UTF8</td>
<td>LATIN3</td>
</tr>
<tr>
<td>utf8_to_iso_8859_4</td>
<td>UTF8</td>
<td>LATIN4</td>
</tr>
<tr>
<td>utf8_to_iso_8859_5</td>
<td>UTF8</td>
<td>ISO_8859_5</td>
</tr>
<tr>
<td>utf8_to_iso_8859_6</td>
<td>UTF8</td>
<td>ISO_8859_6</td>
</tr>
<tr>
<td>utf8_to_iso_8859_7</td>
<td>UTF8</td>
<td>ISO_8859_7</td>
</tr>
<tr>
<td>utf8_to_iso_8859_8</td>
<td>UTF8</td>
<td>ISO_8859_8</td>
</tr>
<tr>
<td>utf8_to_iso_8859_9</td>
<td>UTF8</td>
<td>LATIN5</td>
</tr>
<tr>
<td>utf8_to_johab</td>
<td>UTF8</td>
<td>JOHAB</td>
</tr>
<tr>
<td>utf8_to_koi8_r</td>
<td>UTF8</td>
<td>KOI8</td>
</tr>
<tr>
<td>utf8_to_sjis</td>
<td>UTF8</td>
<td>SJIS</td>
</tr>
<tr>
<td>utf8_to_tcvn</td>
<td>UTF8</td>
<td>WIN1258</td>
</tr>
<tr>
<td>utf8_to_uhc</td>
<td>UTF8</td>
<td>UHC</td>
</tr>
<tr>
<td>utf8_to_windows_1250</td>
<td>UTF8</td>
<td>WIN1250</td>
</tr>
<tr>
<td>utf8_to_windows_1251</td>
<td>UTF8</td>
<td>WIN1251</td>
</tr>
</tbody>
</table>
### Conversion Name Table

<table>
<thead>
<tr>
<th>Conversion Name</th>
<th>Source Encoding</th>
<th>Destination Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>utf8_to_windows_1252</td>
<td>UTF8</td>
<td>WIN1252</td>
</tr>
<tr>
<td>utf8_to_windows_1253</td>
<td>UTF8</td>
<td>WIN1253</td>
</tr>
<tr>
<td>utf8_to_windows_1254</td>
<td>UTF8</td>
<td>WIN1254</td>
</tr>
<tr>
<td>utf8_to_windows_1255</td>
<td>UTF8</td>
<td>WIN1255</td>
</tr>
<tr>
<td>utf8_to_windows_1256</td>
<td>UTF8</td>
<td>WIN1256</td>
</tr>
<tr>
<td>utf8_to_windows_1257</td>
<td>UTF8</td>
<td>WIN1257</td>
</tr>
<tr>
<td>utf8_to_windows_866</td>
<td>UTF8</td>
<td>WIN866</td>
</tr>
<tr>
<td>utf8_to_windows_874</td>
<td>UTF8</td>
<td>WIN874</td>
</tr>
<tr>
<td>windows_1250_to_iso_8859</td>
<td>WIN1250</td>
<td>LATIN2</td>
</tr>
<tr>
<td>windows_1250_to_mic</td>
<td>WIN1250</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>windows_1250_to_utf8</td>
<td>WIN1250</td>
<td>UTF8</td>
</tr>
<tr>
<td>windows_1251_to_iso_8859</td>
<td>WIN1251</td>
<td>ISO_8859_5</td>
</tr>
<tr>
<td>windows_1251_to_koi8_r</td>
<td>WIN1251</td>
<td>KOI8</td>
</tr>
<tr>
<td>windows_1251_to_mic</td>
<td>WIN1251</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>windows_1251_to_utf8</td>
<td>WIN1251</td>
<td>UTF8</td>
</tr>
<tr>
<td>windows_1252_to_utf8</td>
<td>WIN1252</td>
<td>UTF8</td>
</tr>
<tr>
<td>windows_1256_to_utf8</td>
<td>WIN1256</td>
<td>UTF8</td>
</tr>
<tr>
<td>windows_866_to_iso_8859</td>
<td>WIN866</td>
<td>ISO_8859_5</td>
</tr>
<tr>
<td>windows_866_to_koi8_r</td>
<td>WIN866</td>
<td>KOI8</td>
</tr>
<tr>
<td>windows_866_to_mic</td>
<td>WIN866</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>windows_866_to_utf8</td>
<td>WIN866</td>
<td>UTF8</td>
</tr>
<tr>
<td>windows_866_to_windows_1251</td>
<td>WIN866</td>
<td>WIN866</td>
</tr>
<tr>
<td>windows_874_to_utf8</td>
<td>WIN874</td>
<td>UTF8</td>
</tr>
<tr>
<td>euc_jis_2004_to_utf8</td>
<td>EUC_JIS_2004</td>
<td>UTF8</td>
</tr>
<tr>
<td>ut8_to_euc_jis_2004</td>
<td>UTF8</td>
<td>EUC_JIS_2004</td>
</tr>
<tr>
<td>shift_jis_2004_to_utf8</td>
<td>SHIFT_JIS_2004</td>
<td>UTF8</td>
</tr>
<tr>
<td>ut8_to_shift_jis_2004</td>
<td>UTF8</td>
<td>SHIFT_JIS_2004</td>
</tr>
<tr>
<td>euc_jis_2004_to_shift_jis</td>
<td>EUC_JIS_2004</td>
<td>SHIFT_JIS_2004</td>
</tr>
<tr>
<td>shift_jis_2004_to_euc_jis</td>
<td>SHIFT_JIS_2004</td>
<td>EUC_JIS_2004</td>
</tr>
</tbody>
</table>

**Notes:**

a. The conversion names follow a standard naming scheme: The official name of the source encoding with all non-alphanumeric characters replaced by underscores followed by _to_ followed by the equally processed destination encoding name. Therefore the names might deviate from the customary encoding names.
9.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating values of type `bytea`. SQL defines some string functions with a special syntax where certain key words rather than commas are used to separate the arguments. Details are in Table 9-8. Some functions are also implemented using the regular syntax for function invocation. (See Table 9-9.)

Table 9-8. SQL Binary String Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>`string</td>
<td></td>
<td>string`</td>
<td><code>bytea</code></td>
<td>String concatenation</td>
</tr>
<tr>
<td><code>get_bit(string, int offset)</code></td>
<td><code>int</code></td>
<td>Extract bit from string</td>
<td><code>get_bit(E'Thomas'::bytea, 45)</code></td>
<td></td>
</tr>
<tr>
<td><code>get_byte(string, int offset)</code></td>
<td><code>int</code></td>
<td>Extract byte from string</td>
<td><code>get_byte(E'Thomas'::bytea, 4)</code></td>
<td></td>
</tr>
<tr>
<td><code>octet_length(string)</code></td>
<td><code>int</code></td>
<td>Number of bytes in binary string</td>
<td><code>octet_length(E'jose'::bytea)</code></td>
<td></td>
</tr>
<tr>
<td><code>position(substring in string)</code></td>
<td><code>int</code></td>
<td>Location of specified substring</td>
<td><code>position(E'om'::bytea in E'Thomas'::bytea)</code></td>
<td></td>
</tr>
<tr>
<td><code>set_bit(string, int offset, newvalue)</code></td>
<td><code>bytea</code></td>
<td>Set bit in string</td>
<td><code>set_bit(E'Thomas'::bytea, 45, 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>set_byte(string, int offset, newvalue)</code></td>
<td><code>bytea</code></td>
<td>Set byte in string</td>
<td><code>set_byte(E'Thomas'::bytea, 4, 64)</code></td>
<td></td>
</tr>
<tr>
<td><code>substring(string [from int] [for int])</code></td>
<td><code>bytea</code></td>
<td>Extract substring</td>
<td><code>substring(E'Thomas'::bytea from 2 for 3)</code></td>
<td></td>
</tr>
<tr>
<td><code>trim([both] bytes from string)</code></td>
<td><code>bytea</code></td>
<td>Remove the longest string containing only the bytes in bytes from the start and end of string</td>
<td><code>trim(E''000'':bytea from E''000Tom'':bytea)</code></td>
<td></td>
</tr>
</tbody>
</table>

Additional binary string manipulation functions are available and are listed in Table 9-9. Some of them are used internally to implement the SQL-standard string functions listed in Table 9-8.

Table 9-9. Other Binary String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
### 9.6. Bit String Functions and Operators

This section describes functions and operators for examining and manipulating bit strings, that is, values of the types \texttt{bit} and \texttt{bit varying}. Aside from the usual comparison operators, the operators shown in Table 9-10 can be used. Bit string operands of \&, |, and \# must be of equal length. When bit shifting, the original length of the string is preserved, as shown in the examples.

#### Table 9-10. Bit String Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>||</td>
<td>concatenation</td>
<td>\texttt{B’10001’</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>\texttt{B’10001’ &amp; B’01101’}</td>
<td>00001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td>\texttt{B’10001’</td>
</tr>
</tbody>
</table>
## Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>bitwise XOR</td>
<td>B'10001' #</td>
<td>11100</td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>~ B'10001'</td>
<td>01110</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bitwise shift left</td>
<td>B'10001' &lt;&lt; 3</td>
<td>01000</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bitwise shift right</td>
<td>B'10001' &gt;&gt; 2</td>
<td>00100</td>
</tr>
</tbody>
</table>

The following SQL-standard functions work on bit strings as well as character strings: `length`, `bit_length`, `octet_length`, `position`, `substring`.

In addition, it is possible to cast integral values to and from type `bit`. Some examples:

```
44::bit(10)       0000101100
44::bit(3)        100
cast(-44 as bit(12)) 111111010100
'1110)::bit(4)::integer 14
```

Note that casting to just “bit” means casting to `bit(1)`, and so it will deliver only the least significant bit of the integer.

**Note:** Prior to PostgreSQL 8.0, casting an integer to `bit(n)` would copy the leftmost `n` bits of the integer, whereas now it copies the rightmost `n` bits. Also, casting an integer to a bit string width wider than the integer itself will sign-extend on the left.

### 9.7. Pattern Matching

There are three separate approaches to pattern matching provided by PostgreSQL: the traditional SQL `LIKE` operator, the more recent `SIMILAR TO` operator (added in SQL:1999), and POSIX-style regular expressions. Aside from the basic “does this string match this pattern?” operators, functions are available to extract or replace matching substrings and to split a string at the matches.

**Tip:** If you have pattern matching needs that go beyond this, consider writing a user-defined function in Perl or Tcl.

#### 9.7.1. LIKE

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

Every `pattern` defines a set of strings. The `LIKE` expression returns true if the `string` is contained in the set of strings represented by `pattern`. (As expected, the `NOT LIKE` expression returns false if `LIKE` returns true, and vice versa. An equivalent expression is `NOT (string LIKE pattern).`)

If `pattern` does not contain percent signs or underscore, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in `pattern` stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters.

Some examples:
LIKE pattern matches always cover the entire string. To match a sequence anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in pattern must be preceded by the escape character. The default escape character is the backslash but a different one can be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in an SQL statement (assuming escape string syntax is used, see Section 4.1.2.1). Thus, writing a pattern that actually matches a literal backslash means writing four backslashes in the statement. You can avoid this by selecting a different escape character with ESCAPE; then a backslash is not special to LIKE anymore. (But it is still special to the string literal parser, so you still need two of them.)

It’s also possible to select no escape character by writing ESCAPE ". This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

The key word ILIKE can be used instead of LIKE to make the match case-insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator ~~~ is equivalent to LIKE, and ~~~* corresponds to ILIKE. There are also !~~ and !~~* operators that represent NOT LIKE and NOT ILIKE, respectively. All of these operators are PostgreSQL-specific.

### 9.7.2. SIMILAR TO Regular Expressions

`string SIMILAR TO pattern [ESCAPE escape-character]`

`string NOT SIMILAR TO pattern [ESCAPE escape-character]`

The SIMILAR TO operator returns true or false depending on whether its pattern matches the given string. It is much like LIKE, except that it interprets the pattern using the SQL standard’s definition of a regular expression. SQL regular expressions are a curious cross between LIKE notation and common regular expression notation.

Like LIKE, the SIMILAR TO operator succeeds only if its pattern matches the entire string; this is unlike common regular expression practice, wherein the pattern can match any part of the string. Also like LIKE, SIMILAR TO uses _ and % as wildcard characters denoting any single character and any string, respectively (these are comparable to . and .* in POSIX regular expressions).

In addition to these facilities borrowed from LIKE, SIMILAR TO supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

- | denotes alternation (either of two alternatives).
- * denotes repetition of the previous item zero or more times.
- + denotes repetition of the previous item one or more times.
- Parentheses () can be used to group items into a single logical item.
• A bracket expression [ . . .] specifies a character class, just as in POSIX regular expressions.

Notice that bounded repetition ( ? and { . . .}) are not provided, though they exist in POSIX. Also, the
dot ( .) is not a metacharacter.

As with LIKE, a backslash disables the special meaning of any of these metacharacters; or a different
escape character can be specified with ESCAPE.

Some examples:

'abc' SIMILAR TO 'abc'  true
'abc' SIMILAR TO 'a'  false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'  false

The substring function with three parameters, substring(string from pattern for
escape-character), provides extraction of a substring that matches an SQL regular expression
pattern. As with SIMILAR TO, the specified pattern must match to the entire data string, else the
function fails and returns null. To indicate the part of the pattern that should be returned on success,
the pattern must contain two occurrences of the escape character followed by a double quote (%). The
text matching the portion of the pattern between these markers is returned.

Some examples:

substring('foobar' from '%"o_b"%' for '#')  oob
substring('foobar' from '#"o_b"%' for '#')  NULL

9.7.3. POSIX Regular Expressions

Table 9-11 lists the available operators for pattern matching using POSIX regular expressions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Matches regular expression, case sensitive</td>
<td>'thomas' ~ '.<em>thomas.</em>'</td>
</tr>
<tr>
<td>~*</td>
<td>Matches regular expression, case insensitive</td>
<td>'thomas' ~* '.<em>Thomas.</em>'</td>
</tr>
<tr>
<td>!~</td>
<td>Does not match regular expression, case sensitive</td>
<td>'thomas' !~ '.<em>Thomas.</em>'</td>
</tr>
<tr>
<td>!~*</td>
<td>Does not match regular expression, case insensitive</td>
<td>'thomas' !~* '.<em>vadim.</em>'</td>
</tr>
</tbody>
</table>

POSIX regular expressions provide a more powerful means for pattern matching than the LIKE and
SIMILAR TO operators. Many Unix tools such as egrep, sed, or awk use a pattern matching language
that is similar to the one described here.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a
regular set). A string is said to match a regular expression if it is a member of the regular set described
by the regular expression. As with LIKE, pattern characters match string characters exactly unless
they are special characters in the regular expression language — but regular expressions use different
special characters than LIKE does. Unlike LIKE patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Some examples:

'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^'(b|c) false

The POSIX pattern language is described in much greater detail below.

The substring function with two parameters, substring(string from pattern), provides extraction of a substring that matches a POSIX regular expression pattern. It returns null if there is no match, otherwise the portion of the text that matched the pattern. But if the pattern contains any parentheses, the portion of the text that matched the first parenthesized subexpression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception. If you need parentheses in the pattern before the subexpression you want to extract, see the non-capturing parentheses described below.

Some examples:

substring('foobar' from 'o.b')    oob
substring('foobar' from 'o(.)b')   o

The regexp_replace function provides substitution of new text for substrings that match POSIX regular expression patterns. It has the syntax regexp_replace(source, pattern, replacement []). The source string is returned unchanged if there is no match to the pattern. If there is a match, the source string is returned with the replacement string substituted for the matching substring. The replacement string can contain \n, where n is 1 through 9, to indicate that the source substring matching the n’th parenthesized subexpression of the pattern should be inserted, and it can contain \ to indicate that the substring matching the entire pattern should be inserted. Write \ if you need to put a literal backslash in the replacement text. (As always, remember to double backslashes written in literal constant strings, assuming escape string syntax is used.) The flags parameter is an optional text string containing zero or more single-letter flags that change the function’s behavior. Flag i specifies case-insensitive matching, while flag g specifies replacement of each matching substring rather than only the first one. Other supported flags are described in Table 9-19.

Some examples:

regexp_replace('foobarbaz', 'b..', 'X')    fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g') fooXX
regexp_replace('foobarbaz', 'b(...)', E'X\1Y', 'g') fooXarYXazY

The regexp_matches function returns all of the captured substrings resulting from matching a POSIX regular expression pattern. It has the syntax regexp_matches(string, pattern [, flags ]). If there is no match to the pattern, the function returns no rows. If there is a match, the function returns a text array whose n’th element is the substring matching the n’th parenthesized subexpression
of the pattern (not counting “non-capturing” parentheses; see below for details). If the pattern does
not contain any parenthesized subexpressions, then the result is a single-element text array containing
the substring matching the whole pattern. The flags parameter is an optional text string containing
zero or more single-letter flags that change the function’s behavior. Flag g causes the function to find
each match in the string, not only the first one, and return a row for each such match. Other supported
flags are described in Table 9-19.

Some examples:

```sql
SELECT regexp_matches('foobardequebaz', '(bar)(beque)');
regexp_matches
----------------
{bar,beque}
(1 row)
```

```sql
SELECT regexp_matches('foobardequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
regexp_matches
----------------
{bar,beque}
{bazil,barf}
(2 rows)
```

```sql
SELECT regexp_matches('foobardequebaz', 'barbeque');
regexp_matches
----------------
{barbeque}
(1 row)
```

The `regexp_split_to_table` function splits a string using a POSIX regular expression pattern as
a delimiter. It has the syntax `regexp_split_to_table(string, pattern [, flags ])`. If there is
no match to the pattern, the function returns the string. If there is at least one match, for each
match it returns the text from the end of the last match (or the beginning of the string) to the beginning
of the match. When there are no more matches, it returns the text from the end of the last match to the
end of the string. The flags parameter is an optional text string containing zero or more single-letter
flags that change the function’s behavior. `regexp_split_to_table` supports the flags described in
Table 9-19.

The `regexp_split_to_array` function behaves the same as `regexp_split_to_table`,
except that `regexp_split_to_array` returns its result as an array of text. It has the syntax
`regexp_split_to_array(string, pattern [, flags ])`. The parameters are the same as for
`regexp_split_to_table`.

Some examples:

```sql
SELECT foo FROM regexp_split_to_table('the quick brown fox jumped over the lazy dog', E'\s+');
foo
--------
the
quick
brown
fox
jumped
over
the
lazy
dog
```

---

154
As the last example demonstrates, the regexp split functions ignore zero-length matches that occur at the start or end of the string or immediately after a previous match. This is contrary to the strict definition of regexp matching that is implemented by \texttt{regexp_matches}, but is usually the most convenient behavior in practice. Other software systems such as Perl use similar definitions.

### 9.7.3.1. Regular Expression Details

PostgreSQL’s regular expressions are implemented using a package written by Henry Spencer. Much of the description of regular expressions below is copied verbatim from his manual entry.

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: \textit{extended} REs or EREs (roughly those of egrep), and \textit{basic} REs or BREs (roughly those of ed). PostgreSQL supports both forms, and also implements some extensions that are not in the POSIX standard, but have become widely used anyway due to their availability in programming languages such as Perl and Tcl. REs using these non-POSIX extensions are called \textit{advanced} REs or AREs in this documentation. AREs are almost an exact superset of EREs, but BREs have several notational incompatibilities (as well as being much more limited). We first describe the ARE and ERE forms, noting features that apply only to AREs, and then describe how BREs differ.

\textbf{Note:} The form of regular expressions accepted by PostgreSQL can be chosen by setting the \texttt{regex_flavor} run-time parameter. The usual setting is \texttt{advanced}, but one might choose \texttt{extended} for maximum backwards compatibility with pre-7.4 releases of PostgreSQL.
A regular expression is defined as one or more branches, separated by \|. It matches anything that matches one of the branches.

A branch is zero or more quantified atoms or constraints, concatenated. It matches a match for the first, followed by a match for the second, etc; an empty branch matches the empty string.

A quantified atom is an atom possibly followed by a single quantifier. Without a quantifier, it matches a match for the atom. With a quantifier, it can match some number of matches of the atom. An atom can be any of the possibilities shown in Table 9-12. The possible quantifiers and their meanings are shown in Table 9-13.

A constraint matches an empty string, but matches only when specific conditions are met. A constraint can be used where an atom could be used, except it cannot be followed by a quantifier. The simple constraints are shown in Table 9-14; some more constraints are described later.

### Table 9-12. Regular Expression Atoms

<table>
<thead>
<tr>
<th>Atom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(re)</td>
<td>(where re is any regular expression) matches a match for re, with the match noted for possible reporting</td>
</tr>
<tr>
<td>(?:re)</td>
<td>as above, but the match is not noted for reporting (a “non-capturing” set of parentheses) (AREs only)</td>
</tr>
<tr>
<td>.</td>
<td>matches any single character</td>
</tr>
<tr>
<td>[chars]</td>
<td>a bracket expression, matching any one of the chars (see Section 9.7.3.2 for more detail)</td>
</tr>
<tr>
<td>\k</td>
<td>(where k is a non-alphanumeric character) matches that character taken as an ordinary character, e.g. \ \ matches a backslash character</td>
</tr>
<tr>
<td>\c</td>
<td>where c is alphanumeric (possibly followed by other characters) is an escape, see Section 9.7.3.3 (AREs only; in EREs and BREs, this matches c)</td>
</tr>
<tr>
<td>{</td>
<td>when followed by a character other than a digit, matches the left-brace character ; when followed by a digit, it is the beginning of a bound (see below)</td>
</tr>
<tr>
<td>x</td>
<td>where x is a single character with no other significance, matches that character</td>
</tr>
</tbody>
</table>

An RE cannot end with \. 

**Note:** Remember that the backslash (\) already has a special meaning in PostgreSQL string literals. To write a pattern constant that contains a backslash, you must write two backslashes in the statement, assuming escape string syntax is used (see Section 4.1.2.1).

### Table 9-13. Regular Expression Quantifiers

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Matches</th>
</tr>
</thead>
</table>

156
Quantifier | Matches
---|---
* | a sequence of 0 or more matches of the atom
+ | a sequence of 1 or more matches of the atom
? | a sequence of 0 or 1 matches of the atom
\{m\} | a sequence of exactly \(m\) matches of the atom
\{m,\} | a sequence of \(m\) or more matches of the atom
\{m, n\} | a sequence of \(m\) through \(n\) (inclusive) matches of the atom; \(m\) cannot exceed \(n\)
*? | non-greedy version of *
+? | non-greedy version of +
?? | non-greedy version of ?
\{m\}? | non-greedy version of \{m\}
\{m,\}? | non-greedy version of \{m,\}
\{m, n\}? | non-greedy version of \{m, n\}

The forms using \{...\} are known as **bounds**. The numbers \(m\) and \(n\) within a bound are unsigned decimal integers with permissible values from 0 to 255 inclusive.

**Non-greedy** quantifiers (available in AREs only) match the same possibilities as their corresponding normal (greedy) counterparts, but prefer the smallest number rather than the largest number of matches. See Section 9.7.3.5 for more detail.

**Note:** A quantifier cannot immediately follow another quantifier. A quantifier cannot begin an expression or subexpression or follow ^ or |.

### Table 9-14. Regular Expression Constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>matches at the beginning of the string</td>
</tr>
<tr>
<td>$</td>
<td>matches at the end of the string</td>
</tr>
<tr>
<td>((?=xe))</td>
<td><em>positive lookahead</em> matches at any point where a substring matching (xe) begins (AREs only)</td>
</tr>
<tr>
<td>((?!xe))</td>
<td><em>negative lookahead</em> matches at any point where no substring matching (xe) begins (AREs only)</td>
</tr>
</tbody>
</table>

Lookahead constraints cannot contain back references (see Section 9.7.3.3), and all parentheses within them are considered non-capturing.

#### 9.7.3.2. Bracket Expressions

A **bracket expression** is a list of characters enclosed in \[\]. It normally matches any single character from the list (but see below). If the list begins with ^, it matches any single character not from the rest of the list. If two characters in the list are separated by -, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g. [0-9] in ASCII matches any decimal digit. It is illegal for two ranges to share an endpoint, e.g. a-c-e. Ranges are very collating-sequence-dependent, so portable programs should avoid relying on them.
To include a literal ] in the list, make it the first character (following a possible ^). To include a literal ^, make it the first or last character, or the second endpoint of a range. To use a literal ^ as the first endpoint of a range, enclose it in [ and ] to make it a collating element (see below). With the exception of these characters, some combinations using ] (see next paragraphs), and escapes (AREs only), all other special characters lose their special significance within a bracket expression. In particular, \ is not special when following ERE or BRE rules, though it is special (as introducing an escape) in AREs.

Within a bracket expression, a collating element (a character, a multiple-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in [ and ] stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression’s list. A bracket expression containing a multiple-character collating element can thus match more than one character, e.g. if the collating sequence includes a ch collating element, then the RE [ [.ch. ]] matches the first five characters of chchcc.

Note: PostgreSQL currently has no multicharacter collating elements. This information describes possible future behavior.

Within a bracket expression, a collating element enclosed in [ = and = ] is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were [ and ]). For example, if o and ^ are the members of an equivalence class, then [ [ = o = ] ] , [ [ = ^ = ] ] , and [ o ^ ] are all synonymous. An equivalence class cannot be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in [: and : ] stands for the list of all characters belonging to that class. Standard character class names are: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit. These stand for the character classes defined in ctype. A locale can provide others. A character class cannot be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions [: < : ] and [: > : ] are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is an alnum character (as defined by ctype) or an underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. The constraint escapes described below are usually preferable (they are no more standard, but are certainly easier to type).

9.7.3.3. Regular Expression Escapes

Escapes are special sequences beginning with \ followed by an alphanumeric character. Escapes come in several varieties: character entry, class shorthands, constraint escapes, and back references. A \ followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a \ followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, \ is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes exist to make it easier to specify non-printing and otherwise inconvenient characters in REs. They are shown in Table 9-15.

Class-shorthand escapes provide shorthands for certain commonly-used character classes. They are shown in Table 9-16.
A *constraint escape* is a constraint, matching the empty string if specific conditions are met, written as an escape. They are shown in Table 9-17.

A *back reference* \( \text{\(n\)} \) matches the same string matched by the previous parenthesized subexpression specified by the number \( n \) (see Table 9-18). For example, \( ([bc]) \text{\(\backslash\)} \) matches \( bb \) or \( cc \) but not \( bc \) or \( cb \). The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions.

**Note:** Keep in mind that an escape's leading \( \backslash \) will need to be doubled when entering the pattern as an SQL string constant. For example:

\[
'123' \sim E'^{\backslash d(3)}' \text{ true}
\]

**Table 9-15. Regular Expression Character-Entry Escapes**

<table>
<thead>
<tr>
<th>Escape</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>alert (bell) character, as in C</td>
</tr>
<tr>
<td>\b</td>
<td>backspace, as in C</td>
</tr>
<tr>
<td>\B</td>
<td>synonym for \ to help reduce the need for backslash doubling</td>
</tr>
<tr>
<td>\cX</td>
<td>(where ( X ) is any character) the character whose low-order 5 bits are the same as those of ( X ), and whose other bits are all zero</td>
</tr>
<tr>
<td>\e</td>
<td>the character whose collating-sequence name is ( \text{ESC} ), or failing that, the character with octal value 033</td>
</tr>
<tr>
<td>\f</td>
<td>form feed, as in C</td>
</tr>
<tr>
<td>\n</td>
<td>newline, as in C</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return, as in C</td>
</tr>
<tr>
<td>\t</td>
<td>horizontal tab, as in C</td>
</tr>
<tr>
<td>\uwxxyz</td>
<td>(where ( wxyz ) is exactly four hexadecimal digits) the UTF16 (Unicode, 16-bit) character ( \text{U+}wxyz ) in the local byte ordering</td>
</tr>
<tr>
<td>\Ustuvwxyz</td>
<td>(where ( stuvwxyz ) is exactly eight hexadecimal digits) reserved for a somewhat-hypothetical Unicode extension to 32 bits</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab, as in C</td>
</tr>
<tr>
<td>\xhhhh</td>
<td>(where ( hh ) is any sequence of hexadecimal digits) the character whose hexadecimal value is ( 0xhhh ) (a single character no matter how many hexadecimal digits are used)</td>
</tr>
<tr>
<td>\0</td>
<td>the character whose value is 0</td>
</tr>
<tr>
<td>\xy</td>
<td>(where ( xy ) is exactly two octal digits, and is not a back reference) the character whose octal value is ( 0xy )</td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Escape</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\xyz</td>
<td>(where \xyz is exactly three octal digits, and is not a back reference) the character whose octal value is 0\xyz</td>
</tr>
</tbody>
</table>

Hexadecimal digits are 0-9, a-f, and A-F. Octal digits are 0-7.

The character-entry escapes are always taken as ordinary characters. For example, \135 is ] in ASCII, but \135 does not terminate a bracket expression.

Table 9-16. Regular Expression Class-Shorthand Escapes

<table>
<thead>
<tr>
<th>Escape</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>[:digit:]</td>
</tr>
<tr>
<td>\s</td>
<td>[:space:]</td>
</tr>
<tr>
<td>\w</td>
<td>[:alnum:]_ (note underscore is included)</td>
</tr>
<tr>
<td>\D</td>
<td>[^[:digit:]]</td>
</tr>
<tr>
<td>\S</td>
<td>[^[:space:]]</td>
</tr>
<tr>
<td>\W</td>
<td>[^[:alnum:]_] (note underscore is included)</td>
</tr>
</tbody>
</table>

Within bracket expressions, \d, \s, and \w lose their outer brackets, and \D, \S, and \W are illegal. (So, for example, \[a-c\d] is equivalent to \[a-c][:digit:]. Also, \[a-c\D], which is equivalent to \[a-c^[[:digit:]], is illegal.)

Table 9-17. Regular Expression Constraint Escapes

<table>
<thead>
<tr>
<th>Escape</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\A</td>
<td>matches only at the beginning of the string (see Section 9.7.3.5 for how this differs from ^)</td>
</tr>
<tr>
<td>\m</td>
<td>matches only at the beginning of a word</td>
</tr>
<tr>
<td>\M</td>
<td>matches only at the end of a word</td>
</tr>
<tr>
<td>\y</td>
<td>matches only at the beginning or end of a word</td>
</tr>
<tr>
<td>\Y</td>
<td>matches only at a point that is not the beginning or end of a word</td>
</tr>
<tr>
<td>\Z</td>
<td>matches only at the end of the string (see Section 9.7.3.5 for how this differs from $)</td>
</tr>
</tbody>
</table>

A word is defined as in the specification of [:<:] and [:>:] above. Constraint escapes are illegal within bracket expressions.

Table 9-18. Regular Expression Back References

<table>
<thead>
<tr>
<th>Escape</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\m</td>
<td>(where \m is a nonzero digit) a back reference to the \m'th subexpression</td>
</tr>
</tbody>
</table>
Escape | Description
--- | ---
\mnn | (where \(m\) is a nonzero digit, and \(nn\) is some more digits, and the decimal value \(mnn\) is not greater than the number of closing capturing parentheses seen so far) a back reference to the \(mnn\)th subexpression

**Note:** There is an inherent historical ambiguity between octal character-entry escapes and back references, which is resolved by heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multidigit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e. the number is in the legal range for a back reference), and otherwise is taken as octal.

### 9.7.3.4. Regular Expression Metasyntax

In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

Normally the flavor of RE being used is determined by `regex_flavor`. However, this can be overridden by a `director` prefix. If an RE begins with `***`, the rest of the RE is taken as an ARE regardless of `regex_flavor`. If an RE begins with `***=`, the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.

An ARE can begin with embedded options: a sequence `(xyz)` (where `xyz` is one or more alphabetic characters) specifies options affecting the rest of the RE. These options override any previously determined options (including both the RE flavor and case sensitivity). The available option letters are shown in Table 9-19.

**Table 9-19. ARE Embedded-Option Letters**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>rest of RE is a BRE</td>
</tr>
<tr>
<td>c</td>
<td>case-sensitive matching (overrides operator type)</td>
</tr>
<tr>
<td>e</td>
<td>rest of RE is an ERE</td>
</tr>
<tr>
<td>i</td>
<td>case-insensitive matching (see Section 9.7.3.5) (overrides operator type)</td>
</tr>
<tr>
<td>m</td>
<td>historical synonym for n</td>
</tr>
<tr>
<td>n</td>
<td>newline-sensitive matching (see Section 9.7.3.5)</td>
</tr>
<tr>
<td>p</td>
<td>partial newline-sensitive matching (see Section 9.7.3.5)</td>
</tr>
<tr>
<td>q</td>
<td>rest of RE is a literal (&quot;quoted&quot;) string, all ordinary characters</td>
</tr>
<tr>
<td>s</td>
<td>non-newline-sensitive matching (default)</td>
</tr>
<tr>
<td>t</td>
<td>tight syntax (default; see below)</td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\w</td>
<td>inverse partial newline-sensitive (&quot;weird&quot;) matching (see Section 9.7.3.5)</td>
</tr>
<tr>
<td>\x</td>
<td>expanded syntax (see below)</td>
</tr>
</tbody>
</table>

Embedded options take effect at the ) terminating the sequence. They can appear only at the start of an ARE (after the ***: director if any).

In addition to the usual (right) RE syntax, in which all characters are significant, there is an expanded syntax, available by specifying the embedded \x option. In the expanded syntax, white-space characters in the RE are ignored, as are all characters between a # and the following newline (or the end of the RE). This permits paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

- a white-space character or # preceded by \ is retained
- white space or # within a bracket expression is retained
- white space and comments cannot appear within multicharacter symbols, such as (?:

For this purpose, white-space characters are blank, tab, newline, and any character that belongs to the space character class.

Finally, in an ARE, outside bracket expressions, the sequence (?#ttt) (where ttt is any text not containing a ) is a comment, completely ignored. Again, this is not allowed between the characters of multicharacter symbols, like (?:. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if an initial ***= director has specified that the user’s input be treated as a literal string rather than as an RE.

9.7.3.5. Regular Expression Matching Rules

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, either the longest possible match or the shortest possible match will be taken, depending on whether the RE is greedy or non-greedy.

Whether an RE is greedy or not is determined by the following rules:

- Most atoms, and all constraints, have no greediness attribute (because they cannot match variable amounts of text anyway).
- Adding parentheses around an RE does not change its greediness.
- A quantified atom with a fixed-repetition quantifier (\{m\} or \{m\}? has the same greediness (possibly none) as the atom itself.
- A quantified atom with other normal quantifiers (including \{m, n\} with m equal to n) is greedy (prefers longest match).
- A quantified atom with a non-greedy quantifier (including \{m, n\}? with m equal to n) is non-greedy (prefers shortest match).
- A branch — that is, an RE that has no top-level | operator — has the same greediness as the first quantified atom in it that has a greediness attribute.
- An RE consisting of two or more branches connected by the | operator is always greedy.
The above rules associate greediness attributes not only with individual quantified atoms, but with branches and entire REs that contain quantified atoms. What that means is that the matching is done in such a way that the branch, or whole RE, matches the longest or shortest possible substring as a whole. Once the length of the entire match is determined, the part of it that matches any particular subexpression is determined on the basis of the greediness attribute of that subexpression, with subexpressions starting earlier in the RE taking priority over ones starting later.

An example of what this means:

```sql
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
```

```sql
SELECT SUBSTRING('XY1234Z', 'Y*?(0-9){1,3}');
Result: 1
```

In the first case, the RE as a whole is greedy because `Y*` is greedy. It can match beginning at the `Y`, and it matches the longest possible string starting there, i.e., `Y123`. The output is the parenthesized part of that, or `123`. In the second case, the RE as a whole is non-greedy because `Y*?` is non-greedy. It can match beginning at the `Y`, and it matches the shortest possible string starting there, i.e., `Y1`. The subexpression `[0-9]{1,3}` is greedy but it cannot change the decision as to the overall match length; so it is forced to match just `1`.

In short, when an RE contains both greedy and non-greedy subexpressions, the total match length is either as long as possible or as short as possible, according to the attribute assigned to the whole RE. The attributes assigned to the subexpressions only affect how much of that match they are allowed to "eat" relative to each other.

The quantifiers `{1,1}` and `{1,1}?` can be used to force greediness or non-greediness, respectively, on a subexpression or a whole RE.

Match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example: `bb*` matches the three middle characters of `abbbc`; `(week|wee)(night|knights)` matches all ten characters of `weeknights`; when `(.*).*` is matched against `abc` the parenthesized subexpression matches all three characters; and when `(a*)*` is matched against `bc` both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. `x` becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, e.g. `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

If newline-sensitive matching is specified, `^` and bracket expressions using `^` will never match the newline character (so that matches will never cross newlines unless the RE explicitly arranges it) and `^` and `$` will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. But the ARE escapes `\A` and `\Z` continue to match beginning or end of string only.

If partial newline-sensitive matching is specified, this affects `.` and bracket expressions as with newline-sensitive matching, but not `^` and `$`.

If inverse partial newline-sensitive matching is specified, this affects `^` and `.` as with newline-sensitive matching, but not `^` and `$`. bracket expressions. This isn’t very useful but is provided for symmetry.
9.7.3.6. Limits and Compatibility

No particular limit is imposed on the length of REs in this implementation. However, programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that \ does not lose its special significance inside bracket expressions. All other ARE features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the *** syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.

Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include \b, \B, the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-sensitive matching, the restrictions on parentheses and back references in lookahead constraints, and the longest/shortest-match (rather than first-match) matching semantics.

Two significant incompatibilities exist between AREs and the ERE syntax recognized by pre-7.4 releases of PostgreSQL:

- In AREs, \ followed by an alphanumeric character is either an escape or an error, while in previous releases, it was just another way of writing the alphanumeric. This should not be much of a problem because there was no reason to write such a sequence in earlier releases.
- In AREs, \ remains a special character within [ ], so a literal \ within a bracket expression must be written \ \ .

While these differences are unlikely to create a problem for most applications, you can avoid them if necessary by setting regex_flavor to extended.

9.7.3.7. Basic Regular Expressions

BREs differ from EREs in several respects. |, +, and ? are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are \{ and \}, with { and } by themselves ordinary characters. The parentheses for nested subexpressions are \ ( and \), with ( and ) by themselves ordinary characters. ^ is an ordinary character except at the beginning of the RE or the beginning of a parenthesized subexpression. $ is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and * is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading ^). Finally, single-digit back references are available, and \< and \> are synonyms for [[<:]] and [[>:]] respectively; no other escapes are available.

9.8. Data Type Formatting Functions

The PostgreSQL formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. Table 9-20 lists them. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.
The `to_timestamp` function can also take a single double precision argument to convert from Unix epoch to timestamp with time zone. (Integer Unix epochs are implicitly cast to double precision.)

### Table 9-20. Formatting Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_char(timestamp, text)</code></td>
<td>text</td>
<td>convert time stamp to string</td>
<td><code>to_char(current_timestamp, 'HH12:MI:SS')</code></td>
</tr>
<tr>
<td><code>to_char(interval, text)</code></td>
<td>text</td>
<td>convert interval to string</td>
<td><code>to_char(interval '15h 2m 12s', 'HH24:MI:SS')</code></td>
</tr>
<tr>
<td><code>to_char(int, text)</code></td>
<td>text</td>
<td>convert integer to string</td>
<td><code>to_char(125, '999')</code></td>
</tr>
<tr>
<td><code>to_char(double precision, text)</code></td>
<td>text</td>
<td>convert real/double precision to string</td>
<td><code>to_char(125.8::real, '999D9')</code></td>
</tr>
<tr>
<td><code>to_char(numeric, text)</code></td>
<td>text</td>
<td>convert numeric to string</td>
<td><code>to_char(-125.8, '999D999')</code></td>
</tr>
<tr>
<td><code>to_date(text, text)</code></td>
<td>date</td>
<td>convert string to date</td>
<td><code>to_date('05 Dec 2000', 'DD Mon YYYY')</code></td>
</tr>
<tr>
<td><code>to_number(text, text)</code></td>
<td>numeric</td>
<td>convert string to numeric</td>
<td><code>to_number('12,454.8-', '99G999D9S')</code></td>
</tr>
<tr>
<td><code>to_timestamp(text, text)</code></td>
<td>timestamp with time zone</td>
<td>convert string to time stamp</td>
<td><code>to_timestamp('05 Dec 2000', 'DD Mon YYYY')</code></td>
</tr>
<tr>
<td><code>to_timestamp(double precision)</code></td>
<td>timestamp with time zone</td>
<td>convert UNIX epoch to time stamp</td>
<td><code>to_timestamp(200120400)</code></td>
</tr>
</tbody>
</table>

In an output template string (for `to_char`), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for anything but `to_char`), template patterns identify the parts of the input data string to be looked at and the values to be found there.

Table 9-21 shows the template patterns available for formatting date and time values.

### Table 9-21. Template Patterns for Date/Time Formatting

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HH</td>
<td>hour of day (01-12)</td>
</tr>
<tr>
<td>HH12</td>
<td>hour of day (01-12)</td>
</tr>
<tr>
<td>HH24</td>
<td>hour of day (00-23)</td>
</tr>
<tr>
<td>MI</td>
<td>minute (00-59)</td>
</tr>
<tr>
<td>SS</td>
<td>second (00-59)</td>
</tr>
<tr>
<td>MS</td>
<td>millisecond (000-999)</td>
</tr>
<tr>
<td>US</td>
<td>microsecond (000000-999999)</td>
</tr>
<tr>
<td>SSSS</td>
<td>seconds past midnight (0-86399)</td>
</tr>
<tr>
<td>AM or A.M. or PM or P.M.</td>
<td>meridian indicator (uppercase)</td>
</tr>
<tr>
<td>am or a.m. or pm or p.m.</td>
<td>meridian indicator (lowercase)</td>
</tr>
<tr>
<td>Pattern</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Y,YYY</td>
<td>year (4 and more digits) with comma</td>
</tr>
<tr>
<td>YYYY</td>
<td>year (4 and more digits)</td>
</tr>
<tr>
<td>YYY</td>
<td>last 3 digits of year</td>
</tr>
<tr>
<td>YY</td>
<td>last 2 digits of year</td>
</tr>
<tr>
<td>Y</td>
<td>last digit of year</td>
</tr>
<tr>
<td>YYYY</td>
<td>ISO year (4 and more digits)</td>
</tr>
<tr>
<td>IYYYY</td>
<td>ISO year (4 and more digits)</td>
</tr>
<tr>
<td>IYY</td>
<td>last 3 digits of ISO year</td>
</tr>
<tr>
<td>IY</td>
<td>last 2 digits of ISO year</td>
</tr>
<tr>
<td>I</td>
<td>last digit of ISO year</td>
</tr>
<tr>
<td>BC or B.C. or AD or A.D.</td>
<td>era indicator (uppercase)</td>
</tr>
<tr>
<td>bc or b.c. or ad or a.d.</td>
<td>era indicator (lowercase)</td>
</tr>
<tr>
<td>MONTH</td>
<td>full uppercase month name (blank-padded to 9 chars)</td>
</tr>
<tr>
<td>Month</td>
<td>full mixed-case month name (blank-padded to 9 chars)</td>
</tr>
<tr>
<td>month</td>
<td>full lowercase month name (blank-padded to 9 chars)</td>
</tr>
<tr>
<td>MON</td>
<td>abbreviated uppercase month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>Mon</td>
<td>abbreviated mixed-case month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>mon</td>
<td>abbreviated lowercase month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>MM</td>
<td>month number (01-12)</td>
</tr>
<tr>
<td>DAY</td>
<td>full uppercase day name (blank-padded to 9 chars)</td>
</tr>
<tr>
<td>Day</td>
<td>full mixed-case day name (blank-padded to 9 chars)</td>
</tr>
<tr>
<td>day</td>
<td>full lowercase day name (blank-padded to 9 chars)</td>
</tr>
<tr>
<td>DY</td>
<td>abbreviated uppercase day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>Dy</td>
<td>abbreviated mixed-case day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>dy</td>
<td>abbreviated lowercase day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>DDD</td>
<td>day of year (001-366)</td>
</tr>
<tr>
<td>IDDDD</td>
<td>ISO day of year (001-371; day 1 of the year is Monday of the first ISO week.)</td>
</tr>
<tr>
<td>DD</td>
<td>day of month (01-31)</td>
</tr>
<tr>
<td>D</td>
<td>day of the week, Sunday(1) to Saturday(7)</td>
</tr>
<tr>
<td>ID</td>
<td>ISO day of the week, Monday(1) to Sunday(7)</td>
</tr>
<tr>
<td>W</td>
<td>week of month (1-5) (The first week starts on the first day of the month.)</td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WW</td>
<td>week number of year (1-53) (The first week starts on the first day of the year.)</td>
</tr>
<tr>
<td>IW</td>
<td>ISO week number of year (1 - 53; the first Thursday of the new year is in week 1.)</td>
</tr>
<tr>
<td>CC</td>
<td>century (2 digits) (The twenty-first century starts on 2001-01-01.)</td>
</tr>
<tr>
<td>J</td>
<td>Julian Day (days since November 24, 4714 BC at midnight)</td>
</tr>
<tr>
<td>Q</td>
<td>quarter</td>
</tr>
<tr>
<td>RM</td>
<td>month in Roman numerals (I-XII; I=January) (uppercase)</td>
</tr>
<tr>
<td>rm</td>
<td>month in Roman numerals (i-xii; i=January) (lowercase)</td>
</tr>
<tr>
<td>TZ</td>
<td>time-zone name (upercase)</td>
</tr>
<tr>
<td>tz</td>
<td>time-zone name (lowercase)</td>
</tr>
</tbody>
</table>

Certain modifiers can be applied to any template pattern to alter its behavior. For example, `FM` is the `Month` pattern with the `FM` modifier. Table 9-22 shows the modifier patterns for date/time formatting.

Table 9-22. Template Pattern Modifiers for Date/Time Formatting

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>FM prefix</td>
<td>fill mode (suppress padding blanks and zeroes)</td>
<td>FMMonth</td>
</tr>
<tr>
<td>TH suffix</td>
<td>uppercase ordinal number suffix</td>
<td>DDTH</td>
</tr>
<tr>
<td>th suffix</td>
<td>lowercase ordinal number suffix</td>
<td>DDth</td>
</tr>
<tr>
<td>FX prefix</td>
<td>fixed format global option (see usage notes)</td>
<td>FX Month DD Day</td>
</tr>
<tr>
<td>TM prefix</td>
<td>translation mode (print localized day and month names based on lc_messages)</td>
<td>TMMonth</td>
</tr>
<tr>
<td>SP suffix</td>
<td>spell mode (not yet implemented)</td>
<td>DDSP</td>
</tr>
</tbody>
</table>

Usage notes for date/time formatting:

- `FM` suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern be fixed-width.
- `TM` does not include trailing blanks.
- `to_timestamp` and `to_date` skip multiple blank spaces in the input string if the `FX` option is not used. `FX` must be specified as the first item in the template. For example `to_timestamp('2000 JUN', 'YYYY MON')` is correct, but `to_timestamp('2000 JUN', 'FXYYYY MON')` returns an error, because `to_timestamp`
expects one space only.

- Ordinary text is allowed in `to_char` templates and will be output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains pattern key words. For example, in "Hello Year YYYY", the YYYY will be replaced by the year data, but the single Y in Year will not be.

- If you want to have a double quote in the output you must precede it with a backslash, for example E'"YYYY Month"'. (Two backslashes are necessary because the backslash already has a special meaning when using the escape string syntax.)

- The YYYY conversion from string to `timestamp` or `date` has a restriction if you use a year with more than 4 digits. You must use some non-digit character or template after YYYY, otherwise the year is always interpreted as 4 digits. For example (with the year 20000):

```sql
to_date('200001131', 'YYYYMMDD') will be interpreted as a 4-digit year; instead use a non-digit separator after the year, like to_date('20000-1131', 'YYYY-MMDD') or to_date('20000Nov31', 'YYYYMonDD').
```

- In conversions from string to `timestamp` or `date`, the CC field is ignored if there is a Y, YYYY or Y,YYY field. If CC is used with YY or Y then the year is computed as (CC-1)*100+YY.

- An ISO week date (as distinct from a Gregorian date) can be specified to `to_timestamp` and `to_date` in one of two ways:

  - Year, week and weekday, for example `to_date('2006-42-4', 'IYYY-IW-ID')` returns the date 2006-10-19. If you omit the weekday it is assumed to be 1 (Monday).
  
  - Year and day of year, for example `to_date('2006-291', 'IYYY-IDDD')` also returns 2006-10-19.

Attempting to construct a date using a mixture of ISO week and Gregorian date fields is nonsensical, and could yield unexpected results. In the context of an ISO year, the concept of a 'month' or 'day of month' has no meaning. In the context of a Gregorian year, the ISO week has no meaning. Users should take care to keep Gregorian and ISO date specifications separate.

- Millisecond (MS) and microsecond (US) values in a conversion from string to `timestamp` are used as part of the seconds after the decimal point. For example `to_timestamp('12:3', 'SS:MS')` is not 3 milliseconds, but 300, because the conversion counts it as 12 + 0.3 seconds. This means for the format SS:MS, the input values 12:3, 12:30, and 12:300 specify the same number of milliseconds. To get three milliseconds, one must use 12:003, which the conversion counts as 12 + 0.003 = 12.003 seconds.

  Here is a more complex example: `to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US')` is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.

- `to_char(..., 'ID')`'s day of the week numbering matches the `extract(isodow from ...)` function, but `to_char(..., 'D')`'s does not match `extract(dow from ...)`'s day numbering.

- `to_char(interval)` formats HH and HH12 as hours in a single day, while HH24 can output hours exceeding a single day, e.g. >24.

Table 9-23 shows the template patterns available for formatting numeric values.

**Table 9-23. Template Patterns for Numeric Formatting**
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>value with the specified number of digits</td>
</tr>
<tr>
<td>0</td>
<td>value with leading zeros</td>
</tr>
<tr>
<td>. (period)</td>
<td>decimal point</td>
</tr>
<tr>
<td>, (comma)</td>
<td>group (thousand) separator</td>
</tr>
<tr>
<td>PR</td>
<td>negative value in angle brackets</td>
</tr>
<tr>
<td>S</td>
<td>sign anchored to number (uses locale)</td>
</tr>
<tr>
<td>L</td>
<td>currency symbol (uses locale)</td>
</tr>
<tr>
<td>D</td>
<td>decimal point (uses locale)</td>
</tr>
<tr>
<td>G</td>
<td>group separator (uses locale)</td>
</tr>
<tr>
<td>MI</td>
<td>minus sign in specified position (if number &lt; 0)</td>
</tr>
<tr>
<td>PL</td>
<td>plus sign in specified position (if number &gt; 0)</td>
</tr>
<tr>
<td>SG</td>
<td>plus/minus sign in specified position</td>
</tr>
<tr>
<td>RN</td>
<td>roman numeral (input between 1 and 3999)</td>
</tr>
<tr>
<td>TH or th</td>
<td>ordinal number suffix</td>
</tr>
<tr>
<td>V</td>
<td>shift specified number of digits (see notes)</td>
</tr>
<tr>
<td>EEEEE</td>
<td>scientific notation (not implemented yet)</td>
</tr>
</tbody>
</table>

Usage notes for numeric formatting:

- A sign formatted using `SG`, `PL`, or `MI` is not anchored to the number; for example, `to_char(-12, 'S9999')` produces `- 12'`, but `to_char(-12, 'MI9999')` produces `'- 12'`. The Oracle implementation does not allow the use of `MI` ahead of `9`, but rather requires that `9` precede `MI`.
- `9` results in a value with the same number of digits as there are `9`s. If a digit is not available it outputs a space.
- `TH` does not convert values less than zero and does not convert fractional numbers.
- `PL`, `SG`, and `TH` are PostgreSQL extensions.
- `V` effectively multiplies the input values by $10^n$, where $n$ is the number of digits following `V`. `to_char` does not support the use of `V` combined with a decimal point. (E.g., `99.9V99` is not allowed.)

Table 9-24 shows some examples of the use of the `to_char` function.

**Table 9-24. to_char Examples**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_char(current_timestamp, 'Day, DD HH24:MI:SS')</code></td>
<td>'Tuesday, 06 05:39:18'</td>
</tr>
<tr>
<td><code>to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')</code></td>
<td>'Tuesday, 6 05:39:18'</td>
</tr>
<tr>
<td><code>to_char(-0.1, '99.99')</code></td>
<td>'- .10'</td>
</tr>
<tr>
<td><code>to_char(-0.1, 'FM9.99')</code></td>
<td>'- .1'</td>
</tr>
<tr>
<td><code>to_char(0.1, '0.9')</code></td>
<td>' 0.1'</td>
</tr>
</tbody>
</table>
9.9. Date/Time Functions and Operators

Table 9-26 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 9-25 illustrates the behaviors of the basic arithmetic operators (+, *, etc.). For formatting functions, refer to Section 9.8. You should be familiar with the background information on date/time data types from Section 8.5.

All the functions and operators described below that take time or timestamp inputs actually come in two variants: one that takes time with time zone or timestamp with time zone, and one that takes time without time zone or timestamp without time zone. For brevity, these
variants are not shown separately. Also, the + and * operators come in commutative pairs (for example both date + integer and integer + date); we show only one of each such pair.

### Table 9-25. Date/Time Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>date '2001-09-28' + integer '7'</td>
<td>date '2001-10-05'</td>
</tr>
<tr>
<td>+</td>
<td>date '2001-09-28' + interval '1 hour'</td>
<td>timestamp '2001-09-28 01:00:00'</td>
</tr>
<tr>
<td>+</td>
<td>date '2001-09-28' + time '03:00'</td>
<td>timestamp '2001-09-28 03:00:00'</td>
</tr>
<tr>
<td>+</td>
<td>interval '1 day' + interval '1 hour'</td>
<td>interval '1 day 01:00:00'</td>
</tr>
<tr>
<td>+</td>
<td>timestamp '2001-09-28 01:00' + interval '23 hours'</td>
<td>timestamp '2001-09-29 00:00:00'</td>
</tr>
<tr>
<td>+</td>
<td>time '01:00' + interval '3 hours'</td>
<td>time '04:00:00'</td>
</tr>
<tr>
<td>-</td>
<td>- interval '23 hours'</td>
<td>interval '-23:00:00'</td>
</tr>
<tr>
<td>-</td>
<td>date '2001-10-01' - date '2001-09-28'</td>
<td>integer '3'</td>
</tr>
<tr>
<td>-</td>
<td>date '2001-10-01' - integer '7'</td>
<td>date '2001-09-24'</td>
</tr>
<tr>
<td>-</td>
<td>date '2001-09-28' - interval '1 hour'</td>
<td>timestamp '2001-09-27 23:00:00'</td>
</tr>
<tr>
<td>-</td>
<td>time '05:00' - time '03:00'</td>
<td>interval '02:00:00'</td>
</tr>
<tr>
<td>-</td>
<td>time '05:00' - interval '2 hours'</td>
<td>time '03:00:00'</td>
</tr>
<tr>
<td>-</td>
<td>timestamp '2001-09-28 23:00' - interval '23 hours'</td>
<td>timestamp '2001-09-28 00:00:00'</td>
</tr>
<tr>
<td>-</td>
<td>interval '1 day' - interval '1 hour'</td>
<td>interval '1 day -01:00:00'</td>
</tr>
<tr>
<td>-</td>
<td>timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'</td>
<td>interval '1 day 15:00:00'</td>
</tr>
<tr>
<td>*</td>
<td>900 * interval '1 second'</td>
<td>interval '00:15:00'</td>
</tr>
<tr>
<td>*</td>
<td>21 * interval '1 day'</td>
<td>interval '21 days'</td>
</tr>
<tr>
<td>*</td>
<td>double precision '3.5' * interval '1 hour'</td>
<td>interval '03:30:00'</td>
</tr>
<tr>
<td>/</td>
<td>interval '1 hour' / double precision '1.5'</td>
<td>interval '00:40:00'</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>age(timestamp, timestamp)</td>
<td>interval</td>
<td>Subtract arguments, producing a “symbolic” result that uses years and months</td>
</tr>
<tr>
<td>age(timestamp)</td>
<td>interval</td>
<td>Subtract from current_date</td>
</tr>
<tr>
<td>clock_timestamp</td>
<td>timestamp with time zone</td>
<td>Current date and time (changes during statement execution); see Section 9.9.4</td>
</tr>
<tr>
<td>current_date</td>
<td>date</td>
<td>Current date; see Section 9.9.4</td>
</tr>
<tr>
<td>current_time</td>
<td>time with time zone</td>
<td>Current time of day; see Section 9.9.4</td>
</tr>
<tr>
<td>current_timestamp</td>
<td>timestamp with time zone</td>
<td>Current date and time (start of current transaction); see Section 9.9.4</td>
</tr>
<tr>
<td>date_part(text, timestamp)</td>
<td>double precision</td>
<td>Get subfield (equivalent to extract); see Section 9.9.1</td>
</tr>
<tr>
<td>date_part(text, interval)</td>
<td>double precision</td>
<td>Get subfield (equivalent to extract); see Section 9.9.1</td>
</tr>
<tr>
<td>date_trunc(text, timestamp)</td>
<td>timestamp</td>
<td>Truncate to specified precision; see also Section 9.9.2</td>
</tr>
<tr>
<td>extract(field from timestamp)</td>
<td>double precision</td>
<td>Get subfield; see Section 9.9.1</td>
</tr>
<tr>
<td>extract(field from interval)</td>
<td>double precision</td>
<td>Get subfield; see Section 9.9.1</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isfinite(timesta...</td>
<td>boolean</td>
<td>Test for finite time stamp (not equal to infinity)</td>
</tr>
<tr>
<td>isfinite(interval</td>
<td>boolean</td>
<td>Test for finite interval</td>
</tr>
<tr>
<td>justify_days(interval)</td>
<td>interval</td>
<td>Adjust interval so 30-day time periods are represented as months</td>
</tr>
<tr>
<td>justify_hours(interval)</td>
<td>interval</td>
<td>Adjust interval so 24-hour time periods are represented as days</td>
</tr>
<tr>
<td>justify_interval(interval)</td>
<td>interval</td>
<td>Adjust interval using justify_days and justify_hours, with additional sign adjustments</td>
</tr>
<tr>
<td>localtime</td>
<td>time</td>
<td>Current time of day; see Section 9.9.4</td>
</tr>
<tr>
<td>localtimestamp</td>
<td>timestamp</td>
<td>Current date and time (start of current transaction); see Section 9.9.4</td>
</tr>
<tr>
<td>now()</td>
<td>timestamp</td>
<td>Current date and time (start of current transaction); see Section 9.9.4</td>
</tr>
<tr>
<td>statement_timestamp</td>
<td>timestamp</td>
<td>Current date and time (start of current statement); see Section 9.9.4</td>
</tr>
<tr>
<td>to_char()</td>
<td>text</td>
<td>Current date and time (like clock_timestamp, but as a text string); see Section 9.9.4</td>
</tr>
</tbody>
</table>
In addition to these functions, the SQL `OVERLAPS` operator is supported:

\[
(\text{start1}, \text{end1}) \ \text{OVERLAPS} \ (\text{start2}, \text{end2})
\]
\[
(\text{start1}, \text{length1}) \ \text{OVERLAPS} \ (\text{start2}, \text{length2})
\]

This expression yields true when two time periods (defined by their endpoints) overlap, false when they do not overlap. The endpoints can be specified as pairs of dates, times, or time stamps; or as a date, time, or time stamp followed by an interval.

\[
\text{SELECT} \ (\text{DATE} \ '2001-02-16', \ \text{DATE} \ '2001-12-21') \ \text{OVERLAPS} \ (\text{DATE} \ '2001-10-30', \ \text{DATE} \ '2002-10-30');
\]
\[\text{Result: true}\]

\[
\text{SELECT} \ (\text{DATE} \ '2001-02-16', \ \text{INTERVAL} \ '100\ days') \ \text{OVERLAPS} \ (\text{DATE} \ '2001-10-30', \ \text{DATE} \ '2002-10-30');
\]
\[\text{Result: false}\]

When adding an `interval` value to (or subtracting an `interval` value from) a `timestamp` with `time zone` value, the days component advances (or decrements) the date of the `timestamp` with `time zone` by the indicated number of days. Across daylight saving time changes (with the session time zone set to a time zone that recognizes DST), this means `interval '1\ day'` does not necessarily equal `interval '24\ hours'`. For example, with the session time zone set to `CST7CDT`, `timestamp` with `time zone '2005-04-02 12:00-07' + interval '1\ day'` will produce `timestamp` with `time zone '2005-04-03 12:00-06'`, while adding `interval '24\ hours'` to the same initial `timestamp` with `time zone produces `timestamp` with `time zone '2005-04-03 13:00-06'`, as there is a change in daylight saving time at 2005-04-03 02:00 in time zone CST7CDT.

Note there can be ambiguity in the months returned by `age` because different months have a different number of days. PostgreSQL's approach uses the month from the earlier of the two dates when calculating partial months. For example, `age('2004-06-01', '2004-04-30')` uses April to yield 1 mon 1 day, while using May would yield 1 mon 2 days because May has 31 days, while April has only 30.

### 9.9.1. EXTRACT, date_part

\[
\text{EXTRACT}(\text{field FROM source})
\]

The `extract` function retrieves subfields such as year or hour from date/time values. `source` must be a value expression of type `timestamp`, `time`, or `interval`. (Expressions of type `date` will be cast to `timestamp` and can therefore be used as well.) `field` is an identifier or string that selects what field to extract from the source value. The `extract` function returns values of type `double precision`. The following are valid field names:

- `century`
  - The century
Chapter 9. Functions and Operators

SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Result: 20

SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 21

The first century starts at 0001-01-01 00:00:00 AD, although they did not know it at the time. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 to 1. If you disagree with this, please write your complaint to: Pope, Cathedral Saint-Peter of Roma, Vatican.

PostgreSQL releases before 8.0 did not follow the conventional numbering of centuries, but just returned the year field divided by 100.

day

The day (of the month) field (1 - 31)

SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16

decade

The year field divided by 10

SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200

dow

The day of the week as Sunday(0) to Saturday(6)

SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5

Note that extract’s day of the week numbering is different from that of the to_char(..., ’D’) function.

doy

The day of the year (1 - 365/366)

SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47

epoch

For date and timestamp values, the number of seconds since 1970-01-01 00:00:00-00 (can be negative); for interval values, the total number of seconds in the interval

SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-08');
Result: 982384720

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800

Here is how you can convert an epoch value back to a time stamp:

SELECT TIMESTAM WITH TIME ZONE 'epoch' + 982384720 * INTERVAL '1 second';

hour

The hour field (0 - 23)

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
Chapter 9. Functions and Operators

isodow

The day of the week as Monday(1) to Sunday(7)

```sql
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Result: 7
```

This is identical to dow except for Sunday. This matches the ISO 8601 day of the week numbering.

isoyear

The ISO 8601 year that the date falls in (not applicable to intervals).

```sql
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Result: 2005
```

```sql
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Result: 2006
```

Each ISO year begins with the Monday of the week containing the 4th of January, so in early January or late December the ISO year may be different from the Gregorian year. See the week field for more information.

This field is not available in PostgreSQL releases prior to 8.3.

microseconds

The seconds field, including fractional parts, multiplied by 1 000 000. Note that this includes full seconds.

```sql
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

millennium

The millennium

```sql
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 3
```

Years in the 1900s are in the second millennium. The third millennium starts January 1, 2001.

PostgreSQL releases before 8.0 did not follow the conventional numbering of millennia, but just returned the year field divided by 1000.

milliseconds

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```sql
SELECT EXTRACT(MILLISECOND FROM TIME '17:12:28.5');
Result: 28500
```

minute

The minutes field (0 - 59)

```sql
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

month

For timestamp values, the number of the month within the year (1 - 12); for interval values the number of months, modulo 12 (0 - 11)

```sql
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2
```

```sql
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
```
Chapter 9. Functions and Operators

Result: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1

quarter

The quarter of the year (1 - 4) that the day is in

SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1

second

The seconds field, including fractional parts (0 - 59)

SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5

timezone

The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC.

timezone_hour

The hour component of the time zone offset

timezone_minute

The minute component of the time zone offset

week

The number of the week of the year that the day is in. By definition (ISO 8601), the first week of a year contains January 4 of that year. (The ISO-8601 week starts on Monday.) In other words, the first Thursday of a year is in week 1 of that year.

Because of this, it is possible for early January dates to be part of the 52nd or 53rd week of the previous year. For example, 2005-01-01 is part of the 53rd week of year 2004, and 2006-01-01 is part of the 52nd week of year 2005.

SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7

year

The year field. Keep in mind there is no 0 AD, so subtracting BC years from AD years should be done with care.

SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001

The extract function is primarily intended for computational processing. For formatting date/time values for display, see Section 9.8.

The date_part function is modeled on the traditional Ingres equivalent to the SQL-standard function extract:

date_part('field', source)

60 if leap seconds are implemented by the operating system
Note that here the field parameter needs to be a string value, not a name. The valid field names for date_part are the same as for extract.

SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4

9.9.2. date_trunc

The function date_trunc is conceptually similar to the trunc function for numbers.

date_trunc('field', source)

source is a value expression of type timestamp or interval. (Values of type date and time are cast automatically, to timestamp or interval respectively.) field selects to which precision to truncate the input value. The return value is of type timestamp or interval with all fields that are less significant than the selected one set to zero (or one, for day and month).

Valid values for field are:

milliseconds
seconds
minutes
hours
days
weeks
months
quarters
years
decades
centuries
millennia

Examples:

SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00

9.9.3. AT TIME ZONE

The AT TIME ZONE construct allows conversions of time stamps to different time zones. Table 9-27 shows its variants.
Table 9-27. AT TIME ZONE Variants

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp without time zone AT TIME ZONE zone</td>
<td>timestamp with time zone</td>
<td>Treat given time stamp without time zone as located in the specified time zone</td>
</tr>
<tr>
<td>timestamp with time zone AT TIME ZONE zone</td>
<td>timestamp without time zone zone</td>
<td>Convert given time stamp with time zone to the new time zone</td>
</tr>
<tr>
<td>time with time zone AT TIME ZONE zone</td>
<td>time with time zone</td>
<td>Convert given time with time zone to the new time zone</td>
</tr>
</tbody>
</table>

In these expressions, the desired time zone zone can be specified either as a text string (e.g., 'PST') or as an interval (e.g., INTERVAL '-08:00'). In the text case, a time zone name can be specified in any of the ways described in Section 8.5.3.

Examples (supposing that the local time zone is PST8PDT):

SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Result: 2001-02-16 18:38:40

The first example takes a time stamp without time zone and interprets it as MST time (UTC-7), which is then converted to PST (UTC-8) for display. The second example takes a time stamp specified in EST (UTC-5) and converts it to local time in MST (UTC-7).

The function timezone(zone, timestamp) is equivalent to the SQL-conforming construct timestamp AT TIME ZONE zone.

9.9.4. Current Date/Time

PostgreSQL provides a number of functions that return values related to the current date and time. These SQL-standard functions all return values based on the start time of the current transaction:

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)

CURRENT_TIME and CURRENT_TIMESTAMP deliver values with time zone; LOCALTIME and LOCALTIMESTAMP deliver values without time zone.

CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, and LOCALTIMESTAMP can optionally be given a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

Some examples:
SELECT CURRENT_TIME;
Result: 14:39:53.662522-05

SELECT CURRENT_DATE;
Result: 2001-12-23

SELECT CURRENT_TIMESTAMP;
Result: 2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP (2);
Result: 2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Result: 2001-12-23 14:39:53.662522

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications within the same transaction bear the same timestamp.

Note: Other database systems might advance these values more frequently.

PostgreSQL also provides functions that return the start time of the current statement, as well as the actual current time at the instant the function is called. The complete list of non-SQL-standard time functions is:

now()
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()

now() is a traditional PostgreSQL equivalent to CURRENT_TIMESTAMP. transaction_timestamp() is likewise equivalent to CURRENT_TIMESTAMP, but is named to clearly reflect what it returns. statement_timestamp() returns the start time of the current statement (more specifically, the time of receipt of the latest command message from the client). statement_timestamp() and transaction_timestamp() return the same value during the first command of a transaction, but might differ during subsequent commands. clock_timestamp() returns the actual current time, and therefore its value changes even within a single SQL command. timeofday() is a historical PostgreSQL function. Like clock_timestamp(), it returns the actual current time, but as a formatted text string rather than a timestamp with time zone value.

All the date/time data types also accept the special literal value now to specify the current date and time (again, interpreted as the transaction start time). Thus, the following three all return the same result:

SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now'; -- incorrect for use with DEFAULT
Tip: You do not want to use the third form when specifying a `DEFAULT` clause while creating a table. The system will convert now to a timestamp as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion.

### 9.9.5. Delaying Execution

The following function is available to delay execution of the server process:

```sql
pg_sleep(seconds)
```

`pg_sleep` makes the current session’s process sleep until `seconds` seconds have elapsed. `seconds` is a value of type `double precision`, so fractional-second delays can be specified. For example:

```sql
SELECT pg_sleep(1.5);
```

**Note:** The effective resolution of the sleep interval is platform-specific; 0.01 seconds is a common value. The sleep delay will be at least as long as specified. It might be longer depending on factors such as server load.

### Warning

Make sure that your session does not hold more locks than necessary when calling `pg_sleep`. Otherwise other sessions might have to wait for your sleeping process, slowing down the entire system.

### 9.10. Enum Support Functions

For enum types (described in Section 8.7), there are several functions that allow cleaner programming without hard-coding particular values of an enum type. These are listed in Table 9-28. The examples assume an enum type created as:

```sql
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Example Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enum_first(ayenum)</code></td>
<td>Returns the first value of the input enum type</td>
<td><code>enum_first(null::rainbow)</code></td>
<td></td>
</tr>
<tr>
<td><code>enum_last(ayenum)</code></td>
<td>Returns the last value of the input enum type</td>
<td><code>enum_last(null::rainbow)</code></td>
<td><code>purple</code></td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Example Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enum_range(anyenum)</code></td>
<td>Returns all values of the input enum type in an ordered array</td>
<td><code>enum_range(null::rainbow)</code></td>
<td><code>{red,orange,yellow,green,blue,purple}</code></td>
</tr>
<tr>
<td><code>enum_range(anyenum, anyenum)</code></td>
<td>Returns the range between the two given enum values, as an ordered array. The values must be from the same enum type. If the first parameter is null, the result will start with the first value of the enum type. If the second parameter is null, the result will end with the last value of the enum type.</td>
<td><code>enum_range('orange'::rainbow, 'green'::rainbow)</code></td>
<td><code>{orange,yellow,green}</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>enum_range(NULL, 'green'::rainbow)</code></td>
<td><code>{red,orange,yellow,green}</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>enum_range('orange'::rainbow, NULL)</code></td>
<td><code>{orange,yellow,green,blue,purple}</code></td>
</tr>
</tbody>
</table>

Notice that except for the two-argument form of `enum_range`, these functions disregard the specific value passed to them; they care only about its declared data type. Either null or a specific value of the type can be passed, with the same result. It is more common to apply these functions to a table column or function argument than to a hardwired type name as suggested by the examples.

9.11. Geometric Functions and Operators

The geometric types `point`, `box`, `lseg`, `line`, `path`, `polygon`, and `circle` have a large set of native support functions and operators, shown in Table 9-29, Table 9-30, and Table 9-31.

### Caution

Note that the “same as” operator, `~=` represents the usual notion of equality for the `point`, `box`, `polygon`, and `circle` types. Some of these types also have an `=` operator, but `=` compares for equal areas only. The other scalar comparison operators (<, <= and so on) likewise compare areas for these types.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Translation</td>
<td>box '((0,0),(1,1))' + point '(2.0,0)'</td>
</tr>
<tr>
<td>-</td>
<td>Translation</td>
<td>box '((0,0),(1,1))' - point '(2.0,0)'</td>
</tr>
<tr>
<td>*</td>
<td>Scaling/rotation</td>
<td>box '((0,0),(1,1))' * point '(2.0,0)'</td>
</tr>
<tr>
<td>/</td>
<td>Scaling/rotation</td>
<td>box '((0,0),(2,2))' / point '(2.0,0)'</td>
</tr>
</tbody>
</table>
### Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#</code></td>
<td>Point or box of intersection</td>
<td><code>'((1,-1),(-1,1))'</code> <code>#</code> <code>'((1,1),(-1,-1))'</code></td>
</tr>
<tr>
<td><code>#</code></td>
<td>Number of points in path or polygon</td>
<td><code>#</code> <code>'(1,0),(0,1),(-1,0))'</code></td>
</tr>
<tr>
<td><code>@-@</code></td>
<td>Length or circumference</td>
<td><code>@-@</code> path <code>'(0,0),(1,0))'</code></td>
</tr>
<tr>
<td><code>@@</code></td>
<td>Center</td>
<td><code>@@</code> circle <code>'((0,0),10)'</code></td>
</tr>
<tr>
<td><code>##</code></td>
<td>Closest point to first operand on second operand</td>
<td><code>##</code> point <code>'((0,0)' </code>##<code>lseg</code>'((2,0),(0,2))'`</td>
</tr>
<tr>
<td><code>&lt;-&gt;</code></td>
<td>Distance between</td>
<td><code>&lt;-&gt;</code> circle <code>'((0,0),1)'</code> <code>&lt;-&gt;</code> circle <code>'((5,0),1)'</code></td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>Overlaps?</td>
<td><code>&amp;&amp;</code> box <code>'((0,0),(1,1))'</code> <code>&amp;&amp;</code> box <code>'((0,0),(2,2))'</code></td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Is strictly left of?</td>
<td><code>&lt;</code> circle <code>'((0,0),1)'</code> <code>&lt;</code> circle <code>'((5,0),1)'</code></td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Is strictly right of?</td>
<td><code>&gt;&gt;</code> circle <code>'((5,0),1)'</code> <code>&gt;&gt;</code> circle <code>'((0,0),1)'</code></td>
</tr>
<tr>
<td><code>&amp;&lt;</code></td>
<td>Does not extend to the right of?</td>
<td><code>&amp;&lt;</code> box <code>'((0,0),(1,1))'</code> <code>&amp;&lt;</code> box <code>'((0,0),(2,2))'</code></td>
</tr>
<tr>
<td><code>&amp;&gt;</code></td>
<td>Does not extend to the left of?</td>
<td><code>&amp;&gt;</code> box <code>'((0,0),(3,3))'</code> <code>&amp;&gt;</code> box <code>'((0,0),(2,2))'</code></td>
</tr>
<tr>
<td>`&lt;&lt;</td>
<td>`</td>
<td>Is strictly below?</td>
</tr>
<tr>
<td>`</td>
<td>&gt;&gt;`</td>
<td>Is strictly above?</td>
</tr>
<tr>
<td>`&amp;&lt;</td>
<td>`</td>
<td>Does not extend above?</td>
</tr>
<tr>
<td>`</td>
<td>&amp;&gt;`</td>
<td>Does not extend below?</td>
</tr>
<tr>
<td><code>&lt;^</code></td>
<td>Is below (allows touching)?</td>
<td><code>&lt;^</code> circle <code>'((0,0),1)'</code> <code>&lt;^</code> circle <code>'((0,0),1)'</code></td>
</tr>
<tr>
<td><code>&gt;^</code></td>
<td>Is above (allows touching)?</td>
<td><code>&gt;^</code> circle <code>'((0,0),1)'</code> <code>&gt;^</code> circle <code>'((0,0),1)'</code></td>
</tr>
<tr>
<td><code>?#</code></td>
<td>Intersects?</td>
<td><code>?#</code> lseg <code>'((-1,0),(1,0))'</code> <code>?#</code> box <code>'((-2,-2),(2,2))'</code></td>
</tr>
<tr>
<td><code>?-</code></td>
<td>Is horizontal?</td>
<td><code>?-</code> lseg <code>'((-1,0),(1,0))'</code></td>
</tr>
<tr>
<td><code>?-</code></td>
<td>Are horizontally aligned?</td>
<td><code>?-</code> point <code>'((1,0)' </code>?-<code>point</code>'((0,0)'`</td>
</tr>
<tr>
<td>`?</td>
<td>`</td>
<td>Is vertical?</td>
</tr>
<tr>
<td>`?</td>
<td>`</td>
<td>Are vertically aligned?</td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>?-</td>
<td></td>
<td>Is perpendicular?</td>
</tr>
<tr>
<td>lseg '((0,0),(0,1))'</td>
<td></td>
<td>lseg '((-1,0),(1,0))'</td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@&gt;</td>
<td></td>
<td>Contains?</td>
</tr>
<tr>
<td>circle '((0,0),2)' @&gt; point '(1,1)'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;@</td>
<td></td>
<td>Contained in or on?</td>
</tr>
<tr>
<td>point '(1,1)' &lt;@ circle '((0,0),2)'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>~=</td>
<td></td>
<td>Same as?</td>
</tr>
<tr>
<td>polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Before PostgreSQL 8.2, the containment operators @> and <@ were respectively called ~ and @. These names are still available, but are deprecated and will eventually be retired.

### Table 9-30. Geometric Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>area(object)</td>
<td>double precision</td>
<td>area</td>
<td>area(box '((0,0),(1,1))')</td>
</tr>
<tr>
<td>center(object)</td>
<td>point</td>
<td>center</td>
<td>center(box '((0,0),(1,2))')</td>
</tr>
<tr>
<td>diameter(circle)</td>
<td>double precision</td>
<td>diameter of circle</td>
<td>diameter(circle '((0,0),2.0)')</td>
</tr>
<tr>
<td>height(box)</td>
<td>double precision</td>
<td>vertical size of box</td>
<td>height(box '((0,0),(1,1))')</td>
</tr>
<tr>
<td>isclosed(path)</td>
<td>boolean</td>
<td>a closed path?</td>
<td>isclosed(path '((0,0),(1,1),(2,0))')</td>
</tr>
<tr>
<td>isopen(path)</td>
<td>boolean</td>
<td>an open path?</td>
<td>isopen(path '[(0,0),(1,1),(2,0)]')</td>
</tr>
<tr>
<td>length(object)</td>
<td>double precision</td>
<td>length</td>
<td>length(path '((-1,0),(1,0))')</td>
</tr>
<tr>
<td>npoints(path)</td>
<td>int</td>
<td>number of points</td>
<td>npoints(path '[(0,0),(1,1),(2,0)]')</td>
</tr>
<tr>
<td>npoints(polygon)</td>
<td>int</td>
<td>number of points</td>
<td>npoints(polygon '((1,1),(0,0))')</td>
</tr>
<tr>
<td>pclose(path)</td>
<td>path</td>
<td>convert path to closed</td>
<td>pclose(path '[(0,0),(1,1),(2,0)]')</td>
</tr>
</tbody>
</table>
### Table 9-31. Geometric Type Conversion Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>box(circle)</td>
<td>box</td>
<td>circle to box</td>
<td>box(circle ’((0,0),2.0)’)</td>
</tr>
<tr>
<td>box(point, point)</td>
<td>box</td>
<td>points to box</td>
<td>box(point ’(0,0)’, point ’(1,1)’)</td>
</tr>
<tr>
<td>box(polygon)</td>
<td>box</td>
<td>polygon to box</td>
<td>box(polygon ’((0,0),(1,1),(2,0))’)</td>
</tr>
<tr>
<td>circle(box)</td>
<td>circle</td>
<td>box to circle</td>
<td>circle(box ’((0,0),(1,1))’)</td>
</tr>
<tr>
<td>circle(point, double precision)</td>
<td>circle</td>
<td>center and radius to circle</td>
<td>circle(point ’(0,0)’, 2.0)</td>
</tr>
<tr>
<td>circle(polygon)</td>
<td>circle</td>
<td>polygon to circle</td>
<td>circle(polygon ’((0,0),(1,1),(2,0))’)</td>
</tr>
<tr>
<td>lseg(box)</td>
<td>lseg</td>
<td>box diagonal to line segment</td>
<td>lseg(box ’((-1,0),(1,0))’)</td>
</tr>
<tr>
<td>lseg(point, point)</td>
<td>lseg</td>
<td>points to line segment</td>
<td>lseg(point ’(-1,0)’, point ’(1,0)’)</td>
</tr>
<tr>
<td>path(polygon)</td>
<td>path</td>
<td>polygon to path</td>
<td>path(polygon ’((0,0),(1,1),(2,0))’)</td>
</tr>
<tr>
<td>point(double precision, double precision)</td>
<td>point</td>
<td>construct point</td>
<td>point(23.4, -44.5)</td>
</tr>
<tr>
<td>point(box)</td>
<td>point</td>
<td>center of box</td>
<td>point(box ’((-1,0),(1,0))’)</td>
</tr>
<tr>
<td>point(circle)</td>
<td>point</td>
<td>center of circle</td>
<td>point(circle ’((0,0),(2,0))’)</td>
</tr>
<tr>
<td>point(lseg)</td>
<td>point</td>
<td>center of line segment</td>
<td>point(lseg ’((-1,0),(1,0))’)</td>
</tr>
<tr>
<td>point(polygon)</td>
<td>point</td>
<td>center of polygon</td>
<td>point(polygon ’((0,0),(1,1),(2,0))’)</td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>polygon(box)</td>
<td>polygon</td>
<td>box to 4-point polygon</td>
<td>polygon(box '((0,0),(1,1))')</td>
</tr>
<tr>
<td>polygon(circle)</td>
<td>polygon</td>
<td>circle to 12-point polygon</td>
<td>polygon(circle '((0,0),2.0)')</td>
</tr>
<tr>
<td>polygon(npts,</td>
<td>polygon</td>
<td>circle to npts-point</td>
<td>polygon(12, circle '((0,0),2.0)')</td>
</tr>
<tr>
<td>circle)</td>
<td></td>
<td>polygon</td>
<td></td>
</tr>
<tr>
<td>polygon(path)</td>
<td>polygon</td>
<td>path to polygon</td>
<td>polygon(path '((0,0),(1,1),(2,2))')</td>
</tr>
</tbody>
</table>

It is possible to access the two component numbers of a point as though it were an array with indices 0 and 1. For example, if \( t.p \) is a point column then \( \text{SELECT } p[0] \text{ FROM } t \) retrieves the X coordinate and \( \text{UPDATE } t \text{ SET } p[1] = ... \) changes the Y coordinate. In the same way, a value of type box or lseg can be treated as an array of two point values.

The area function works for the types box, circle, and path. The area function only works on the path data type if the points in the path are non-intersecting. For example, the path \(((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))\)::PATH won’t work, however, the following visually identical path \(((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))\)::PATH will work. If the concept of an intersecting versus non-intersecting path is confusing, draw both of the above paths side by side on a piece of graph paper.

9.12. Network Address Functions and Operators

Table 9-32 shows the operators available for the cidr and inet types. The operators <<, <<<, >>, and >>> test for subnet inclusion. They consider only the network parts of the two addresses, ignoring any host part, and determine whether one network part is identical to or a subnet of the other.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>is less than</td>
<td>inet ‘192.168.1.5’ &lt; inet ‘192.168.1.6’</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal</td>
<td>inet ‘192.168.1.5’ &lt;= inet ‘192.168.1.5’</td>
</tr>
<tr>
<td>=</td>
<td>equals</td>
<td>inet ‘192.168.1.5’ = inet ‘192.168.1.5’</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater or equal</td>
<td>inet ‘192.168.1.5’ &gt;= inet ‘192.168.1.5’</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
<td>inet ‘192.168.1.5’ &gt; inet ‘192.168.1.5’</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>is not equal</td>
<td>inet ‘192.168.1.5’ &lt;&gt; inet ‘192.168.1.4’</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>is contained within</td>
<td>inet ‘192.168.1.5’ &lt;&lt; inet ‘192.168.1/24’</td>
</tr>
</tbody>
</table>
Table 9-33 shows the functions available for use with the cidr and inet types. The host, text, and abbrev functions are primarily intended to offer alternative display formats.

### Table 9-33. cidr and inet Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbrev(inet)</td>
<td>text</td>
<td>abbreviated display format as text</td>
<td>abbrev(inet '10.1.0.0/16')</td>
<td>10.1.0.0/16</td>
</tr>
<tr>
<td>abbrev(cidr)</td>
<td>text</td>
<td>abbreviated display format as text</td>
<td>abbrev(cidr '10.1.0.0/16')</td>
<td>10.1/16</td>
</tr>
<tr>
<td>broadcast(inet)</td>
<td>inet</td>
<td>broadcast address for network</td>
<td>broadcast('192.168.1.5/24')</td>
<td>192.168/24</td>
</tr>
<tr>
<td>family(inet)</td>
<td>int</td>
<td>extract family of address; 4 for IPv4, 6 for IPv6</td>
<td>family('::1')</td>
<td>6</td>
</tr>
<tr>
<td>host(inet)</td>
<td>text</td>
<td>extract IP address as text</td>
<td>host('192.168.1.5/24')</td>
<td>192.168.1.5</td>
</tr>
<tr>
<td>hostmask(inet)</td>
<td>inet</td>
<td>construct host mask for network</td>
<td>hostmask('192.168.128/30')</td>
<td>192.168.128/30</td>
</tr>
<tr>
<td>masklen(inet)</td>
<td>int</td>
<td>extract netmask length</td>
<td>masklen('192.168.128/16')</td>
<td>24</td>
</tr>
<tr>
<td>netmask(inet)</td>
<td>inet</td>
<td>construct netmask for network</td>
<td>netmask('192.168.128/16')</td>
<td>192.168.128/24</td>
</tr>
<tr>
<td>network(inet)</td>
<td>cidr</td>
<td>extract network part of address</td>
<td>network('192.168.128/16')</td>
<td>192.168.128/16</td>
</tr>
<tr>
<td>set_masklen(inet)</td>
<td>inet</td>
<td>set netmask length for inet value</td>
<td>set_masklen('192.168.128/16')</td>
<td>24</td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

Table 9-34. Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_masklen</td>
<td>cidr</td>
<td>set netmask length for cidr value</td>
<td>set_masklen('129.26.68.0/16'::cidr, 16)</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>text</td>
<td>extract IP address and netmask</td>
<td>text(inet '192.168.1.5')</td>
<td>192.168.1.5/32</td>
</tr>
</tbody>
</table>

Any cidr value can be cast to inet implicitly or explicitly; therefore, the functions shown above as operating on inet also work on cidr values. (Where there are separate functions for inet and cidr, it is because the behavior should be different for the two cases.) Also, it is permitted to cast an inet value to cidr. When this is done, any bits to the right of the netmask are silently zeroed to create a valid cidr value. In addition, you can cast a text value to inet or cidr using normal casting syntax: for example, inet(expression) or colname::cidr.

Table 9-34 shows the functions available for use with the macaddr type. The function trunc(macaddr) returns a MAC address with the last 3 bytes set to zero. This can be used to associate the remaining prefix with a manufacturer.

Table 9-34. macaddr Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>trunc(macaddr)</td>
<td>macaddr</td>
<td>set last 3 bytes to zero</td>
<td>trunc(macaddr '12:34:56:78:90:ab')</td>
<td>12:34:56:00:00:00</td>
</tr>
</tbody>
</table>

The macaddr type also supports the standard relational operators (>, <, etc.) for lexicographical ordering.

9.13. Text Search Functions and Operators

Table 9-35, Table 9-36 and Table 9-37 summarize the functions and operators that are provided for full text searching. See Chapter 12 for a detailed explanation of PostgreSQL’s text search facility.

Table 9-35. Text Search Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@</td>
<td>tsvector matches tsquery ?</td>
<td>to_tsvector('fat cats ate rats') @@ to_tsquery('cat &amp; rat')</td>
<td>t</td>
</tr>
<tr>
<td>@@@</td>
<td>same as @@, but see Section 12.9</td>
<td>to_tsvector('fat cats ate rats') @@@ to_tsquery('cat &amp; rat')</td>
<td>t</td>
</tr>
</tbody>
</table>
### Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>concatenate tsvector</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND tsquery together</td>
<td>'fat</td>
<td>rat':tsquery &amp;&amp; 'cat':tsquery</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR tsquery together</td>
</tr>
<tr>
<td>!!!</td>
<td>negate tsquery</td>
<td>!! 'cat':tsquery</td>
<td>'cat'</td>
</tr>
<tr>
<td>@&gt;</td>
<td>tsquery contains another ?</td>
<td>'cat':tsquery @&gt; f</td>
<td>'cat &amp; rat':tsquery</td>
</tr>
<tr>
<td>@&lt;</td>
<td>tsquery is contained in ?</td>
<td>'cat':tsquery @&lt;</td>
<td>'cat &amp; rat':tsquery</td>
</tr>
</tbody>
</table>

**Note:** The tsquery containment operators consider only the lexemes listed in the two queries, ignoring the combining operators.

In addition to the operators shown in the table, the ordinary B-tree comparison operators (=, <, etc) are defined for types tsvector and tsquery. These are not very useful for text searching but allow, for example, unique indexes to be built on columns of these types.

### Table 9-36. Text Search Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_tsvector([ config regconfig, ] document_text)</td>
<td>tsvector</td>
<td>reduce document text to tsvector</td>
<td>to_tsvector('english', 'The Fat Rats')</td>
<td>'fat':2 'rat':3</td>
</tr>
<tr>
<td>length(tsvector)</td>
<td>integer</td>
<td>number of lexemes in tsvector</td>
<td>length('fat:2,4 cat:3 rat:5A':tsvector)</td>
<td></td>
</tr>
<tr>
<td>setweight(tsvector, &quot;char&quot;)</td>
<td>tsvector</td>
<td>assign weight to each element of tsvector</td>
<td>setweight('fat:2,4A cat:3 rat:5B':tsvector, 'A')</td>
<td></td>
</tr>
<tr>
<td>strip(tsvector)</td>
<td>tsvector</td>
<td>remove positions and weights from tsvector</td>
<td>strip('fat:2,4A cat:3 rat:5B':tsvector)</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
<td>--------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td><code>to_tsquery</code></td>
<td><code>tsquery</code></td>
<td>normalize words and convert to tsquery</td>
<td><code>to_tsquery('english', 'The &amp; Fat &amp; Rats')</code></td>
<td>'fat' &amp; 'rat'</td>
</tr>
<tr>
<td><code>plainto_tsquery</code></td>
<td><code>tsquery</code></td>
<td>produce tsquery ignoring punctuation</td>
<td><code>plainto_tsquery('english', 'The Fat Rats')</code></td>
<td>'fatenglish' &amp; 'rat'</td>
</tr>
<tr>
<td><code>numnode(tsquery)</code></td>
<td><code>integer</code></td>
<td>number of lexemes plus operators in tsquery</td>
<td>`numnode('fat &amp; rat</td>
<td>cat'::tsquery)`</td>
</tr>
<tr>
<td><code>querytree(query tsquery)</code></td>
<td><code>text</code></td>
<td>get indexable part of a tsquery</td>
<td><code>querytree('foo &amp; ! bar'::tsquery)</code></td>
<td>'foo'</td>
</tr>
<tr>
<td><code>ts_rank</code></td>
<td><code>float4</code></td>
<td>rank document for query</td>
<td><code>ts_rank(textsearch)</code></td>
<td>0.818</td>
</tr>
<tr>
<td><code>ts_rank_cd</code></td>
<td><code>float4</code></td>
<td>rank document for query using cover density</td>
<td><code>ts_rank_cd('0.2, 0.4, 1.0', textsearch, query)</code></td>
<td>2.01317</td>
</tr>
<tr>
<td><code>ts_headline</code></td>
<td><code>text</code></td>
<td>display a query match</td>
<td><code>ts_headline('x &lt;b&gt;y &lt;b&gt;z&lt;/b&gt;', 'z'::tsquery)</code></td>
<td>x y &amp; z</td>
</tr>
<tr>
<td><code>ts_rewrite</code></td>
<td><code>tsquery</code></td>
<td>replace target with substitute within query</td>
<td>`ts_rewrite('a &amp; b':tsquery, 'a':tsquery, 'foo</td>
<td>bar':tsquery)`</td>
</tr>
</tbody>
</table>
### Table 9-37. Text Search Debugging Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts_rewrite(query tsquery, select text)</td>
<td>tsquery</td>
<td>replace using targets and substitutes from a SELECT command</td>
<td><code>SELECT ts_rewrite('a &amp; b'::tsquery, 'SELECT t,s FROM aliases')</code></td>
<td>'b' &amp; ( 'foo'</td>
</tr>
<tr>
<td>get_current_ts_config()</td>
<td>regconfig</td>
<td>get default text search configuration</td>
<td><code>get_current_ts_config()</code></td>
<td></td>
</tr>
<tr>
<td>tsvector_update_trigger()</td>
<td>trigger</td>
<td>trigger function for automatic tsvector column update</td>
<td><code>CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)</code></td>
<td></td>
</tr>
<tr>
<td>tsvector_update_trigger_column()</td>
<td>trigger</td>
<td>trigger function for automatic tsvector column update</td>
<td><code>CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)</code></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** All the text search functions that accept an optional `regconfig` argument will use the configuration specified by `default_text_search_config` when that argument is omitted.

The functions in Table 9-37 are listed separately because they are not usually used in everyday text searching operations. They are helpful for development and debugging of new text search configurations.
### Function Table

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ts_lexize(dict regdictionary, token text)</code></td>
<td><code>text[]</code></td>
<td>test a dictionary</td>
<td><code>ts_lexize('english_stem', 'stars')</code></td>
<td><code>{star}</code></td>
</tr>
<tr>
<td><code>ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)</code></td>
<td><code>setof record</code></td>
<td>test a parser</td>
<td><code>ts_parse('default', 'foo - bar')</code></td>
<td>(1,foo) ...</td>
</tr>
<tr>
<td><code>ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)</code></td>
<td><code>setof record</code></td>
<td>test a parser</td>
<td><code>ts_parse(3722, 'foo - bar')</code></td>
<td>(1,foo) ...</td>
</tr>
<tr>
<td><code>ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description text)</code></td>
<td><code>setof record</code></td>
<td>get token types defined by parser</td>
<td><code>ts_token_type('default')</code></td>
<td>(1,asciiword,&quot;Word, all ASCII&quot;) ...</td>
</tr>
<tr>
<td><code>ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description text)</code></td>
<td><code>setof record</code></td>
<td>get token types defined by parser</td>
<td><code>ts_token_type(3722)</code></td>
<td>(1,asciiword,&quot;Word, all ASCII&quot;) ...</td>
</tr>
<tr>
<td><code>ts_stat(sqlquery text, [weights text, ] OUT word text, OUT ndoc integer, OUT nentry integer)</code></td>
<td><code>setof record</code></td>
<td>get statistics of a tsvector column</td>
<td><code>ts_stat('SELECT vector from apod')</code></td>
<td>(foo,10,15) ...</td>
</tr>
</tbody>
</table>

### 9.14. XML Functions

The functions and function-like expressions described in this section operate on values of type xml. Check Section 8.13 for information about the xml type. The function-like expressions `xmlparse` and `xmlserialize` for converting to and from type xml are not repeated here. Use of many of these functions requires the installation to have been built with `configure --with-libxml`. 
9.14.1. Producing XML Content

A set of functions and function-like expressions are available for producing XML content from SQL data. As such, they are particularly suitable for formatting query results into XML documents for processing in client applications.

9.14.1.1. xmlcomment

xmlcomment(text)

The function xmlcomment creates an XML value containing an XML comment with the specified text as content. The text cannot contain -- or end with a - so that the resulting construct is a valid XML comment. If the argument is null, the result is null.

Example:

```sql
SELECT xmlcomment('hello');
```

xmlcomment
------------------
<!--hello-->

9.14.1.2. xmlconcat

xmlconcat(xml[, ...])

The function xmlconcat concatenates a list of individual XML values to create a single value containing an XML content fragment. Null values are omitted; the result is only null if there are no nonnull arguments.

Example:

```sql
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');
```

xmlconcat
------------------
<abc/><bar>foo</bar>

XML declarations, if present, are combined as follows. If all argument values have the same XML version declaration, that version is used in the result, else no version is used. If all argument values have the standalone declaration value "yes", then that value is used in the result. If all argument values have a standalone declaration value and at least one is "no", then that is used in the result. Else the result will have no standalone declaration. If the result is determined to require a standalone declaration but no version declaration, a version declaration with version 1.0 will be used because XML requires an XML declaration to contain a version declaration. Encoding declarations are ignored and removed in all cases.

Example:

```sql
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');
```
9.14.1.3. `xmlelement`  

`xmlelement(name name [, xmlattributes(value [AS attname] [, ... ])] [, content, ...])`

The `xmlelement` expression produces an XML element with the given name, attributes, and content. Examples:

```
SELECT xmlelement(name foo);

xmlelement
----------
<foo/>
```

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

xmlelement
----------
<foo bar="xyz"/>
```

```
SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

xmlelement
----------
<foo bar="2007-01-26">content</foo>
```

Element and attribute names that are not valid XML names are escaped by replacing the offending characters by the sequence `_xHHHH_`, where `HHHH` is the character’s Unicode codepoint in hexadecimal notation. For example:

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

xmlelement
----------
<foo_x0024_bar a_x0026_b="xyz"/>
```

An explicit attribute name need not be specified if the attribute value is a column reference, in which case the column’s name will be used as attribute name by default. In any other case, the attribute must be given an explicit name. So this example is valid:

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

But these are not:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```
Element content, if specified, will be formatted according to data type. If the content is itself of type xml, complex XML documents can be constructed. For example:

```sql
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
    xmlelement(name abc),
    xmlcomment('test'),
    xmlelement(name xyz));
```

```
xmlelement----------------------------------------------
<foo bar="xyz"><abc/><!--test-->xyz></foo>
```

Content of other types will be formatted into valid XML character data. This means in particular that the characters `<`, `>`, and `&` will be converted to entities. Binary data (data type `bytea`) will be represented in base64 or hex encoding, depending on the setting of the configuration parameter `xmlbinary`. The particular behavior for individual data types is expected to evolve in order to align the SQL and PostgreSQL data types with the XML Schema specification, at which point a more precise description will appear.

### 9.14.1.4. `xmlforest`

```sql
xmlforest(content [AS name] [, ...])
```

The `xmlforest` expression produces an XML forest (sequence) of elements using the given names and content.

Examples:

```sql
SELECT xmlforest('abc' AS foo, 123 AS bar);
```

```
xmlelement-------------------------------
<foo>abc</foo><bar>123</bar>
```

```sql
SELECT xmlforest(table_name, column_name) FROM information_schema.columns WHERE table_schema = 'pg_catalog';
```

```
xmlelement-------------------------------------------------------------------------------------------
<table_name>pg_authid</table_name><column_name>rolname</column_name>
<table_name>pg_authid</table_name><column_name>rolsuper</column_name>
...```

As seen in the second example, the element name can be omitted if the content value is a column reference, in which case the column name is used by default. Otherwise, a name must be specified. Element names that are not valid XML names are escaped as shown for `xmlelement` above. Similarly, content data is escaped to make valid XML content, unless it is already of type `xml`.

Note that XML forests are not valid XML documents if they consist of more than one element. So it might be useful to wrap `xmlforest` expressions in `xmlelement`.

195
9.14.1.5. xmlpi

xmlpi(name target [, content])

The xmlpi expression creates an XML processing instruction. The content, if present, must not contain the character sequence ?>

Example:

SELECT xmlpi(name php, 'echo "hello world";');

xmlpi
----------------------------
<?php echo "hello world";?>

9.14.1.6. xmlroot

xmlroot(xml, version text|no value [, standalone yes|no|no value])

The xmlroot expression alters the properties of the root node of an XML value. If a version is specified, this replaces the value in the version declaration, if a standalone value is specified, this replaces the value in the standalone declaration.

SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'), version '1.0', standalone yes);

xmlroot
----------------------------------------
<?xml version="1.0" standalone="yes"?><content>abc</content>

9.14.1.7. xmlagg

xmlagg(xml)

The function xmlagg is, unlike the other functions described here, an aggregate function. It concatenates the input values to the aggregate function call, like xmlconcat does. See Section 9.18 for general information about aggregate functions.

Example:

CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
xmlagg
----------------------
<foo>abc</foo><bar/>
Note that in the current implementation, the order of the concatenation is in principle undefined. Making the input values to be sorted in some other way will usually work, however. For instance, in the above example, one could influence the order like so:

```sql
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
```

```
xmlagg
----------------------
<bar/><foo>abc</foo>
```

But this approach is not guaranteed to work in all situations and in all versions of PostgreSQL. A future version of PostgreSQL will probably provide an additional feature to control the order in a proper way (`xmlagg(expr ORDER BY expr, expr, ...`).

### 9.14.1.8. XML Predicates

**xml** IS DOCUMENT

The expression **IS DOCUMENT** returns true if the argument XML value is a proper XML document, false if it is not (that is, it is a content fragment), or null if the argument is null. See Section 8.13 about the difference between documents and content fragments.

### 9.14.2. Processing XML

To process values of data type **xml**, PostgreSQL offers the function **xpath**, which evaluates XPath 1.0 expressions.

```sql
xpath(xpath, xml[, nsarray])
```

The function **xpath** evaluates the XPath expression **xpath** against the XML value **xml**. It returns an array of XML values corresponding to the node set produced by the XPath expression.

The third argument of the function is an array of namespace mappings. This array should be a two-dimensional array with the length of the second axis being equal to 2 (i.e., it should be an array of arrays, each of which consists of exactly 2 elements). The first element of each array entry is the namespace name, the second the namespace URI.

Example:

```sql
```

```
xpath
--------
{test}
(1 row)
```
Chapter 9. Functions and Operators

9.14.3. Mapping Tables to XML

The following functions map the contents of relational tables to XML values. They can be thought of as XML export functionality.

```sql
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean, tableforest boolean, targetns text)
```

The return type of each function is `xml`.

- `table_to_xml` maps the content of the named table, passed as parameter `tbl`. The `regclass` type accepts strings identifying tables using the usual notation, including optional schema qualifications and double quotes. `query_to_xml` executes the query whose text is passed as parameter `query` and maps the result set. `cursor_to_xml` fetches the indicated number of rows from the cursor specified by the parameter `cursor`. This variant is recommendable if large tables have to be mapped, because the result value is built up in memory by each function.

If `tableforest` is false, then the resulting XML document looks like this:

```xml
<tablename>
  <row>
    <columnname1 data</columnname1>
    <columnname2 data</columnname2>
  </row>
  ...
</tablename>
```

If `tableforest` is true, the result is an XML content fragment that looks like this:

```xml
<tablename>
  <columnname1 data</columnname1>
  <columnname2 data</columnname2>
</tablename>
<tablename>
  ...
</tablename>
<tablename>
  ...
</tablename>
```

If no table name is available, that is, when mapping a query or a cursor, the string `table` is used in the first format, `row` in the second format.

The choice between these formats is up to the user. The first format is a proper XML document, which will be important in many applications. The second format tends to be more useful in the `cursor_to_xml` function if the result values are to be reassembled into one document later on. The functions for producing XML content discussed above, in particular `xmlelement`, can be used to alter the results to taste.

The data values are mapped in the same way as described for the function `xmlelement` above.

The parameter `nulls` determines whether null values should be included in the output. If true, null values in columns are represented as
where xsi is the XML namespace prefix for XML Schema Instance. An appropriate namespace declaration will be added to the result value. If false, columns containing null values are simply omitted from the output.

The parameter targetns specifies the desired XML namespace of the result. If no particular namespace is wanted, an empty string should be passed.

The following functions return XML Schema documents describing the mappings made by the data mappings produced by the corresponding functions above.

```sql
table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)
```

It is essential that the same parameters are passed in order to obtain matching XML data mappings and XML Schema documents.

The following functions produce XML data mappings and the corresponding XML Schema in one document (or forest), linked together. They can be useful where self-contained and self-describing results are wanted.

```sql
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
```

In addition, the following functions are available to produce analogous mappings of entire schemas or the entire current database.

```sql
schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
```

```sql
database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)
```

Note that these potentially produce a lot of data, which needs to be built up in memory. When requesting content mappings of large schemas or databases, it may be worthwhile to consider mapping the tables separately instead, possibly even through a cursor.

The result of a schema content mapping looks like this:

```xml
<schemaname>
  table1-mapping
  table2-mapping
  ...
</schemaname>
```

where the format of a table mapping depends on the tableforest parameter as explained above.

The result of a database content mapping looks like this:

```xml
<dbname>
</dbname>
```
where the schema mapping is as above.

As an example for using the output produced by these functions, Figure 9-1 shows an XSLT stylesheet that converts the output of `table_to_xml_and_xmlschema` to an HTML document containing a tabular rendition of the table data. In a similar manner, the result data of these functions can be converted into other XML-based formats.

**Figure 9-1. XSLT stylesheet for converting SQL/XML output to HTML**

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml">
    <xsl:output method="xml"
        doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
        doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
        indent="yes"/>
    <xsl:template match="/">
        <html>
            <head>
                <title><xsl:value-of select="name(current())"></title>
            </head>
            <body>
                <table>
                    <tr>
                        <xsl:for-each select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/xsd:element/@name">
                            <th><xsl:value-of select="."/></th>
                        </xsl:for-each>
                    </tr>
                    <xsl:for-each select="row">
                        <tr>
                            <xsl:for-each select="*">
                                <td><xsl:value-of select="."/></td>
                            </xsl:for-each>
                        </tr>
                    </xsl:for-each>
                </table>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```
9.15. Sequence Manipulation Functions

This section describes PostgreSQL’s functions for operating on sequence objects. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with CREATE SEQUENCE. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions, listed in Table 9-38, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

Table 9-38. Sequence Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>currval(regclass)</td>
<td>bigint</td>
<td>Return value most recently obtained with nextval for specified sequence</td>
</tr>
<tr>
<td>lastval()</td>
<td>bigint</td>
<td>Return value most recently obtained with nextval for any sequence</td>
</tr>
<tr>
<td>nextval(regclass)</td>
<td>bigint</td>
<td>Advance sequence and return new value</td>
</tr>
<tr>
<td>setval(regclass, bigint)</td>
<td>bigint</td>
<td>Set sequence’s current value</td>
</tr>
<tr>
<td>setval(regclass, bigint, boolean)</td>
<td>bigint</td>
<td>Set sequence’s current value and is_called flag</td>
</tr>
</tbody>
</table>

The sequence to be operated on by a sequence-function call is specified by a regclass argument, which is just the OID of the sequence in the pg_class system catalog. You do not have to look up the OID by hand, however, since the regclass data type’s input converter will do the work for you. Just write the sequence name enclosed in single quotes, so that it looks like a literal constant. To achieve some compatibility with the handling of ordinary SQL names, the string will be converted to lowercase unless it contains double quotes around the sequence name. Thus:

nextval(‘foo’) operates on sequence foo
nextval(‘FOO’) operates on sequence foo
nextval(‘"Foo“’) operates on sequence Foo

The sequence name can be schema-qualified if necessary:

nextval(‘myschema.foo’) operates on myschema.foo
nextval(‘"myschema".foo’) same as above
nextval(‘foo’) searches search path for foo
See Section 8.16 for more information about regclass.

**Note:** Before PostgreSQL 8.1, the arguments of the sequence functions were of type text, not regclass, and the above-described conversion from a text string to an OID value would happen at run time during each call. For backwards compatibility, this facility still exists, but internally it is now handled as an implicit coercion from text to regclass before the function is invoked.

When you write the argument of a sequence function as an unadorned literal string, it becomes a constant of type regclass. Since this is really just an OID, it will track the originally identified sequence despite later renaming, schema reassignment, etc. This “early binding” behavior is usually desirable for sequence references in column defaults and views. But sometimes you will want “late binding” where the sequence reference is resolved at run time. To get late-binding behavior, force the constant to be stored as a text constant instead of regclass:

```
nextval('foo'::text)  foo is looked up at runtime
```

Note that late binding was the only behavior supported in PostgreSQL releases before 8.1, so you might need to do this to preserve the semantics of old applications.

Of course, the argument of a sequence function can be an expression as well as a constant. If it is a text expression then the implicit coercion will result in a run-time lookup.

The available sequence functions are:

- **nextval**
  
  Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute `nextval` concurrently, each will safely receive a distinct sequence value.

- **currval**
  
  Return the value most recently obtained by `nextval` for this sequence in the current session. (An error is reported if `nextval` has never been called for this sequence in this session.) Notice that because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed `nextval` since the current session did.

- **lastval**
  
  Return the value most recently returned by `nextval` in the current session. This function is identical to `currval`, except that instead of taking the sequence name as an argument it fetches the value of the last sequence that `nextval` was used on in the current session. It is an error to call `lastval` if `nextval` has not yet been called in the current session.

- **setval**
  
  Reset the sequence object’s counter value. The two-parameter form sets the sequence’s `last_value` field to the specified value and sets its `is_called` field to true, meaning that the next `nextval` will advance the sequence before returning a value. The value reported by `currval` is also set to the specified value. In the three-parameter form, `is_called` can be set either true or false. true has the same effect as the two-parameter form. If it’s set to false, the next `nextval` will return exactly the specified value, and sequence advancement commences with the following `nextval`. Furthermore, the value reported by `currval` is not changed in this case (this is a change from pre-8.3 behavior). For example,

```
SELECT setval('foo', 42);  Next nextval will return 43
SELECT setval('foo', 42, true);  Same as above
SELECT setval('foo', 42, false);  Next nextval will return 42
```
The result returned by `setval` is just the value of its second argument.

If a sequence object has been created with default parameters, `nextval` calls on it will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the `CREATE SEQUENCE` command; see its command reference page for more information.

**Important:** To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `nextval` operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the `nextval` later aborts. This means that aborted transactions might leave unused “holes” in the sequence of assigned values. `setval` operations are never rolled back, either.

### 9.16. Conditional Expressions

This section describes the SQL-compliant conditional expressions available in PostgreSQL.

**Tip:** If your needs go beyond the capabilities of these conditional expressions you might want to consider writing a stored procedure in a more expressive programming language.

#### 9.16.1. CASE

The SQL `CASE` expression is a generic conditional expression, similar to if/else statements in other languages:

```
CASE WHEN condition THEN result
    [WHEN ...]
    [ELSE result]
END
```

CASE clauses can be used wherever an expression is valid. `condition` is an expression that returns a boolean result. If the result is true then the value of the `CASE` expression is the `result` that follows the condition. If the result is false any subsequent `WHEN` clauses are searched in the same manner. If no `WHEN condition` is true then the value of the case expression is the `result` in the `ELSE` clause. If the `ELSE` clause is omitted and no condition matches, the result is null.

An example:

```
SELECT * FROM test;
```

```
a
---
1
2
3
```

```
SELECT a,
    CASE WHEN a=1 THEN 'one'
         WHEN a=2 THEN 'two'
```

203
The data types of all the result expressions must be convertible to a single output type. See Section 10.5 for more detail.

The following “simple” CASE expression is a specialized variant of the general form above:

```
CASE  expression
  WHEN  value  THEN  result
  [WHEN  ...]
  [ELSE  result]
END
```

The expression is computed and compared to all the value specifications in the WHEN clauses until one is found that is equal. If no match is found, the result in the ELSE clause (or a null value) is returned. This is similar to the switch statement in C.

The example above can be written using the simple CASE syntax:

```
SELECT a,
    CASE a WHEN 1 THEN 'one'
           WHEN 2 THEN 'two'
           ELSE 'other'
    END
FROM test;
```

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

\section{COALESCE}

\texttt{COALESCE(value [, ...])}
The **COALESCE** function returns the first of its arguments that is not null. Null is returned only if all arguments are null. It is often used to substitute a default value for null values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Like a **CASE** expression, **COALESCE** will not evaluate arguments that are not needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to **NVL** and **IFNULL**, which are used in some other database systems.

### 9.16.3. NULLIF

**NULLIF**(value1, value2)

The **NULLIF** function returns a null value if value1 and value2 are equal; otherwise it returns value1. This can be used to perform the inverse operation of the **COALESCE** example given above:

```
SELECT NULLIF(value, '(none)') ...
```

If value1 is (none), return a null, otherwise return value1.

### 9.16.4. GREATEST and LEAST

**GREATEST**(value [, ...])

**LEAST**(value [, ...])

The **GREATEST** and **LEAST** functions select the largest or smallest value from a list of any number of expressions. The expressions must all be convertible to a common data type, which will be the type of the result (see Section 10.5 for details). NULL values in the list are ignored. The result will be NULL only if all the expressions evaluate to NULL.

Note that **GREATEST** and **LEAST** are not in the SQL standard, but are a common extension. Some other databases make them return NULL if any argument is NULL, rather than only when all are NULL.

### 9.17. Array Functions and Operators

Table 9-39 shows the operators available for array types.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-</code></td>
<td>equal</td>
<td>ARRAY[1.1,2.1,3.1] == int[] = ARRAY[1,2,3]</td>
<td></td>
</tr>
</tbody>
</table>
Array comparisons compare the array contents element-by-element, using the default B-Tree comparison function for the element data type. In multidimensional arrays the elements are visited in row-major order (last subscript varies most rapidly). If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order. (This is a change from versions of PostgreSQL prior to 8.2: older versions would claim that two arrays with the same contents were equal, even if the number of dimensions or subscript ranges were different.)

See Section 8.14 for more details about array operator behavior.

Table 9-40 shows the functions available for use with array types. See Section 8.14 for more discussion and examples of the use of these functions.

Table 9-40. array Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_append</td>
<td>anyarray</td>
<td>append an element to the end of an array</td>
<td>array_append(ARRAY[1,2], 3)</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>array_cat</td>
<td>anyarray</td>
<td>concatenate two arrays</td>
<td>array_cat(ARRAY[1,2,3], ARRAY[4,5])</td>
<td>{1,2,3,4,5}</td>
</tr>
<tr>
<td>@</td>
<td></td>
<td>contains</td>
<td>ARRAY[1,4,3] @ ARRAY[3,1]</td>
<td>t</td>
</tr>
<tr>
<td>&lt; @</td>
<td></td>
<td>is contained by</td>
<td>ARRAY[2,7] &lt; @ ARRAY[1,7,4,2,6]</td>
<td>t</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
<td>overlap (have elements in common)</td>
<td>ARRAY[1,4,3] &amp;&amp; ARRAY[2,1]</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>array-to-array concatenation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>array-to-array concatenation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>element-to-array concatenation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>array-to-element concatenation</td>
</tr>
</tbody>
</table>
### 9.18. Aggregate Functions

**Aggregate functions** compute a single result value from a set of input values. The built-in aggregate functions are listed in Table 9-41 and Table 9-42. The special syntax considerations for aggregate functions are explained in Section 4.2.7. Consult Section 2.7 for additional introductory information.

#### Table 9-41. General-Purpose Aggregate Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument Type</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg(expression)</td>
<td>smallint, int, bigint, real,</td>
<td>numeric</td>
<td>the average (arithmetic mean) of all input values</td>
<td>avg(10, 20, 30)</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>double precision, numeric, or</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>interval</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit_and(expression)</td>
<td>smallint, bigint, or bit</td>
<td>same as</td>
<td>the bitwise AND of all non-null input values, or null if none</td>
<td>bit_and(10, 20, 30)</td>
<td>10</td>
</tr>
<tr>
<td>bit_or(expression)</td>
<td>smallint, bigint, or bit</td>
<td>same as</td>
<td>the bitwise OR of all non-null input values, or null if none</td>
<td>bit_or(10, 20, 30)</td>
<td>11</td>
</tr>
</tbody>
</table>
### Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument Type</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool_and(expression)</code></td>
<td><code>bool</code></td>
<td><code>bool</code></td>
<td>true if all input values are true, otherwise false</td>
</tr>
<tr>
<td><code>bool_or(expression)</code></td>
<td><code>bool</code></td>
<td><code>bool</code></td>
<td>true if at least one input value is true, otherwise false</td>
</tr>
<tr>
<td><code>count(*)</code></td>
<td></td>
<td><code>bigint</code></td>
<td>number of input rows</td>
</tr>
<tr>
<td><code>count(expression)</code></td>
<td>any</td>
<td><code>bigint</code></td>
<td>number of input rows for which the value of <code>expression</code> is not null</td>
</tr>
<tr>
<td><code>every(expression)</code></td>
<td><code>bool</code></td>
<td><code>bool</code></td>
<td>equivalent to <code>bool_and</code></td>
</tr>
<tr>
<td><code>max(expression)</code></td>
<td>any array, numeric,</td>
<td><code>same as</code></td>
<td>maximum value of <code>expression</code> across all input values</td>
</tr>
<tr>
<td></td>
<td>string, or date/time type</td>
<td>argument type</td>
<td></td>
</tr>
<tr>
<td><code>min(expression)</code></td>
<td>any array, numeric,</td>
<td><code>same as</code></td>
<td>minimum value of <code>expression</code> across all input values</td>
</tr>
<tr>
<td></td>
<td>string, or date/time type</td>
<td>argument type</td>
<td></td>
</tr>
<tr>
<td><code>sum(expression)</code></td>
<td><code>smallint</code>, <code>int</code>,</td>
<td><code>bigint</code></td>
<td>sum of <code>expression</code> across all input values</td>
</tr>
<tr>
<td></td>
<td><code>bigint</code>, <code>real</code>,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>double precision</code>,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>numeric</code>, <code>or</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>interval</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It should be noted that except for `count`, these functions return a null value when no rows are selected. In particular, `sum` of no rows returns null, not zero as one might expect. The `coalesce` function can be used to substitute zero for null when necessary.

**Note:** Boolean aggregates `bool_and` and `bool_or` correspond to standard SQL aggregates `every` and `any` or `some`. As for `any` and `some`, it seems that there is an ambiguity built into the standard syntax:

```sql
SELECT b1 = ANY(SELECT b2 FROM t2 ...)) FROM t1 ...
```

Here `ANY` can be considered both as leading to a subquery or as an aggregate if the select expression returns 1 row. Thus the standard name cannot be given to these aggregates.

**Note:** Users accustomed to working with other SQL database management systems might be surprised by the performance of the `count` aggregate when it is applied to the entire table. A query like:

```sql
SELECT count(*) FROM sometable;
```

will be executed by PostgreSQL using a sequential scan of the entire table.
Table 9-42 shows aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions \( N \), it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when \( N \) is zero.

### Table 9-42. Aggregate Functions for Statistics

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument Type</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{corr}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>correlation coefficient</td>
</tr>
<tr>
<td>( \text{covar}_{\text{pop}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>population covariance</td>
</tr>
<tr>
<td>( \text{covar}_{\text{samp}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>sample covariance</td>
</tr>
<tr>
<td>( \text{regr}_{\text{avgx}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>average of the independent variable ((\text{sum}(X)/N))</td>
</tr>
<tr>
<td>( \text{regr}_{\text{avgx}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>average of the dependent variable ((\text{sum}(Y)/N))</td>
</tr>
<tr>
<td>( \text{regr}_{\text{count}}(Y, X) )</td>
<td>double precision</td>
<td>bigint</td>
<td>number of input rows in which both expressions are nonnull</td>
</tr>
<tr>
<td>( \text{regr}_{\text{intercept}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>y-intercept of the least-squares-fit linear equation determined by the ((X, Y)) pairs</td>
</tr>
<tr>
<td>( \text{regr}_{\text{r2}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>square of the correlation coefficient</td>
</tr>
<tr>
<td>( \text{regr}_{\text{slope}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>slope of the least-squares-fit linear equation determined by the ((X, Y)) pairs</td>
</tr>
<tr>
<td>( \text{regr}_{\text{sxx}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>(\text{sum}(X^2) - \text{sum}(X)^2/N) (&quot;sum of squares&quot; of the independent variable)</td>
</tr>
<tr>
<td>( \text{regr}_{\text{sxy}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>(\text{sum}(X\cdot Y) - \text{sum}(X)\cdot \text{sum}(Y)/N) (&quot;sum of products&quot; of independent times dependent variable)</td>
</tr>
<tr>
<td>( \text{regr}_{\text{syy}}(Y, X) )</td>
<td>double precision</td>
<td>double precision</td>
<td>(\text{sum}(Y^2) - \text{sum}(Y)^2/N) (&quot;sum of squares&quot; of the dependent variable)</td>
</tr>
</tbody>
</table>
9.19. Subquery Expressions

This section describes the SQL-compliant subquery expressions available in PostgreSQL. All of the expression forms documented in this section return Boolean (true/false) results.

9.19.1. EXISTS

 EXISTS (subquery)

The argument of EXISTS is an arbitrary SELECT statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is “true”; if the subquery returns no rows, the result of EXISTS is “false”.

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side effects (such as calling sequence functions); whether the side effects occur or not might be difficult to predict.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all EXISTS tests in the form EXISTS(SELECT 1 WHERE ...). There are exceptions to this rule however, such as subqueries that use INTERSECT.
This simple example is like an inner join on `col2`, but it produces at most one output row for each `tab1` row, even if there are multiple matching `tab2` rows:

```sql
SELECT col1 FROM tab1 WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

### 9.19.2. **IN**

`expression IN (subquery)`

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of **IN** is “true” if any equal subquery row is found. The result is “false” if no equal row is found (including the special case where the subquery returns no rows).

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the **IN** construct will be null, not false. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

As with **EXISTS**, it’s unwise to assume that the subquery will be evaluated completely.

```sql
row_constructor IN (subquery)
```

The left-hand side of this form of **IN** is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of **IN** is “true” if any equal subquery row is found. The result is “false” if no equal row is found (including the special case where the subquery returns no rows).

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of **IN** is null.

### 9.19.3. **NOT IN**

`expression NOT IN (subquery)`

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of **NOT IN** is “true” if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is “false” if any equal row is found.

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the **NOT IN** construct will be null, not true. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

As with **EXISTS**, it’s unwise to assume that the subquery will be evaluated completely.

```sql
row_constructor NOT IN (subquery)
```
Chapter 9. Functions and Operators

The left-hand side of this form of \texttt{NOT IN} is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of \texttt{NOT IN} is “true” if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is “false” if any equal row is found.

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of \texttt{NOT IN} is null.

9.19.4. \texttt{ANY}/\texttt{SOME}

\begin{verbatim}
expression operator ANY (subquery)
expression operator SOME (subquery)
\end{verbatim}

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given \texttt{operator}, which must yield a Boolean result. The result of \texttt{ANY} is “true” if any true result is obtained. The result is “false” if no true result is found (including the special case where the subquery returns no rows). \texttt{SOME} is a synonym for \texttt{ANY}. \texttt{IN} is equivalent to \texttt{= ANY}.

Note that if there are no successes and at least one right-hand row yields null for the operator’s result, the result of the \texttt{ANY} construct will be null, not false. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

As with \texttt{EXISTS}, it’s unwise to assume that the subquery will be evaluated completely.

\begin{verbatim}
row_constructor operator ANY (subquery)
row_constructor operator SOME (subquery)
\end{verbatim}

The left-hand side of this form of \texttt{ANY} is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given \texttt{operator}. The result of \texttt{ANY} is “true” if the comparison returns true for any subquery row. The result is “false” if the comparison returns false for every subquery row (including the special case where the subquery returns no rows). The result is NULL if the comparison does not return true for any row, and it returns NULL for at least one row.

See Section 9.20.5 for details about the meaning of a row-wise comparison.

9.19.5. \texttt{ALL}

\begin{verbatim}
expression operator ALL (subquery)
\end{verbatim}

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given \texttt{operator}, which must yield a Boolean result. The result of \texttt{ALL} is “true” if all rows yield true (including the special case where the subquery returns no rows). The result is “false” if any false result is found. The result is NULL if the comparison does not return false for any row, and it returns NULL for at least one row.
Chapter 9. Functions and Operators

**NOT IN** is equivalent to `<>` **ALL**.

As with **EXISTS**, it’s unwise to assume that the subquery will be evaluated completely.

```
row_constructor operator ALL (subquery)
```

The left-hand side of this form of **ALL** is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given **operator**. The result of **ALL** is “true” if the comparison returns true for all subquery rows (including the special case where the subquery returns no rows). The result is “false” if the comparison returns false for any subquery row. The result is NULL if the comparison does not return false for any subquery row, and it returns NULL for at least one row.

See Section 9.20.5 for details about the meaning of a row-wise comparison.

### 9.19.6. Row-wise Comparison

```
row_constructor operator (subquery)
```

The left-hand side is a row constructor, as described in Section 4.2.11. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. Furthermore, the subquery cannot return more than one row. (If it returns zero rows, the result is taken to be null.) The left-hand side is evaluated and compared row-wise to the single subquery result row.

See Section 9.20.5 for details about the meaning of a row-wise comparison.

### 9.20. Row and Array Comparisons

This section describes several specialized constructs for making multiple comparisons between groups of values. These forms are syntactically related to the subquery forms of the previous section, but do not involve subqueries. The forms involving array subexpressions are PostgreSQL extensions; the rest are SQL-compliant. All of the expression forms documented in this section return Boolean (true/false) results.

#### 9.20.1. IN

```
expression IN (value [, ...])
```

The right-hand side is a parenthesized list of scalar expressions. The result is “true” if the left-hand expression’s result is equal to any of the right-hand expressions. This is a shorthand notation for

```
expression = value1
OR
expression = value2
OR
...
```
Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the IN construct will be null, not false. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

### 9.20.2. NOT IN

```sql
expression NOT IN (value [, ...])
```

The right-hand side is a parenthesized list of scalar expressions. The result is “true” if the left-hand expression’s result is unequal to all of the right-hand expressions. This is a shorthand notation for

```sql
expression <> value1 
AND 
expression <> value2 
AND 
...
```

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the NOT IN construct will be null, not true as one might naively expect. This is in accordance with SQL’s normal rules for Boolean combinations of null values.

**Tip:** \( x \ NOT \ IN \ y \) is equivalent to \( NOT \ (x \ IN \ y) \) in all cases. However, null values are much more likely to trip up the novice when working with NOT IN than when working with IN. It’s best to express your condition positively if possible.

### 9.20.3. ANY/SOME (array)

```sql
expression operator ANY (array expression) 
expression operator SOME (array expression) 
```

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given operator, which must yield a Boolean result. The result of ANY is “true” if any true result is obtained. The result is “false” if no true result is found (including the special case where the array has zero elements).

If the array expression yields a null array, the result of ANY will be null. If the left-hand expression yields null, the result of ANY is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no true comparison result is obtained, the result of ANY will be null, not false (again, assuming a strict comparison operator). This is in accordance with SQL’s normal rules for Boolean combinations of null values.

SOME is a synonym for ANY.

### 9.20.4. ALL (array)

```sql
expression operator ALL (array expression) 
```
The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given operator, which must yield a Boolean result. The result of \texttt{ALL} is "true" if all comparisons yield true (including the special case where the array has zero elements). The result is "false" if any false result is found.

If the array expression yields a null array, the result of \texttt{ALL} will be null. If the left-hand expression yields null, the result of \texttt{ALL} is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no false comparison result is obtained, the result of \texttt{ALL} will be null, not true (again, assuming a strict comparison operator). This is in accordance with SQL’s normal rules for Boolean combinations of null values.

\section*{9.20.5. Row-wise Comparison}

Each side is a row constructor, as described in Section 4.2.11. The two row values must have the same number of fields. Each side is evaluated and they are compared row-wise. Row comparisons are allowed when the operator is \texttt{=}, \texttt{<>}, \texttt{<}, \texttt{<=}, \texttt{>}, or \texttt{>=}, or has semantics similar to one of these. (To be specific, an operator can be a row comparison operator if it is a member of a B-Tree operator class, or is the negator of the \texttt{=} member of a B-Tree operator class.)

The \texttt{=} and \texttt{<>} cases work slightly differently from the others. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of the row comparison is unknown (null).

For the \texttt{<}, \texttt{<}, \texttt{=}, \texttt{>}, and \texttt{>=} cases, the row elements are compared left-to-right, stopping as soon as an unequal or null pair of elements is found. If either of this pair of elements is null, the result of the row comparison is unknown (null); otherwise comparison of this pair of elements determines the result. For example, \texttt{ROW(1,2,NULL) < ROW(1,3,0)} yields true, not null, because the third pair of elements are not considered.

\textbf{Note:} Prior to PostgreSQL 8.2, the \texttt{<}, \texttt{<}, \texttt{>}, and \texttt{>=} cases were not handled per SQL specification. A comparison like \texttt{ROW(a,b) < ROW(c,d)} was implemented as \texttt{a < c AND b < d} whereas the correct behavior is equivalent to \texttt{a < c OR (a = c AND b < d)}.

\texttt{row_constructor IS DISTINCT FROM row_constructor}

This construct is similar to a \texttt{<>} row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will always be either true or false, never null.

\texttt{row_constructor IS NOT DISTINCT FROM row_constructor}

This construct is similar to a \texttt{=} row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will always be either true or false, never null.
9.21. Set Returning Functions

This section describes functions that possibly return more than one row. Currently the only functions in this class are series generating functions, as detailed in Table 9-43.

Table 9-43. Series Generating Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument Type</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate_series(start, stop)</td>
<td>int or bigint</td>
<td>setof int or setof bigint (same as argument type)</td>
<td>Generate a series of values, from start to stop with a step size of one</td>
</tr>
<tr>
<td>generate_series(start, stop, step)</td>
<td>int or bigint</td>
<td>setof int or setof bigint (same as argument type)</td>
<td>Generate a series of values, from start to stop with a step size of step</td>
</tr>
</tbody>
</table>

When step is positive, zero rows are returned if start is greater than stop. Conversely, when step is negative, zero rows are returned if start is less than stop. Zero rows are also returned for NULL inputs. It is an error for step to be zero. Some examples follow:

```sql
select * from generate_series(2,4);
generate_series
---
2
3
4
(3 rows)

select * from generate_series(5,1,-2);
generate_series
---
5
3
1
(3 rows)

select * from generate_series(4,3);
generate_series
---
(0 rows)

select current_date + s.a as dates from generate_series(0,14,7) as s(a);
dates
---
2004-02-05
2004-02-12
2004-02-19
(3 rows)
```
9.22. System Information Functions

Table 9-44 shows several functions that extract session and system information.

In addition to the functions listed in this section, there are a number of functions related to the statistics system that also provide system information. See Section 26.2.2 for more information.

Table 9-44. Session Information Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_database()</td>
<td>name</td>
<td>name of current database</td>
</tr>
<tr>
<td>current_schema()</td>
<td>name</td>
<td>name of current schema</td>
</tr>
<tr>
<td>current_schemas(boolean)</td>
<td>name[]</td>
<td>names of schemas in search path optionally including implicit schemas</td>
</tr>
<tr>
<td>current_user</td>
<td>name</td>
<td>user name of current execution context</td>
</tr>
<tr>
<td>inet_client_addr()</td>
<td>inet</td>
<td>address of the remote connection</td>
</tr>
<tr>
<td>inet_client_port()</td>
<td>int</td>
<td>port of the remote connection</td>
</tr>
<tr>
<td>inet_server_addr()</td>
<td>inet</td>
<td>address of the local connection</td>
</tr>
<tr>
<td>inet_server_port()</td>
<td>int</td>
<td>port of the local connection</td>
</tr>
<tr>
<td>pg_my_temp_schema()</td>
<td>oid</td>
<td>OID of session’s temporary schema, or 0 if none</td>
</tr>
<tr>
<td>pg_is_other_temp_schema(oid)</td>
<td>boolean</td>
<td>is schema another session’s temporary schema?</td>
</tr>
<tr>
<td>pg_postmaster_start_time()</td>
<td>timestamp with time zone</td>
<td>server start time</td>
</tr>
<tr>
<td>session_user</td>
<td>name</td>
<td>session user name</td>
</tr>
<tr>
<td>user</td>
<td>name</td>
<td>equivalent to current_user</td>
</tr>
<tr>
<td>version()</td>
<td>text</td>
<td>PostgreSQL version information</td>
</tr>
</tbody>
</table>

The `session_user` is normally the user who initiated the current database connection; but superusers can change this setting with `SET SESSION AUTHORIZATION`. The `current_user` is the user identifier that is applicable for permission checking. Normally, it is equal to the session user, but it can be changed with `SET ROLE`. It also changes during the execution of functions with the attribute `SECURITY DEFINER`. In Unix parlance, the session user is the “real user” and the current user is the “effective user”.

**Note:** `current_user`, `session_user`, and `user` have special syntactic status in SQL: they must be called without trailing parentheses.

current_schema returns the name of the schema that is at the front of the search path (or a null value if the search path is empty). This is the schema that will be used for any tables or other named objects that are created without specifying a target schema. current_schemas(boolean) returns an array of the names of all schemas presently in the search path. The Boolean option determines whether or not implicitly included system schemas such as `pg_catalog` are included in the search path returned.
Chapter 9. Functions and Operators

**Note:** The search path can be altered at run time. The command is:

```
SET search_path TO schema [, schema, ...]
```

inet_client_addr returns the IP address of the current client, and inet_client_port returns the port number. inet_server_addr returns the IP address on which the server accepted the current connection, and inet_server_port returns the port number. All these functions return NULL if the current connection is via a Unix-domain socket.

pg_my_temp_schema returns the OID of the current session’s temporary schema, or 0 if it has none (because it has not created any temporary tables). pg_is_other_temp_schema returns true if the given OID is the OID of any other session’s temporary schema. (This can be useful, for example, to exclude other sessions’ temporary tables from a catalog display.)

pg_postmaster_start_time returns the timestamp with time zone when the server started. version returns a string describing the PostgreSQL server’s version.

Table 9-45 lists functions that allow the user to query object access privileges programmatically. See Section 5.6 for more information about privileges.

### Table 9-45. Access Privilege Inquiry Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_database_privilege(user,</td>
<td>boolean</td>
<td>does user have privilege for database</td>
</tr>
<tr>
<td>database, privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_database_privilege(database)</td>
<td>boolean</td>
<td>does current user have privilege for database</td>
</tr>
<tr>
<td>has_function_privilege(user,</td>
<td>boolean</td>
<td>does user have privilege for function</td>
</tr>
<tr>
<td>function, privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_function_privilege(function)</td>
<td>boolean</td>
<td>does current user have privilege for function</td>
</tr>
<tr>
<td>has_language_privilege(user,</td>
<td>boolean</td>
<td>does user have privilege for language</td>
</tr>
<tr>
<td>language, privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_language_privilege(language)</td>
<td>boolean</td>
<td>does current user have privilege for language</td>
</tr>
<tr>
<td>has_schema_privilege(user,</td>
<td>boolean</td>
<td>does user have privilege for schema</td>
</tr>
<tr>
<td>schema, privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_schema_privilege(schema,</td>
<td>boolean</td>
<td>does current user have privilege for schema</td>
</tr>
<tr>
<td>privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_table_privilege(user,</td>
<td>boolean</td>
<td>does user have privilege for table</td>
</tr>
<tr>
<td>table, privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_table_privilege(table,</td>
<td>boolean</td>
<td>does current user have privilege for table</td>
</tr>
<tr>
<td>privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_tablespace_privilege(user,</td>
<td>boolean</td>
<td>does user have privilege for tablespace</td>
</tr>
<tr>
<td>tablespace, privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_tablespace_privilege(tablespace, privilege)</td>
<td>boolean</td>
<td>does current user have privilege for tablespace</td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pg_has_role(user, role, privilege)</code></td>
<td>boolean</td>
<td>does user have privilege for role</td>
</tr>
<tr>
<td><code>pg_has_role(role, privilege)</code></td>
<td>boolean</td>
<td>does current user have privilege for role</td>
</tr>
</tbody>
</table>

**has_database_privilege** checks whether a user can access a database in a particular way. The possibilities for its arguments are analogous to **has_table_privilege**. The desired access privilege type must evaluate to **CREATE**, **CONNECT**, **TEMPORARY**, or **TEMP** (which is equivalent to **TEMPORARY**).

**has_function_privilege** checks whether a user can access a function in a particular way. The possibilities for its arguments are analogous to **has_table_privilege**. When specifying a function by a text string rather than by OID, the allowed input is the same as for the **regprocedure** data type (see Section 8.16). The desired access privilege type must evaluate to **EXECUTE**. An example is:

```sql
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

**has_language_privilege** checks whether a user can access a procedural language in a particular way. The possibilities for its arguments are analogous to **has_table_privilege**. The desired access privilege type must evaluate to **USAGE**.

**has_schema_privilege** checks whether a user can access a schema in a particular way. The possibilities for its arguments are analogous to **has_table_privilege**. The desired access privilege type must evaluate to **CREATE** or **USAGE**.

**has_table_privilege** checks whether a user can access a table in a particular way. The user can be specified by name or by OID (**pg_authid.oid**), or if the argument is omitted **current_user** is assumed. The table can be specified by name or by OID. (Thus, there are actually six variants of **has_table_privilege**, which can be distinguished by the number and types of their arguments.) When specifying by name, the name can be schema-qualified if necessary. The desired access privilege type is specified by a text string, which must evaluate to one of the values **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **REFERENCES**, or **TRIGGER**. (Case of the string is not significant, however.) An example is:

```sql
SELECT has_table_privilege('myschema.mytable', 'select');
```

**has_tablespace_privilege** checks whether a user can access a tablespace in a particular way. The possibilities for its arguments are analogous to **has_table_privilege**. The desired access privilege type must evaluate to **CREATE**.

**pg_has_role** checks whether a user can access a role in a particular way. The possibilities for its arguments are analogous to **has_table_privilege**. The desired access privilege type must evaluate to **MEMBER** or **USAGE**. **MEMBER** denotes direct or indirect membership in the role (that is, the right to do **SET ROLE**), while **USAGE** denotes whether the privileges of the role are immediately available without doing **SET ROLE**.

To test whether a user holds a grant option on the privilege, append **WITH GRANT OPTION** to the privilege key word; for example **'UPDATE WITH GRANT OPTION'**.

Table 9-46 shows functions that determine whether a certain object is **visible** in the current schema search path. For example, a table is said to be visible if its containing schema is in the search path and no table of the same name appears earlier in the search path. This is equivalent to the statement
that the table can be referenced by name without explicit schema qualification. To list the names of all visible tables:

```sql
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

**Table 9-46. Schema Visibility Inquiry Functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_conversion_is_visible(function_oid)</td>
<td>boolean</td>
<td>is conversion visible in search path</td>
</tr>
<tr>
<td>pg_function_is_visible(function_oid)</td>
<td>boolean</td>
<td>is function visible in search path</td>
</tr>
<tr>
<td>pg_operator_is_visible(operator_oid)</td>
<td>boolean</td>
<td>is operator visible in search path</td>
</tr>
<tr>
<td>pg_opclass_is_visible(opclass_oid)</td>
<td>boolean</td>
<td>is operator class visible in search path</td>
</tr>
<tr>
<td>pg_table_is_visible(table_oid)</td>
<td>boolean</td>
<td>is table visible in search path</td>
</tr>
<tr>
<td>pg_ts_config_is_visible(config_oid)</td>
<td>boolean</td>
<td>is text search configuration visible in search path</td>
</tr>
<tr>
<td>pg_ts_dict_is_visible(dict_oid)</td>
<td>boolean</td>
<td>is text search dictionary visible in search path</td>
</tr>
<tr>
<td>pg_ts_parser_is_visible(parser_oid)</td>
<td>boolean</td>
<td>is text search parser visible in search path</td>
</tr>
<tr>
<td>pg_ts_template_is_visible(template_oid)</td>
<td>boolean</td>
<td>is text search template visible in search path</td>
</tr>
<tr>
<td>pg_type_is_visible(type_oid)</td>
<td>boolean</td>
<td>is type (or domain) visible in search path</td>
</tr>
</tbody>
</table>

Each function performs the visibility check for one type of database object. Note that `pg_table_is_visible` can also be used with views, indexes and sequences; `pg_type_is_visible` can also be used with domains. For functions and operators, an object in the search path is visible if there is no object of the same name and argument data type(s) earlier in the path. For operator classes, both name and associated index access method are considered.

All these functions require object OIDs to identify the object to be checked. If you want to test an object by name, it is convenient to use the OID alias types (`regclass`, `regtype`, `regprocedure`, `regoperator`, `regconfig`, or `regdictionary`), for example:

```sql
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Note that it would not make much sense to test an unqualified name in this way — if the name can be recognized at all, it must be visible.

Table 9-47 lists functions that extract information from the system catalogs.

**Table 9-47. System Catalog Information Functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
</table>

220
<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>format_type(type_oid, typemod)</code></td>
<td>text</td>
<td>get SQL name of a data type</td>
</tr>
<tr>
<td><code>pg_get_constraintdef(constraint_oid)</code></td>
<td>text</td>
<td>get definition of a constraint</td>
</tr>
<tr>
<td><code>pg_get_constraintdef(constraint_oid, pretty_bool)</code></td>
<td>text</td>
<td>get definition of a constraint</td>
</tr>
<tr>
<td><code>pg_get_expr(expr_text, relation_oid)</code></td>
<td>text</td>
<td>decompile internal form of an expression, assuming that any Vars in it refer to the relation indicated by the second parameter</td>
</tr>
<tr>
<td><code>pg_get_expr(expr_text, relation_oid, pretty_bool)</code></td>
<td>text</td>
<td>decompile internal form of an expression, assuming that any Vars in it refer to the relation indicated by the second parameter</td>
</tr>
<tr>
<td><code>pg_get_indexdef(index_oid)</code></td>
<td>text</td>
<td>get CREATE INDEX command for index</td>
</tr>
<tr>
<td><code>pg_get_indexdef(index_oid, column_no, pretty_bool)</code></td>
<td>text</td>
<td>get CREATE INDEX command for index, or definition of just one index column when column_no is not zero</td>
</tr>
<tr>
<td><code>pg_get_ruledef(rule_oid)</code></td>
<td>text</td>
<td>get CREATE RULE command for rule</td>
</tr>
<tr>
<td><code>pg_get_ruledef(rule_oid, pretty_bool)</code></td>
<td>text</td>
<td>get CREATE RULE command for rule</td>
</tr>
<tr>
<td><code>pg_get_serial_sequence(table_name, column_name)</code></td>
<td>text</td>
<td>get name of the sequence that a serial or bigserial column uses</td>
</tr>
<tr>
<td><code>pg_get_triggerdef(trigger_oid)</code></td>
<td>text</td>
<td>get CREATE [ CONSTRAINT ] TRIGGER command for trigger</td>
</tr>
<tr>
<td><code>pg_get_userbyid(roleid)</code></td>
<td>name</td>
<td>get role name with given ID</td>
</tr>
<tr>
<td><code>pg_get_viewdef(view_name)</code></td>
<td>text</td>
<td>get underlying SELECT command for view (deprecated)</td>
</tr>
<tr>
<td><code>pg_get_viewdef(view_name, pretty_bool)</code></td>
<td>text</td>
<td>get underlying SELECT command for view (deprecated)</td>
</tr>
<tr>
<td><code>pg_get_viewdef(view_oid)</code></td>
<td>text</td>
<td>get underlying SELECT command for view</td>
</tr>
<tr>
<td><code>pg_get_viewdef(view_oid, pretty_bool)</code></td>
<td>text</td>
<td>get underlying SELECT command for view</td>
</tr>
<tr>
<td><code>pg_tablespace_databases(tablespace_oid)</code></td>
<td>setof oid</td>
<td>get the set of database OIDs that have objects in the tablescape</td>
</tr>
</tbody>
</table>

`format_type` returns the SQL name of a data type that is identified by its type OID and possibly a
type modifier. Pass NULL for the type modifier if no specific modifier is known.

pg_get_constraintdef, pg_get_indexdef, pg_get_ruledef, and pg_get_triggerdef, respectively reconstruct the creating command for a constraint, index, rule, or trigger. (Note that this is a decompiled reconstruction, not the original text of the command.) pg_get_expr decompiles the internal form of an individual expression, such as the default value for a column. It can be useful when examining the contents of system catalogs. pg_get_viewdef reconstructs the SELECT query that defines a view. Most of these functions come in two variants, one of which can optionally “pretty-print” the result. The pretty-printed format is more readable, but the default format is more likely to be interpreted the same way by future versions of PostgreSQL; avoid using pretty-printed output for dump purposes. Passing false for the pretty-print parameter yields the same result as the variant that does not have the parameter at all.

pg_get_serial_sequence returns the name of the sequence associated with a column, or NULL if no sequence is associated with the column. The first input parameter is a table name with optional schema, and the second parameter is a column name. Because the first parameter is potentially a schema and table, it is not treated as a double-quoted identifier, meaning it is lowercased by default, while the second parameter, being just a column name, is treated as double-quoted and has its case preserved. The function returns a value suitably formatted for passing to the sequence functions (see Section 9.15). This association can be modified or removed with ALTER SEQUENCE OWNED BY. (The function probably should have been called pg_get_owned_sequence; its name reflects the fact that it’s typically used with serial or bigserial columns.)

pg_get_userbyid extracts a role’s name given its OID.

pg_tablespace_databases allows a tablespace to be examined. It returns the set of OIDs of databases that have objects stored in the tablespace. If this function returns any rows, the tablespace is not empty and cannot be dropped. To display the specific objects populating the tablespace, you will need to connect to the databases identified by pg_tablespace_databases and query their pg_class catalogs.

The functions shown in Table 9-48 extract comments previously stored with the COMMENT command. A null value is returned if no comment could be found matching the specified parameters.

### Table 9-48. Comment Information Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_description(table_oid, column_number)</td>
<td>text</td>
<td>get comment for a table column</td>
</tr>
<tr>
<td>obj_description(object_oid, catalog_name)</td>
<td>text</td>
<td>get comment for a database object</td>
</tr>
<tr>
<td>obj_description(object_oid)</td>
<td>text</td>
<td>get comment for a database object (deprecated)</td>
</tr>
<tr>
<td>shobj_description(object_oid, catalog_name)</td>
<td>text</td>
<td>get comment for a shared database object</td>
</tr>
</tbody>
</table>

col_description returns the comment for a table column, which is specified by the OID of its table and its column number. obj_description cannot be used for table columns since columns do not have OIDs of their own.

The two-parameter form of obj_description returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, obj_description(123456, ‘pg_class’) would retrieve the comment for a table with OID 123456. The one-parameter form of obj_description requires only the object OID. It is now deprecated since there is no guarantee that OIDs are unique across different system catalogs;
therefore, the wrong comment could be returned.

shobj_description is used just like obj_description only that it is used for retrieving comments on shared objects. Some system catalogs are global to all databases within each cluster and their descriptions are stored globally as well.

The functions shown in Table 9-49 export server internal transaction information to user level. The main use of these functions is to determine which transactions were committed between two snapshots.

### Table 9-49. Transaction IDs and snapshots

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txid_current()</td>
<td>bigint</td>
<td>get current transaction ID</td>
</tr>
<tr>
<td>txid_current_snapshot()</td>
<td>txid_snapshot</td>
<td>get current snapshot</td>
</tr>
<tr>
<td>txid_snapshot_xmin(txid_snapshot)</td>
<td>bigint</td>
<td>get xmin of snapshot</td>
</tr>
<tr>
<td>txid_snapshot_xmax(txid_snapshot)</td>
<td>bigint</td>
<td>get xmax of snapshot</td>
</tr>
<tr>
<td>txid_snapshot_xip(txid_snapshot)</td>
<td>setof bigint</td>
<td>get in-progress transaction IDs in snapshot</td>
</tr>
<tr>
<td>txid_visible_in_snapshot(txid, txid_snapshot)</td>
<td>boolean</td>
<td>is transaction ID visible in snapshot?</td>
</tr>
</tbody>
</table>

The internal transaction ID type (xid) is 32 bits wide and so it wraps around every 4 billion transactions. However, these functions export a 64-bit format that is extended with an “epoch” counter so that it will not wrap around for the life of an installation. The data type used by these functions, txid_snapshot, stores information about transaction ID visibility at a particular moment in time. Its components are described in Table 9-50.

### Table 9-50. Snapshot components

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmin</td>
<td>Earliest transaction ID (txid) that is still active. All earlier transactions will either be committed and visible, or rolled back and dead.</td>
</tr>
<tr>
<td>xmax</td>
<td>First as-yet-unassigned txid. All txids later than this one are not yet started as of the time of the snapshot, and thus invisible.</td>
</tr>
<tr>
<td>xip_list</td>
<td>Active txids at the time of the snapshot. All of them are between xmin and xmax. A txid that is xmin &lt;= txid &lt; xmax and not in this list was already completed at the time of the snapshot, and thus either visible or dead according to its commit status.</td>
</tr>
</tbody>
</table>

txid_snapshot’s textual representation is xmin:xmax:xip_list. For example 10:20:10,14,15 means xmin=10, xmax=20, xip_list=10, 14, 15.
9.23. System Administration Functions

Table 9-51 shows the functions available to query and alter run-time configuration parameters.

Table 9-51. Configuration Settings Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_setting(setting_name)</td>
<td>text</td>
<td>current value of setting</td>
</tr>
<tr>
<td>set_config(setting_name, new_value, is_local)</td>
<td>text</td>
<td>set parameter and return new value</td>
</tr>
</tbody>
</table>

The function `current_setting` yields the current value of the setting `setting_name`. It corresponds to the SQL command `SHOW`. An example:

SELECT current_setting('datestyle');

```
current_setting
--------------------
ISO, MDY
(1 row)
```

`set_config` sets the parameter `setting_name` to `new_value`. If `is_local` is true, the new value will only apply to the current transaction. If you want the new value to apply for the current session, use `false` instead. The function corresponds to the SQL command `SET`. An example:

SELECT set_config('log_statement_stats', 'off', false);

```
set_config
----------
off
(1 row)
```

The functions shown in Table 9-52 send control signals to other server processes. Use of these functions is restricted to superusers.

Table 9-52. Server Signalling Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_cancel_backend(pid int)</td>
<td>boolean</td>
<td>Cancel a backend’s current query</td>
</tr>
<tr>
<td>pg_reload_conf()</td>
<td>boolean</td>
<td>Cause server processes to reload their configuration files</td>
</tr>
<tr>
<td>pg_rotate_logfile()</td>
<td>boolean</td>
<td>Rotate server’s log file</td>
</tr>
</tbody>
</table>

Each of these functions returns `true` if successful and `false` otherwise.

`pg_cancel_backend` sends a query cancel (SIGINT) signal to a backend process identified by process ID. The process ID of an active backend can be found from the `procpid` column in the `pg_stat_activity` view, or by listing the `postgres` processes on the server with `ps`. 
pg_reload_conf sends a SIGHUP signal to the server, causing the configuration files to be reloaded by all server processes.

pg_rotate_logfile signals the log-file manager to switch to a new output file immediately. This works only when the built-in log collector is running, since otherwise there is no log-file manager subprocess.

The functions shown in Table 9-53 assist in making on-line backups. Use of the first three functions is restricted to superusers.

Table 9-53. Backup Control Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_start_backup(label text)</td>
<td>text</td>
<td>Set up for performing on-line backup</td>
</tr>
<tr>
<td>pg_stop_backup()</td>
<td>text</td>
<td>Finish performing on-line backup</td>
</tr>
<tr>
<td>pg_switch_xlog()</td>
<td>text</td>
<td>Force switch to a new transaction log file</td>
</tr>
<tr>
<td>pg_current_xlog_location()</td>
<td>text</td>
<td>Get current transaction log write location</td>
</tr>
<tr>
<td>pg_current_xlog_insert_location()</td>
<td>text</td>
<td>Get current transaction log insert location</td>
</tr>
<tr>
<td>pg_xlogfile_name_offset(location text)</td>
<td>text, integer</td>
<td>Convert transaction log location string to file name and decimal byte offset within file</td>
</tr>
<tr>
<td>pg_xlogfile_name(location text)</td>
<td>text</td>
<td>Convert transaction log location string to file name</td>
</tr>
</tbody>
</table>

pg_start_backup accepts a single parameter which is an arbitrary user-defined label for the backup. (Typically this would be the name under which the backup dump file will be stored.) The function writes a backup label file into the database cluster’s data directory, and then returns the backup’s starting transaction log location as text. The user need not pay any attention to this result value, but it is provided in case it is of use.

postgres=# select pg_start_backup('label_goes_here');
pg_start_backup
-----------------
0/D4445B8
(1 row)

pg_stop_backup removes the label file created by pg_start_backup, and instead creates a backup history file in the transaction log archive area. The history file includes the label given to pg_start_backup, the starting and ending transaction log locations for the backup, and the starting and ending times of the backup. The return value is the backup’s ending transaction log location (which again might be of little interest). After noting the ending location, the current transaction log insertion point is automatically advanced to the next transaction log file, so that the ending transaction log file can be archived immediately to complete the backup.
pg_switch_xlog moves to the next transaction log file, allowing the current file to be archived (assuming you are using continuous archiving). The result is the ending transaction log location + 1 within the just-completed transaction log file. If there has been no transaction log activity since the last transaction log switch, pg_switch_xlog does nothing and returns the start location of the transaction log file currently in use.

pg_current_xlog_location displays the current transaction log write location in the same format used by the above functions. Similarly, pg_current_xlog_insert_location displays the current transaction log insertion point. The insertion point is the “logical” end of the transaction log at any instant, while the write location is the end of what has actually been written out from the server’s internal buffers. The write location is the end of what can be examined from outside the server, and is usually what you want if you are interested in archiving partially-complete transaction log files. The insertion point is made available primarily for server debugging purposes. These are both read-only operations and do not require superuser permissions.

You can use pg_xlogfile_name_offset to extract the corresponding transaction log file name and byte offset from the results of any of the above functions. For example:

```sql
postgres=# select * from pg_xlogfile_name_offset(pg_stop_backup());

<table>
<thead>
<tr>
<th>file_name</th>
<th>file_offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001000000000000000D</td>
<td>4039624</td>
</tr>
</tbody>
</table>

(1 row)
```

Similarly, pg_xlogfile_name extracts just the transaction log file name. When the given transaction log location is exactly at a transaction log file boundary, both these functions return the name of the preceding transaction log file. This is usually the desired behavior for managing transaction log archiving behavior, since the preceding file is the last one that currently needs to be archived.

For details about proper usage of these functions, see Section 24.3.

The functions shown in Table 9-54 calculate the actual disk space usage of database objects.

### Table 9-54. Database Object Size Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_column_size(any)</td>
<td>int</td>
<td>Number of bytes used to store a particular value (possibly compressed)</td>
</tr>
<tr>
<td>pg_database_size(oid)</td>
<td>bigint</td>
<td>Disk space used by the database with the specified OID</td>
</tr>
<tr>
<td>pg_database_size(name)</td>
<td>bigint</td>
<td>Disk space used by the database with the specified name</td>
</tr>
<tr>
<td>pg_relation_size(oid)</td>
<td>bigint</td>
<td>Disk space used by the table or index with the specified OID</td>
</tr>
<tr>
<td>pg_relation_size(text)</td>
<td>bigint</td>
<td>Disk space used by the table or index with the specified name. The table name can be qualified with a schema name</td>
</tr>
<tr>
<td>pg_size_pretty(bigint)</td>
<td>text</td>
<td>Converts a size in bytes into a human-readable format with size units</td>
</tr>
</tbody>
</table>
Chapter 9. Functions and Operators

### Table 9-55. Generic File Access Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pg_ls_dir</code> (dirname text)</td>
<td>setof text</td>
<td>List the contents of a directory</td>
</tr>
<tr>
<td><code>pg_read_file</code> (filename text, offset bigint, length bigint)</td>
<td>text</td>
<td>Return the contents of a text file</td>
</tr>
<tr>
<td><code>pg_stat_file</code> (filename text)</td>
<td>record</td>
<td>Return information about a file</td>
</tr>
</tbody>
</table>

**pg_column_size** shows the space used to store any individual data value.

**pg_database_size** and **pg_tablespace_size** accept the OID or name of a database or tablespace, and return the total disk space used therein.

**pg_relation_size** accepts the OID or name of a table, index or toast table, and returns the size in bytes.

**pg_size_pretty** can be used to format the result of one of the other functions in a human-readable way, using kB, MB, GB or TB as appropriate.

**pg_total_relation_size** accepts the OID or name of a table or toast table, and returns the size in bytes of the data and all associated indexes and toast tables.

The functions shown in Table 9-55 provide native file access to files on the machine hosting the server. Only files within the database cluster directory and the `log_directory` can be accessed. Use a relative path for files within the cluster directory, and a path matching the `log_directory` configuration setting for log files. Use of these functions is restricted to superusers.
The functions shown in Table 9-56 manage advisory locks. For details about proper usage of these functions, see Section 13.3.4.

Table 9-56. Advisory Lock Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_advisory_lock(key bigint)</td>
<td>void</td>
<td>Obtain exclusive advisory lock</td>
</tr>
<tr>
<td>pg_advisory_lock(key1 int, key2 int)</td>
<td>void</td>
<td>Obtain exclusive advisory lock</td>
</tr>
<tr>
<td>pg_advisory_lock_shared(key bigint)</td>
<td>void</td>
<td>Obtain shared advisory lock</td>
</tr>
<tr>
<td>pg_advisory_lock_shared(key1 int, key2 int)</td>
<td>void</td>
<td>Obtain shared advisory lock</td>
</tr>
<tr>
<td>pg_try_advisory_lock(key bigint)</td>
<td>boolean</td>
<td>Obtain exclusive advisory lock if available</td>
</tr>
<tr>
<td>pg_try_advisory_lock(key1 int, key2 int)</td>
<td>boolean</td>
<td>Obtain exclusive advisory lock if available</td>
</tr>
<tr>
<td>pg_try_advisory_lock_shared(key bigint)</td>
<td>boolean</td>
<td>Obtain shared advisory lock if available</td>
</tr>
<tr>
<td>pg_try_advisory_lock_shared(key1 int, key2 int)</td>
<td>boolean</td>
<td>Obtain shared advisory lock if available</td>
</tr>
<tr>
<td>pg_advisory_unlock(key bigint)</td>
<td>boolean</td>
<td>Release an exclusive advisory lock</td>
</tr>
<tr>
<td>pg_advisory_unlock(key1 int, key2 int)</td>
<td>boolean</td>
<td>Release an exclusive advisory lock</td>
</tr>
<tr>
<td>pg_advisory_unlock_shared(key bigint)</td>
<td>boolean</td>
<td>Release a shared advisory lock</td>
</tr>
<tr>
<td>pg_advisory_unlock_shared(key1 int, key2 int)</td>
<td>boolean</td>
<td>Release a shared advisory lock</td>
</tr>
<tr>
<td>pg_advisory_unlock_all()</td>
<td>void</td>
<td>Release all advisory locks held by the current session</td>
</tr>
</tbody>
</table>

pg_advisory_lock locks an application-defined resource, which can be identified either by a single 64-bit key value or two 32-bit key values (note that these two key spaces do not overlap). If another session already holds a lock on the same resource, the function will wait until the resource becomes available. The lock is exclusive. Multiple lock requests stack, so that if the same resource is locked three times it must be also unlocked three times to be released for other sessions’ use.

pg_advisory_lock_shared works the same as pg_advisory_lock, except the lock can be
shared with other sessions requesting shared locks. Only would-be exclusive lockers are locked out.

`pg_try_advisory_lock` is similar to `pg_advisory_lock`, except the function will not wait for
the lock to become available. It will either obtain the lock immediately and return `true`, or return `false` if the lock cannot be acquired now.

`pg_try_advisory_lock_shared` works the same as `pg_try_advisory_lock`, except it attempts
to acquire shared rather than exclusive lock.

`pg_advisory_unlock` will release a previously-acquired exclusive advisory lock. It will return `true` if the lock is successfully released. If the lock was in fact not held, it will return `false`, and in addition, an SQL warning will be raised by the server.

`pg_advisory_unlock_shared` works the same as `pg_advisory_unlock`, except to release a
shared advisory lock.

`pg_advisory_unlock_all` will release all advisory locks held by the current session. (This func-
tion is implicitly invoked at session end, even if the client disconnects ungracefully.)
Chapter 10. Type Conversion

SQL statements can, intentionally or not, require mixing of different data types in the same expression. PostgreSQL has extensive facilities for evaluating mixed-type expressions.

In many cases a user will not need to understand the details of the type conversion mechanism. However, the implicit conversions done by PostgreSQL can affect the results of a query. When necessary, these results can be tailored by using explicit type conversion.

This chapter introduces the PostgreSQL type conversion mechanisms and conventions. Refer to the relevant sections in Chapter 8 and Chapter 9 for more information on specific data types and allowed functions and operators.

10.1. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. PostgreSQL has an extensible type system that is much more general and flexible than other SQL implementations. Hence, most type conversion behavior in PostgreSQL is governed by general rules rather than by ad hoc heuristics. This allows mixed-type expressions to be meaningful even with user-defined types.

The PostgreSQL scanner/parser divides lexical elements into only five fundamental categories: integers, non-integer numbers, strings, identifiers, and key words. Constants of most non-numeric types are first classified as strings. The SQL language definition allows specifying type names with strings, and this mechanism can be used in PostgreSQL to start the parser down the correct path. For example, the query

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";
```

<table>
<thead>
<tr>
<th>label</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin</td>
<td>(0,0)</td>
</tr>
</tbody>
</table>
(1 row)

has two literal constants, of type text and point. If a type is not specified for a string literal, then the placeholder type unknown is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the PostgreSQL parser:

Function calls

Much of the PostgreSQL type system is built around a rich set of functions. Functions can have one or more arguments. Since PostgreSQL permits function overloading, the function name alone does not uniquely identify the function to be called; the parser must select the right function based on the data types of the supplied arguments.

Operators

PostgreSQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators. Like functions, operators can be overloaded, and so the same problem of selecting the right operator exists.
Value Storage

SQL INSERT and UPDATE statements place the results of expressions into a table. The expressions in the statement must be matched up with, and perhaps converted to, the types of the target columns.

UNION, CASE, and related constructs

Since all query results from a unionized SELECT statement must appear in a single set of columns, the types of the results of each SELECT clause must be matched up and converted to a uniform set. Similarly, the result expressions of a CASE construct must be converted to a common type so that the CASE expression as a whole has a known output type. The same holds for ARRAY constructs, and for the GREATEST and LEAST functions.

The system catalogs store information about which conversions, called casts, between data types are valid, and how to perform those conversions. Additional casts can be added by the user with the CREATE CAST command. (This is usually done in conjunction with defining new data types. The set of casts between the built-in types has been carefully crafted and is best not altered.)

An additional heuristic is provided in the parser to allow better guesses at proper behavior for SQL standard types. There are several basic type categories defined: boolean, numeric, string, bitstring, datetime, timespan, geometric, network, and user-defined. Each category, with the exception of user-defined, has one or more preferred types which are preferentially selected when there is ambiguity. In the user-defined category, each type is its own preferred type. Ambiguous expressions (those with multiple candidate parsing solutions) can therefore often be resolved when there are multiple possible built-in types, but they will raise an error when there are multiple choices for user-defined types.

All type conversion rules are designed with several principles in mind:

- Implicit conversions should never have surprising or unpredictable outcomes.
- User-defined types, of which the parser has no a priori knowledge, should be “higher” in the type hierarchy. In mixed-type expressions, native types shall always be converted to a user-defined type (of course, only if conversion is necessary).
- User-defined types are not related. Currently, PostgreSQL does not have information available to it on relationships between types, other than hardcoded heuristics for built-in types and implicit relationships based on available functions and casts.
- There should be no extra overhead from the parser or executor if a query does not need implicit type conversion. That is, if a query is well formulated and the types already match up, then the query should proceed without spending extra time in the parser and without introducing unnecessary implicit conversion calls into the query.

Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines a new function with the correct argument types, the parser should use this new function and will no longer do the implicit conversion using the old function.

10.2. Operators

The specific operator to be used in an operator invocation is determined by following the procedure below. Note that this procedure is indirectly affected by the precedence of the involved operators. See
Chapter 10. Type Conversion

Section 4.1.6 for more information.

Operator Type Resolution

1. Select the operators to be considered from the pg_operator system catalog. If an unqualified operator name was used (the usual case), the operators considered are those of the right name and argument count that are visible in the current search path (see Section 5.7.3). If a qualified operator name was given, only operators in the specified schema are considered.

   a. If the search path finds multiple operators of identical argument types, only the one appearing earliest in the path is considered. But operators of different argument types are considered on an equal footing regardless of search path position.

2. Check for an operator accepting exactly the input argument types. If one exists (there can be only one exact match in the set of operators considered), use it.

   a. If one argument of a binary operator invocation is of the unknown type, then assume it is the same type as the other argument for this check. Other cases involving unknown will never find a match at this step.

3. Look for the best match.

   a. Discard candidate operators for which the input types do not match and cannot be converted (using an implicit conversion) to match. unknown literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.

   b. Run through all candidates and keep those with the most exact matches on input types. (Domains are considered the same as their base type for this purpose.) Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.

   c. Run through all candidates and keep those that accept preferred types (of the input data type’s type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

   d. If any input arguments are unknown, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the string category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal does look like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.

   e. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Some examples follow.
Chapter 10. Type Conversion

Example 10-1. Factorial Operator Type Resolution

There is only one factorial operator (postfix !) defined in the standard catalog, and it takes an argument of type bigint. The scanner assigns an initial type of integer to the argument in this query expression:

```
SELECT 40 ! AS "40 factorial";
```

```
40 factorial
------------------
8159152832478977343456112695961158942720000000000
```

So the parser does a type conversion on the operand and the query is equivalent to

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Example 10-2. String Concatenation Operator Type Resolution

A string-like syntax is used for working with string types as well as for working with complex extension types. Strings with unspecified type are matched with likely operator candidates.

An example with one unspecified argument:

```
SELECT text 'abc' || 'def' AS "text and unknown";
```

```
text and unknown
------------------
abcdef
```

In this case the parser looks to see if there is an operator taking text for both arguments. Since there is, it assumes that the second argument should be interpreted as of type text.

Here is a concatenation on unspecified types:

```
SELECT 'abc' || 'def' AS "unspecified";
```

```
unspecified
----------
abcdef
```

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected, and then the preferred type for strings, text, is used as the specific type to resolve the unknown literals to.

Example 10-3. Absolute-Value and Negation Operator Type Resolution

The PostgreSQL operator catalog has several entries for the prefix operator @, all of which implement absolute-value operations for various numeric data types. One of these entries is for type float8, which is the preferred type in the numeric category. Therefore, PostgreSQL will use that entry when faced with an unknown input:

```
SELECT @ '-4.5' AS "abs";
```
Here the system has implicitly resolved the unknown-type literal as type float8 before applying the chosen operator. We can verify that float8 and not some other type was used:

```
SELECT @ '-4.5e500' AS "abs";
```

ERROR: "-4.5e500" is out of range for type double precision

On the other hand, the prefix operator ~ (bitwise negation) is defined only for integer data types, not for float8. So, if we try a similar case with ~, we get:

```
SELECT ~ '20' AS "negation";
```

ERROR: operator is not unique: ~ "unknown"
HINT: Could not choose a best candidate operator. You might need to add explicit type casts.

This happens because the system cannot decide which of the several possible ~ operators should be preferred. We can help it out with an explicit cast:

```
SELECT ~ CAST('20' AS int8) AS "negation";
```

negation
----------
-21
(1 row)

10.3. Functions

The specific function to be used in a function invocation is determined according to the following steps.

**Function Type Resolution**

1. Select the functions to be considered from the `pg_proc` system catalog. If an unqualified function name was used, the functions considered are those of the right name and argument count that are visible in the current search path (see Section 5.7.3). If a qualified function name was given, only functions in the specified schema are considered.

   a. If the search path finds multiple functions of identical argument types, only the one appearing earliest in the path is considered. But functions of different argument types are considered on an equal footing regardless of search path position.

2. Check for a function accepting exactly the input argument types. If one exists (there can be only one exact match in the set of functions considered), use it. (Cases involving unknown will never find a match at this step.)

3. If no exact match is found, see whether the function call appears to be a special type conversion request. This happens if the function call has just one argument and the function name is the same as the (internal) name of some data type. Furthermore, the function argument must be either an unknown-type literal, or a type that is binary-compatible with the named data type, or a type that could be converted to the named data type by applying that type’s I/O functions (that is, the
conversion is either to or from one of the standard string types). When these conditions are met, the function call is treated as a form of CAST specification.  

4. Look for the best match.

   a. Discard candidate functions for which the input types do not match and cannot be converted (using an implicit conversion) to match. unknown literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.

   b. Run through all candidates and keep those with the most exact matches on input types. (Domains are considered the same as their base type for this purpose.) Keep all candidates if none have any exact matches. If only one candidate remains, use it; else continue to the next step.

   c. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

   d. If any input arguments are unknown, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the string category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal does look like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type at a given argument position, discard candidates that accept non-preferred types for that argument.

   e. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Note that the “best match” rules are identical for operator and function type resolution. Some examples follow.

Example 10-4. Rounding Function Argument Type Resolution

There is only one round function with two arguments. (The first is numeric, the second is integer.) So the following query automatically converts the first argument of type integer to numeric:

```
SELECT round(4, 4);
```

```
round
--------
 4.0000
(1 row)
```

That query is actually transformed by the parser to

```
SELECT round(CAST (4 AS numeric), 4);
```

Since numeric constants with decimal points are initially assigned the type numeric, the following query will require no type conversion and might therefore be slightly more efficient:

```
SELECT round(4.0, 4);
```

---

1. The reason for this step is to support function-style cast specifications in cases where there is not an actual cast function. If there is a cast function, it is conventionally named after its output type, and so there is no need to have a special case. See CREATE CAST for additional commentary.
Example 10-5. Substring Function Type Resolution

There are several `substr` functions, one of which takes types `text` and `integer`. If called with a string constant of unspecified type, the system chooses the candidate function that accepts an argument of the preferred category `string` (namely of type `text`).

```
SELECT substr('1234', 3);
```

```
substr
--------
34
(1 row)
```

If the string is declared to be of type `varchar`, as might be the case if it comes from a table, then the parser will try to convert it to become `text`:

```
SELECT substr(varchar '1234', 3);
```

```
substr
--------
34
(1 row)
```

This is transformed by the parser to effectively become

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

Note: The parser learns from the `pg_cast` catalog that `text` and `varchar` are binary-compatible, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no type conversion call is really inserted in this case.

And, if the function is called with an argument of type `integer`, the parser will try to convert that to `text`:

```
SELECT substr(1234, 3);
```

ERROR: function substr(integer, integer) does not exist
HINT: No function matches the given name and argument types. You might need to add explicit type casts.

This does not work because `integer` does not have an implicit cast to `text`. An explicit cast will work, however:

```
SELECT substr(CAST (1234 AS text), 3);
```

```
substr
--------
34
(1 row)
```
Chapter 10. Type Conversion

10.4. Value Storage

Values to be inserted into a table are converted to the destination column’s data type according to the following steps.

Value Storage Type Conversion

1. Check for an exact match with the target.
2. Otherwise, try to convert the expression to the target type. This will succeed if there is a registered cast between the two types. If the expression is an unknown-type literal, the contents of the literal string will be fed to the input conversion routine for the target type.
3. Check to see if there is a sizing cast for the target type. A sizing cast is a cast from that type to itself. If one is found in the `pg_cast` catalog, apply it to the expression before storing into the destination column. The implementation function for such a cast always takes an extra parameter of type `integer`, which receives the destination column’s `atttypmod` value (typically its declared length, although the interpretation of `atttypmod` varies for different data types), and it may take a third `boolean` parameter that says whether the cast is explicit or implicit. The cast function is responsible for applying any length-dependent semantics such as size checking or truncation.

Example 10-6. character Storage Type Conversion

For a target column declared as `character(20)` the following statement shows that the stored value is sized correctly:

```sql
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;
```

<table>
<thead>
<tr>
<th>v</th>
<th>octet_length</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcdef</td>
<td>20</td>
</tr>
</tbody>
</table>

What has really happened here is that the two unknown literals are resolved to `text` by default, allowing the `||` operator to be resolved as `text` concatenation. Then the `text` result of the operator is converted to `bpchar` (“blank-padded char”, the internal name of the `character` data type) to match the target column type. (Since the types `text` and `bpchar` are binary-compatible, this conversion does not insert any real function call.) Finally, the sizing function `bpchar(bpchar, integer, boolean)` is found in the system catalog and applied to the operator’s result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

10.5. UNION, CASE, and Related Constructs

SQL `UNION` constructs must match up possibly dissimilar types to become a single result set. The resolution algorithm is applied separately to each output column of a union query. The `INTERSECT` and `EXCEPT` constructs resolve dissimilar types in the same way as `UNION`. The `CASE`, `ARRAY`, `VALUES`, `GREATEST` and `LEAST` constructs use the identical algorithm to match up their component expressions and select a result data type.
Chapter 10. Type Conversion

Type Resolution for UNION, CASE, and Related Constructs

1. If all inputs are of the same type, and it is not unknown, resolve as that type. Otherwise, replace any domain types in the list with their underlying base types.

2. If all inputs are of type unknown, resolve as type text (the preferred type of the string category). Otherwise, the unknown inputs will be ignored.

3. If the non-unknown inputs are not all of the same type category, fail.

4. Choose the first non-unknown input type which is a preferred type in that category, if there is one.

5. Otherwise, choose the last non-unknown input type that allows all the preceding non-unknown inputs to be implicitly converted to it. (There always is such a type, since at least the first type in the list must satisfy this condition.)

6. Convert all inputs to the selected type. Fail if there is not a conversion from a given input to the selected type.

Some examples follow.

Example 10-7. Type Resolution with Underspecified Types in a Union

```sql
SELECT text 'a' AS "text" UNION SELECT 'b';
```

```
text
------
a
b
```

Here, the unknown-type literal 'b' will be resolved as type text.

Example 10-8. Type Resolution in a Simple Union

```sql
SELECT 1.2 AS "numeric" UNION SELECT 1;
```

```
numeric
--------
1
1.2
```

The literal 1.2 is of type numeric, and the integer value 1 can be cast implicitly to numeric, so that type is used.

Example 10-9. Type Resolution in a Transposed Union

```sql
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
```

```
real
-----
1
2.2
```
(2 rows)
Here, since type real cannot be implicitly cast to integer, but integer can be implicitly cast to real, the union result type is resolved as real.
Chapter 11. Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

11.1. Introduction

Suppose we have a table similar to this:

```sql
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

and the application requires a lot of queries of the form:

```sql
SELECT content FROM test1 WHERE id = constant;
```

With no advance preparation, the system would have to scan the entire `test1` table, row by row, to find all matching entries. If there are a lot of rows in `test1` and only a few rows (perhaps only zero or one) that would be returned by such a query, then this is clearly an inefficient method. But if the system has been instructed to maintain an index on the `id` column, then it can use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.

A similar approach is used in most books of non-fiction: terms and concepts that are frequently looked up by readers are collected in an alphabetic index at the end of the book. The interested reader can scan the index relatively quickly and flip to the appropriate page(s), rather than having to read the entire book to find the material of interest. Just as it is the task of the author to anticipate the items that the readers are likely to look up, it is the task of the database programmer to foresee which indexes will be of advantage.

The following command would be used to create the index on the `id` column, as discussed:

```sql
CREATE INDEX test1_id_index ON test1 (id);
```

The name `test1_id_index` can be chosen freely, but you should pick something that enables you to remember later what the index was for.

To remove an index, use the `DROP INDEX` command. Indexes can be added to and removed from tables at any time.

Once an index is created, no further intervention is required: the system will update the index when the table is modified, and it will use the index in queries when it thinks this would be more efficient than a sequential table scan. But you might have to run the `ANALYZE` command regularly to update statistics to allow the query planner to make educated decisions. See Chapter 14 for information about how to find out whether an index is used and when and why the planner might choose not to use an index.

Indexes can also benefit `UPDATE` and `DELETE` commands with search conditions. Indexes can moreover be used in join searches. Thus, an index defined on a column that is part of a join condition can significantly speed up queries with joins.
Creating an index on a large table can take a long time. By default, PostgreSQL allows reads (selects) to occur on the table in parallel with creation of an index, but writes (inserts, updates, deletes) are blocked until the index build is finished. In production environments this is often unacceptable. It is possible to allow writes to occur in parallel with index creation, but there are several caveats to be aware of — for more information see Building Indexes Concurrently.

After an index is created, the system has to keep it synchronized with the table. This adds overhead to data manipulation operations. Therefore indexes that are seldom or never used in queries should be removed.

11.2. Index Types

PostgreSQL provides several index types: B-tree, Hash, GiST and GIN. Each index type uses a different algorithm that is best suited to different types of queries. By default, the `CREATE INDEX` command will create a B-tree index, which fits the most common situations.

B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:

- `<`
- `<=`
- `=`
- `>=`
- `>`

Constructs equivalent to combinations of these operators, such as `BETWEEN` and `IN`, can also be implemented with a B-tree index search. Also, an `IS NULL` condition on an index column can be used with a B-tree index.

The optimizer can also use a B-tree index for queries involving the pattern matching operators `LIKE` and `~` if the pattern is a constant and is anchored to the beginning of the string — for example, `col LIKE 'foo%'` or `col ~ '^foo'`, but not `col LIKE '%bar'`. However, if your server does not use the C locale you will need to create the index with a special operator class to support indexing of pattern-matching queries. See Section 11.9 below. It is also possible to use B-tree indexes for `ILIKE` and `~*`, but only if the pattern starts with non-alphabetic characters, i.e. characters that are not affected by upper/lower case conversion.

Hash indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the `=` operator. (But hash indexes do not support `IS NULL` searches.) The following command is used to create a hash index:

```
CREATE INDEX name ON table USING hash (column);
```

**Note**: Testing has shown PostgreSQL’s hash indexes to perform no better than B-tree indexes, and the index size and build time for hash indexes is much worse. Furthermore, hash index operations are not presently WAL-logged, so hash indexes might need to be rebuilt with `REINDEX` after a database crash. For these reasons, hash index use is presently discouraged.

GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST
index can be used vary depending on the indexing strategy (the *operator class*). As an example, the standard distribution of PostgreSQL includes GiST operator classes for several two-dimensional geometric data types, which support indexed queries using these operators:

```
<< & << & >> <<| & <<| | & |> | >> 0> <0 ~= &&
```

(See Section 9.11 for the meaning of these operators.) Many other GiST operator classes are available in the `contrib` collection or as separate projects. For more information see Chapter 51.

GIN indexes are inverted indexes which can handle values that contain more than one key, arrays for example. Like GiST, GIN can support many different user-defined indexing strategies and the particular operators with which a GIN index can be used vary depending on the indexing strategy. As an example, the standard distribution of PostgreSQL includes GIN operator classes for one-dimensional arrays, which support indexed queries using these operators:

```
<0 0> = &&
```

(See Section 9.17 for the meaning of these operators.) Many other GIN operator classes are available in the `contrib` collection or as separate projects. For more information see Chapter 52.

### 11.3. Multicolumn Indexes

An index can be defined on more than one column of a table. For example, if you have a table of this form:

```sql
CREATE TABLE test2 (  
    major int,  
    minor int,  
    name varchar
);
```

(say, you keep your `/dev` directory in a database...) and you frequently make queries like:

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

then it might be appropriate to define an index on the columns `major` and `minor` together, e.g.:

```sql
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```
Currently, only the B-tree and GiST index types support multicolumn indexes. Up to 32 columns can be specified. (This limit can be altered when building PostgreSQL; see the file pg_config_manual.h.)

A multicolumn B-tree index can be used with query conditions that involve any subset of the index’s columns, but the index is most efficient when there are constraints on the leading (leftmost) columns. The exact rule is that equality constraints on leading columns, plus any inequality constraints on the first column that does not have an equality constraint, will be used to limit the portion of the index that is scanned. Constraints on columns to the right of these columns are checked in the index, so they save visits to the table proper, but they do not reduce the portion of the index that has to be scanned. For example, given an index on \((a, b, c)\) and a query condition \(\text{WHERE } a = 5 \text{ AND } b >= 42 \text{ AND } c < 77\), the index would have to be scanned from the first entry with \(a = 5\) and \(b = 42\) up through the last entry with \(a = 5\). Index entries with \(c >= 77\) would be skipped, but they’d still have to be scanned through. This index could in principle be used for queries that have constraints on \(b\) and/or \(c\) with no constraint on \(a\) — but the entire index would have to be scanned, so in most cases the planner would prefer a sequential table scan over using the index.

A multicolumn GiST index can be used with query conditions that involve any subset of the index’s columns. Conditions on additional columns restrict the entries returned by the index, but the condition on the first column is the most important one for determining how much of the index needs to be scanned. A GiST index will be relatively ineffective if its first column has only a few distinct values, even if there are many distinct values in additional columns.

Of course, each column must be used with operators appropriate to the index type; clauses that involve other operators will not be considered.

Multicolumn indexes should be used sparingly. In most situations, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are unlikely to be helpful unless the usage of the table is extremely stylized. See also Section 11.5 for some discussion of the merits of different index setups.

11.4. Indexes and ORDER BY

In addition to simply finding the rows to be returned by a query, an index may be able to deliver them in a specific sorted order. This allows a query’s ORDER BY specification to be met without a separate sorting step. Of the index types currently supported by PostgreSQL, only B-tree can produce sorted output — the other index types return matching rows in an unspecified, implementation-dependent order.

The planner will consider satisfying an ORDER BY specification either by scanning any available index that matches the specification, or by scanning the table in physical order and doing an explicit sort. For a query that requires scanning a large fraction of the table, the explicit sort is likely to be faster because it requires less disk I/O due to a better-ordered access pattern. Indexes are more useful when only a few rows need be fetched. An important special case is ORDER BY in combination with LIMIT \(n\): an explicit sort will have to process all the data to identify the first \(n\) rows, but if there is an index matching the ORDER BY then the first \(n\) rows can be retrieved directly, without scanning the remainder at all.

By default, B-tree indexes store their entries in ascending order with nulls last. This means that a forward scan of an index on a column \(x\) produces output satisfying ORDER BY \(x\) (or more verbosely, ORDER BY \(x\) ASC NULLS LAST). The index can also be scanned backward, producing output satisfying ORDER BY \(x\) DESC (or more verbosely, ORDER BY \(x\) DESC NULLS FIRST, since NULLS FIRST is the default for ORDER BY DESC).
You can adjust the ordering of a B-tree index by including the options `ASC`, `DESC`, `NULLS FIRST`, and/or `NULLS LAST` when creating the index; for example:

```sql
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

An index stored in ascending order with nulls first can satisfy either `ORDER BY x ASC NULLS FIRST` or `ORDER BY x DESC NULLS LAST` depending on which direction it is scanned in.

You might wonder why bother providing all four options, when two options together with the possibility of backward scan would cover all the variants of `ORDER BY`. In single-column indexes the options are indeed redundant, but in multicolumn indexes they can be useful. Consider a two-column index on `(x, y)`: this can satisfy `ORDER BY x, y` if we scan forward, or `ORDER BY x DESC, y DESC` if we scan backward. But it might be that the application frequently needs to use `ORDER BY x ASC, y DESC`. There is no way to get that ordering from a regular index, but it is possible if the index is defined as `(x ASC, y DESC)` or `(x DESC, y ASC).

Obviously, indexes with non-default sort orderings are a fairly specialized feature, but sometimes they can produce tremendous speedups for certain queries. Whether it’s worth keeping such an index depends on how often you use queries that require a special sort ordering.

### 11.5. Combining Multiple Indexes

A single index scan can only use query clauses that use the index’s columns with operators of its operator class and are joined with `AND`. For example, given an index on `(a, b)` a query condition like `WHERE a = 5 AND b = 6` could use the index, but a query like `WHERE a = 5 OR b = 6` could not directly use the index.

Beginning in release 8.1, PostgreSQL has the ability to combine multiple indexes (including multiple uses of the same index) to handle cases that cannot be implemented by single index scans. The system can form `AND` and `OR` conditions across several index scans. For example, a query like `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` could be broken down into four separate scans of an index on `x`, each scan using one of the query clauses. The results of these scans are then ORed together to produce the result. Another example is that if we have separate indexes on `x` and `y`, one possible implementation of a query like `WHERE x = 5 AND y = 6` is to use each index with the appropriate query clause and then `AND` together the index results to identify the result rows.

To combine multiple indexes, the system scans each needed index and prepares a `bitmap` in memory giving the locations of table rows that are reported as matching that index’s conditions. The bitmaps are then `AND`ed and `OR`ed together as needed by the query. Finally, the actual table rows are visited and returned. The table rows are visited in physical order, because that is how the bitmap is laid out; this means that any ordering of the original indexes is lost, and so a separate sort step will be needed if the query has an `ORDER BY` clause. For this reason, and because each additional index scan adds extra time, the planner will sometimes choose to use a simple index scan even though additional indexes are available that could have been used as well.

In all but the simplest applications, there are various combinations of indexes that might be useful, and the database developer must make trade-offs to decide which indexes to provide. Sometimes multicolumn indexes are best, but sometimes it’s better to create separate indexes and rely on the index-combination feature. For example, if your workload includes a mix of queries that sometimes involve only column `x`, sometimes only column `y`, and sometimes both columns, you might choose to create two separate indexes on `x` and `y`, relying on index combination to process the queries that use both columns. You could also create a multicolumn index on `(x, y)`. This index would typically be more efficient than index combination for queries involving both columns, but as discussed in Section...
11.3. it would be almost useless for queries involving only y, so it could not be the only index. A combination of the multicolumn index and a separate index on y would serve reasonably well. For queries involving only x, the multicolumn index could be used, though it would be larger and hence slower than an index on x alone. The last alternative is to create all three indexes, but this is probably only reasonable if the table is searched much more often than it is updated and all three types of query are common. If one of the types of query is much less common than the others, you’d probably settle for creating just the two indexes that best match the common types.

11.6. Unique Indexes

Indexes can also be used to enforce uniqueness of a column’s value, or the uniqueness of the combined values of more than one column.

```sql
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

Currently, only B-tree indexes can be declared unique.

When an index is declared unique, multiple table rows with equal indexed values will not be allowed. Null values are not considered equal. A multicolumn unique index will only reject cases where all of the indexed columns are equal in two rows.

PostgreSQL automatically creates a unique index when a unique constraint or a primary key is defined for a table. The index covers the columns that make up the primary key or unique columns (a multicolumn index, if appropriate), and is the mechanism that enforces the constraint.

**Note:** The preferred way to add a unique constraint to a table is `ALTER TABLE ... ADD CONSTRAINT`. The use of indexes to enforce unique constraints could be considered an implementation detail that should not be accessed directly. One should, however, be aware that there's no need to manually create indexes on unique columns; doing so would just duplicate the automatically-created index.

11.7. Indexes on Expressions

An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table. This feature is useful to obtain fast access to tables based on the results of computations.

For example, a common way to do case-insensitive comparisons is to use the `lower` function:

```sql
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

This query can use an index, if one has been defined on the result of the `lower(col1)` operation:

```sql
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

If we were to declare this index `UNIQUE`, it would prevent creation of rows whose `col1` values differ only in case, as well as rows whose `col1` values are actually identical. Thus, indexes on expressions can be used to enforce constraints that are not definable as simple unique constraints.

As another example, if one often does queries like this:
Chapter 11. Indexes

SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
then it might be worth creating an index like this:

CREATE INDEX people_names ON people ((first_name || ' ' || last_name));

The syntax of the CREATE INDEX command normally requires writing parentheses around index expressions, as shown in the second example. The parentheses can be omitted when the expression is just a function call, as in the first example.

Index expressions are relatively expensive to maintain, because the derived expression(s) must be computed for each row upon insertion and whenever it is updated. However, the index expressions are not recomputed during an indexed search, since they are already stored in the index. In both examples above, the system sees the query as just WHERE indexedcolumn = 'constant’ and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

11.8. Partial Indexes

A partial index is an index built over a subset of a table; the subset is defined by a conditional expression (called the predicate of the partial index). The index contains entries for only those table rows that satisfy the predicate. Partial indexes are a specialized feature, but there are several situations in which they are useful.

One major reason for using a partial index is to avoid indexing common values. Since a query searching for a common value (one that accounts for more than a few percent of all the table rows) will not use the index anyway, there is no point in keeping those rows in the index at all. This reduces the size of the index, which will speed up queries that do use the index. It will also speed up many table update operations because the index does not need to be updated in all cases. Example 11-1 shows a possible application of this idea.

Example 11-1. Setting up a Partial Index to Exclude Common Values

Suppose you are storing web server access logs in a database. Most accesses originate from the IP address range of your organization but some are from elsewhere (say, employees on dial-up connections). If your searches by IP are primarily for outside accesses, you probably do not need to index the IP range that corresponds to your organization’s subnet.

Assume a table like this:

```sql
CREATE TABLE access_log {
    url varchar,
    client_ip inet,
    ...
};
```

To create a partial index that suits our example, use a command such as this:

```sql
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet '192.168.100.255');
```

A typical query that can use this index would be:

```sql
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

A query that cannot use this index is:
Chapter 11. Indexes

SELECT * FROM access_log WHERE client_ip = inet '192.168.100.23';

Observe that this kind of partial index requires that the common values be predetermined. If the distribution of values is inherent (due to the nature of the application) and static (not changing over time), this is not difficult, but if the common values are merely due to the coincidental data load this can require a lot of maintenance work to change the index definition from time to time.

Another possible use for a partial index is to exclude values from the index that the typical query workload is not interested in; this is shown in Example 11-2. This results in the same advantages as listed above, but it prevents the “uninteresting” values from being accessed via that index at all, even if an index scan might be profitable in that case. Obviously, setting up partial indexes for this kind of scenario will require a lot of care and experimentation.

Example 11-2. Setting up a Partial Index to Exclude Uninteresting Values

If you have a table that contains both billed and unbilled orders, where the unbilled orders take up a small fraction of the total table and yet those are the most-accessed rows, you can improve performance by creating an index on just the unbilled rows. The command to create the index would look like this:

CREATE INDEX orders_unbilled_index ON orders (order_nr)
WHERE billed is not true;

A possible query to use this index would be:

SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;

However, the index can also be used in queries that do not involve order_nr at all, e.g.:

SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;

This is not as efficient as a partial index on the amount column would be, since the system has to scan the entire index. Yet, if there are relatively few unbilled orders, using this partial index just to find the unbilled orders could be a win.

Note that this query cannot use this index:

SELECT * FROM orders WHERE order_nr = 3501;

The order 3501 might be among the billed or among the unbilled orders.

Example 11-2 also illustrates that the indexed column and the column used in the predicate do not need to match. PostgreSQL supports partial indexes with arbitrary predicates, so long as only columns of the table being indexed are involved. However, keep in mind that the predicate must match the conditions used in the queries that are supposed to benefit from the index. To be precise, a partial index can be used in a query only if the system can recognize that the WHERE condition of the query mathematically implies the predicate of the index. PostgreSQL does not have a sophisticated theorem prover that can recognize mathematically equivalent expressions that are written in different forms. (Not only is such a general theorem prover extremely difficult to create, it would probably be too slow to be of any real use.) The system can recognize simple inequality implications, for example “x < 1” implies “x < 2”; otherwise the predicate condition must exactly match part of the query’s WHERE condition or the index will not be recognized to be usable. Matching takes place at query planning time, not at run time. As a result, parameterized query clauses will not work with a partial index. For example a prepared query with a parameter might specify “x < ?” which will never imply “x < 2” for all possible values of the parameter.
Chapter 11. Indexes

A third possible use for partial indexes does not require the index to be used in queries at all. The idea here is to create a unique index over a subset of a table, as in Example 11-3. This enforces uniqueness among the rows that satisfy the index predicate, without constraining those that do not.

Example 11-3. Setting up a Partial Unique Index

Suppose that we have a table describing test outcomes. We wish to ensure that there is only one “successful” entry for a given subject and target combination, but there might be any number of “unsuccessful” entries. Here is one way to do it:

```sql
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target) WHERE success;
```

This is a particularly efficient way of doing it when there are few successful tests and many unsuccessful ones.

Finally, a partial index can also be used to override the system’s query plan choices. Also, data sets with peculiar distributions might cause the system to use an index when it really should not. In that case the index can be set up so that it is not available for the offending query. Normally, PostgreSQL makes reasonable choices about index usage (e.g., it avoids them when retrieving common values, so the earlier example really only saves index size, it is not required to avoid index usage), and grossly incorrect plan choices are cause for a bug report.

Keep in mind that setting up a partial index indicates that you know at least as much as the query planner knows, in particular you know when an index might be profitable. Forming this knowledge requires experience and understanding of how indexes in PostgreSQL work. In most cases, the advantage of a partial index over a regular index will not be much.

More information about partial indexes can be found in The case for partial indexes, Partial indexing in POSTGRES: research project, and Generalized Partial Indexes (cached version).

11.9. Operator Classes and Operator Families

An index definition can specify an operator class for each column of an index.

```
CREATE INDEX name ON table (column opclass [sort options] [, ...]);
```

The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on the type int4 would use the int4_ops class; this operator class includes comparison functions for values of type int4. In practice the default operator class for the column’s data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful index behavior. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. The operator class determines the basic sort ordering (which can then be modified by adding sort options ASC/DESC and/or NULLS FIRST/NULLS LAST).
There are also some built-in operator classes besides the default ones:

- The operator classes `text_pattern_ops`, `varchar_pattern_ops`, `bpchar_pattern_ops`, and `name_pattern_ops` support B-tree indexes on the types `text`, `varchar`, `char`, and `name`, respectively. The difference from the default operator classes is that the values are compared strictly character by character rather than according to the locale-specific collation rules. This makes these operator classes suitable for use by queries involving pattern matching expressions (LIKE or POSIX regular expressions) when the server does not use the standard “C” locale. As an example, you might index a `varchar` column like this:

  ```sql
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Note that you should also create an index with the default operator class if you want queries involving ordinary comparisons to use an index. Such queries cannot use the `xxx_pattern_ops` operator classes. It is allowed to create multiple indexes on the same column with different operator classes. If you do use the C locale, you do not need the `xxx_pattern_ops` operator classes, because an index with the default operator class is usable for pattern-matching queries in the C locale.

The following query shows all defined operator classes:

```sql
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name
FROM pg_am am, pg_opclass opc
WHERE opc.opcmethod = am.oid
ORDER BY index_method, opclass_name;
```

An operator class is actually just a subset of a larger structure called an operator family. In cases where several data types have similar behaviors, it is frequently useful to define cross-data-type operators and allow these to work with indexes. To do this, the operator classes for each of the types must be grouped into the same operator family. The cross-type operators are members of the family, but are not associated with any single class within the family.

This query shows all defined operator families and all the operators included in each family:

```sql
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
FROM pg_am am, pg_opfamily opf, pg_amop amop
WHERE opf.opfmethod = am.oid AND
      amop.amopfamily = opf.oid
ORDER BY index_method, opfamily_name, opfamily_operator;
```

### 11.10. Examining Index Usage

Although indexes in PostgreSQL do not need maintenance and tuning, it is still important to check which indexes are actually used by the real-life query workload. Examining index usage for an individual query is done with the `EXPLAIN` command; its application for this purpose is illustrated in Section 14.1. It is also possible to gather overall statistics about index usage in a running server, as described in Section 26.2.
Chapter 11. Indexes

It is difficult to formulate a general procedure for determining which indexes to set up. There are a number of typical cases that have been shown in the examples throughout the previous sections. A good deal of experimentation will be necessary in most cases. The rest of this section gives some tips for that.

- Always run `ANALYZE` first. This command collects statistics about the distribution of the values in the table. This information is required to guess the number of rows returned by a query, which is needed by the planner to assign realistic costs to each possible query plan. In absence of any real statistics, some default values are assumed, which are almost certain to be inaccurate. Examining an application’s index usage without having run `ANALYZE` is therefore a lost cause.

- Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.

  It is especially fatal to use very small test data sets. While selecting 1000 out of 100000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows will probably fit within a single disk page, and there is no plan that can beat sequentially fetching 1 disk page.

  Also be careful when making up test data, which is often unavoidable when the application is not in production use yet. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.

- When indexes are not used, it can be useful for testing to force their use. There are run-time parameters that can turn off various plan types (see Section 18.6.1). For instance, turning off sequential scans (`enable_seqscan`) and nested-loop joins (`enable_nestloop`), which are the most basic plans, will force the system to use a different plan. If the system still chooses a sequential scan or nested-loop join then there is probably a more fundamental reason why the index is not used; for example, the query condition does not match the index. (What kind of query can use what kind of index is explained in the previous sections.)

- If forcing index usage does use the index, then there are two possibilities: Either the system is right and using the index is indeed not appropriate, or the cost estimates of the query plans are not reflecting reality. So you should time your query with and without indexes. The `EXPLAIN ANALYZE` command can be useful here.

- If it turns out that the cost estimates are wrong, there are, again, two possibilities. The total cost is computed from the per-row costs of each plan node times the selectivity estimate of the plan node. The costs estimated for the plan nodes can be adjusted via run-time parameters (described in Section 18.6.2). An inaccurate selectivity estimate is due to insufficient statistics. It might be possible to improve this by tuning the statistics-gathering parameters (see `ALTER TABLE`).

  If you do not succeed in adjusting the costs to be more appropriate, then you might have to resort to forcing index usage explicitly. You might also want to contact the PostgreSQL developers to examine the issue.
Chapter 12. Full Text Search

12.1. Introduction

Full Text Searching (or just text search) provides the capability to identify natural-language documents that satisfy a query, and optionally to sort them by relevance to the query. The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query. Notions of query and similarity are very flexible and depend on the specific application. The simplest search considers query as a set of words and similarity as the frequency of query words in the document.

Textual search operators have existed in databases for years. PostgreSQL has ~, ~*, LIKE, and ILIKE operators for textual data types, but they lack many essential properties required by modern information systems:

• There is no linguistic support, even for English. Regular expressions are not sufficient because they cannot easily handle derived words, e.g., satisfies and satisfy. You might miss documents that contain satisfies, although you probably would like to find them when searching for satisfy. It is possible to use OR to search for multiple derived forms, but this is tedious and error-prone (some words can have several thousand derivatives).
• They provide no ordering (ranking) of search results, which makes them ineffective when thousands of matching documents are found.
• They tend to be slow because there is no index support, so they must process all documents for every search.

Full text indexing allows documents to be preprocessed and an index saved for later rapid searching. Preprocessing includes:

 Parsing documents into tokens. It is useful to identify various classes of tokens, e.g. numbers, words, complex words, email addresses, so that they can be processed differently. In principle token classes depend on the specific application, but for most purposes it is adequate to use a predefined set of classes. PostgreSQL uses a parser to perform this step. A standard parser is provided, and custom parsers can be created for specific needs.

 Converting tokens into lexemes. A lexeme is a string, just like a token, but it has been normalized so that different forms of the same word are made alike. For example, normalization almost always includes folding upper-case letters to lower-case, and often involves removal of suffixes (such as s or es in English). This allows searches to find variant forms of the same word, without tediously entering all the possible variants. Also, this step typically eliminates stop words, which are words that are so common that they are useless for searching. (In short, then, tokens are raw fragments of the document text, while lexemes are words that are believed useful for indexing and searching.) PostgreSQL uses dictionaries to perform this step. Various standard dictionaries are provided, and custom ones can be created for specific needs.

 Storing preprocessed documents optimized for searching. For example, each document can be represented as a sorted array of normalized lexemes. Along with the lexemes it is often desirable to store positional information to use for proximity ranking, so that a document that contains a more “dense” region of query words is assigned a higher rank than one with scattered query words.

Dictionaries allow fine-grained control over how tokens are normalized. With appropriate dictionaries, you can:
- Define stop words that should not be indexed.
- Map synonyms to a single word using Ispell.
- Map phrases to a single word using a thesaurus.
- Map different variations of a word to a canonical form using an Ispell dictionary.
- Map different variations of a word to a canonical form using Snowball stemmer rules.

A data type `tsvector` is provided for storing preprocessed documents, along with a type `tsquery` for representing processed queries (Section 8.11). There are many functions and operators available for these data types (Section 9.13), the most important of which is the match operator `@@`, which we introduce in Section 12.1.2. Full text searches can be accelerated using indexes (Section 12.9).

### 12.1.1. What Is a Document?

A *document* is the unit of searching in a full text search system; for example, a magazine article or email message. The text search engine must be able to parse documents and store associations of lexemes (key words) with their parent document. Later, these associations are used to search for documents that contain query words.

For searches within PostgreSQL, a document is normally a textual field within a row of a database table, or possibly a combination (concatenation) of such fields, perhaps stored in several tables or obtained dynamically. In other words, a document can be constructed from different parts for indexing and it might not be stored anywhere as a whole. For example:

```sql
SELECT title || ' ' || author || ' ' || abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

**Note:** Actually, in these example queries, `coalesce` should be used to prevent a single `NULL` attribute from causing a `NULL` result for the whole document.

Another possibility is to store the documents as simple text files in the file system. In this case, the database can be used to store the full text index and to execute searches, and some unique identifier can be used to retrieve the document from the file system. However, retrieving files from outside the database requires superuser permissions or special function support, so this is usually less convenient than keeping all the data inside PostgreSQL. Also, keeping everything inside the database allows easy access to document metadata to assist in indexing and display.

For text search purposes, each document must be reduced to the preprocessed `tsvector` format. Searching and ranking are performed entirely on the `tsvector` representation of a document — the original text need only be retrieved when the document has been selected for display to a user. We therefore often speak of the `tsvector` as being the document, but of course it is only a compact representation of the full document.
12.1.2. Basic Text Matching

Full text searching in PostgreSQL is based on the match operator @@, which returns true if a \texttt{tsvector} (document) matches a \texttt{tsquery} (query). It doesn’t matter which data type is written first:

```
SELECT 'a fat cat sat on a mat and ate a fat rat':tsvector @@ 'cat & rat':tsquery;
```

```
?column?
----------
t
```

```
SELECT 'fat & cow':tsquery @@ 'a fat cat sat on a mat and ate a fat rat':tsvector;
```

```
?column?
----------
f
```

As the above example suggests, a \texttt{tsquery} is not just raw text, any more than a \texttt{tsvector} is. A \texttt{tsquery} contains search terms, which must be already-normalized lexemes, and may combine multiple terms using AND, OR, and NOT operators. (For details see Section 8.11.) There are functions \texttt{to\_tsquery} and \texttt{plainto\_tsquery} that are helpful in converting user-written text into a proper \texttt{tsquery}, for example by normalizing words appearing in the text. Similarly, \texttt{to\_tsvector} is used to parse and normalize a document string. So in practice a text search match would look more like this:

```
SELECT to\_tsvector('fat cats ate fat rats') @@ to\_tsquery('fat & rat');
```

```
?column?
----------
t
```

Observe that this match would not succeed if written as

```
SELECT 'fat cats ate fat rats':tsvector @@ to\_tsquery('fat & rat');
```

```
?column?
----------
f
```

since here no normalization of the word \texttt{rats} will occur. The elements of a \texttt{tsvector} are lexemes, which are assumed already normalized, so \texttt{rats} does not match \texttt{rat}.

The @@ operator also supports \texttt{text} input, allowing explicit conversion of a text string to \texttt{tsvector} or \texttt{tsquery} to be skipped in simple cases. The variants available are:

\texttt{tsvector @@ tsquery}
\texttt{tsquery @@ tsvector}
\texttt{text @@ tsquery}
\texttt{text @@ text}

The first two of these we saw already. The form \texttt{text @@ tsquery} is equivalent to \texttt{to\_tsvector(x) @@ y}. The form \texttt{text @@ text} is equivalent to \texttt{to\_tsvector(x) @@ plainto\_tsquery(y)}.
Chapter 12. Full Text Search

12.1.3. Configurations

The above are all simple text search examples. As mentioned before, full text search functionality includes the ability to do many more things: skip indexing certain words (stop words), process synonyms, and use sophisticated parsing, e.g. parse based on more than just white space. This functionality is controlled by text search configurations. PostgreSQL comes with predefined configurations for many languages, and you can easily create your own configurations. (psql’s \dF command shows all available configurations.)

During installation an appropriate configuration is selected and default_text_search_config is set accordingly in postgresql.conf. If you are using the same text search configuration for the entire cluster you can use the value in postgresql.conf. To use different configurations throughout the cluster but the same configuration within any one database, use ALTER DATABASE ... SET. Otherwise, you can set default_text_search_config in each session.

Each text search function that depends on a configuration has an optional regconfig argument, so that the configuration to use can be specified explicitly. default_text_search_config is used only when this argument is omitted.

To make it easier to build custom text search configurations, a configuration is built up from simpler database objects. PostgreSQL’s text search facility provides four types of configuration-related database objects:

- **Text search parsers** break documents into tokens and classify each token (for example, as words or numbers).
- **Text search dictionaries** convert tokens to normalized form and reject stop words.
- **Text search templates** provide the functions underlying dictionaries. (A dictionary simply specifies a template and a set of parameters for the template.)
- **Text search configurations** select a parser and a set of dictionaries to use to normalize the tokens produced by the parser.

Text search parsers and templates are built from low-level C functions; therefore it requires C programming ability to develop new ones, and superuser privileges to install one into a database. (There are examples of add-on parsers and templates in the contrib/ area of the PostgreSQL distribution.) Since dictionaries and configurations just parameterize and connect together some underlying parsers and templates, no special privilege is needed to create a new dictionary or configuration. Examples of creating custom dictionaries and configurations appear later in this chapter.

12.2. Tables and Indexes

The examples in the previous section illustrated full text matching using simple constant strings. This section shows how to search table data, optionally using indexes.

12.2.1. Searching a Table

It is possible to do full text search with no index. A simple query to print the title of each row that contains the word friend in its body field is:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```
This will also find related words such as friends and friendly, since all these are reduced to the same normalized lexeme.

The query above specifies that the english configuration is to be used to parse and normalize the strings. Alternatively we could omit the configuration parameters:

```sql
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

This query will use the configuration set by default_text_search_config.

A more complex example is to select the ten most recent documents that contain create and table in the title or body:

```sql
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC LIMIT 10;
```

For clarity we omitted the coalesce function calls which would be needed to find rows that contain NULL in one of the two fields.

Although these queries will work without an index, most applications will find this approach too slow, except perhaps for occasional ad-hoc searches. Practical use of text searching usually requires creating an index.

### 12.2.2. Creating Indexes

We can create a GIN index (Section 12.9) to speed up text searches:

```sql
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', body));
```

Notice that the 2-argument version of `to_tsvector` is used. Only text search functions that specify a configuration name can be used in expression indexes (Section 11.7). This is because the index contents must be unaffected by default_text_search_config. If they were affected, the index contents might be inconsistent because different entries could contain tsvector that were created with different text search configurations, and there would be no way to guess which was which. It would be impossible to dump and restore such an index correctly.

Because the two-argument version of `to_tsvector` was used in the index above, only a query reference that uses the 2-argument version of `to_tsvector` with the same configuration name will use that index. That is, WHERE `to_tsvector('english', body) @@ 'a & b'` can use the index, but WHERE `to_tsvector(body) @@ 'a & b'` cannot. This ensures that an index will be used only with the same configuration used to create the index entries.

It is possible to set up more complex expression indexes wherein the configuration name is specified by another column, e.g.:

```sql
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector(config_name, body));
```

where `config_name` is a column in the `pgweb` table. This allows mixed configurations in the same index while recording which configuration was used for each index entry. This would be useful, for example, if the document collection contained documents in different languages. Again, queries that are to use the index must be phrased to match, e.g. WHERE `to_tsvector(config_name, body) @@ 'a & b'`. 
Indexes can even concatenate columns:

```sql
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', title || ' ' || body));
```

Another approach is to create a separate `tsvector` column to hold the output of `to_tsvector`. This example is a concatenation of `title` and `body`, using `coalesce` to ensure that one field will still be indexed when the other is `NULL`:

```sql
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
    to_tsvector('english', coalesce(title,"") || ' ' || coalesce(body,""));
```

Then we create a GIN index to speed up the search:

```sql
CREATE INDEX textsearch_idx ON pgweb USING gin(textsearchable_index_col);
```

Now we are ready to perform a fast full text search:

```sql
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC LIMIT 10;
```

When using a separate column to store the `tsvector` representation, it is necessary to create a trigger to keep the `tsvector` column current anytime `title` or `body` changes. Section 12.4.3 explains how to do that.

One advantage of the separate-column approach over an expression index is that it is not necessary to explicitly specify the text search configuration in queries in order to make use of the index. As shown in the example above, the query can depend on `default_text_search_config`. Another advantage is that searches will be faster, since it will not be necessary to redo the `to_tsvector` calls to verify index matches. (This is more important when using a GiST index than a GIN index; see Section 12.9.) The expression-index approach is simpler to set up, however, and it requires less disk space since the `tsvector` representation is not stored explicitly.

### 12.3. Controlling Text Search

To implement full text searching there must be a function to create a `tsvector` from a document and a `tsquery` from a user query. Also, we need to return results in a useful order, so we need a function that compares documents with respect to their relevance to the query. It’s also important to be able to display the results nicely. PostgreSQL provides support for all of these functions.

#### 12.3.1. Parsing Documents

PostgreSQL provides the function `to_tsvector` for converting a document to the `tsvector` data type.

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```
Chapter 12. Full Text Search

to_tsvector parses a textual document into tokens, reduces the tokens to lexemes, and returns a tsvector which lists the lexemes together with their positions in the document. The document is processed according to the specified or default text search configuration. Here is a simple example:

```sql
SELECT to_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');
```

```plaintext
to_tsvector
-----------------------------------------------------
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

In the example above we see that the resulting tsvector does not contain the words a, on, or it, the word rats became rat, and the punctuation sign – was ignored.

The to_tsvector function internally calls a parser which breaks the document text into tokens and assigns a type to each token. For each token, a list of dictionaries (Section 12.6) is consulted, where the list can vary depending on the token type. The first dictionary that recognizes the token emits one or more normalized lexemes to represent the token. For example, rats became rat because one of the dictionaries recognized that the word rats is a plural form of rat. Some words are recognized as stop words (Section 12.6.1), which causes them to be ignored since they occur too frequently to be useful in searching. In our example these are a, on, and it. If no dictionary in the list recognizes the token then it is also ignored. In this example that happened to the punctuation sign – because there are in fact no dictionaries assigned for its token type (Space symbols), meaning space tokens will never be indexed. The choices of parser, dictionaries and which types of tokens to index are determined by the selected text search configuration (Section 12.7). It is possible to have many different configurations in the same database, and predefined configurations are available for various languages. In our example we used the default configuration english for the English language.

The function setweight can be used to label the entries of a tsvector with a given weight, where a weight is one of the letters A, B, C, or D. This is typically used to mark entries coming from different parts of a document, such as title versus body. Later, this information can be used for ranking of search results.

Because to_tsvector(NULL) will return NULL, it is recommended to use coalesce whenever a field might be null. Here is the recommended method for creating a tsvector from a structured document:

```sql
UPDATE tt SET ti =
  setweight(to_tsvector(coalesce(title,""), 'A') ||
  setweight(to_tsvector(coalesce(keyword,""), 'B') ||
  setweight(to_tsvector(coalesce(abstract,""), 'C') ||
  setweight(to_tsvector(coalesce(body,""), 'D');
```

Here we have used setweight to label the source of each lexeme in the finished tsvector, and then merged the labeled tsvector values using the tsvector concatenation operator ||. (Section 12.4.1 gives details about these operations.)

12.3.2. Parsing Queries

PostgreSQL provides the functions to_tsquery and plainto_tsquery for converting a query to the tsquery data type. to_tsquery offers access to more features than plainto_tsquery, but is less forgiving about its input.

```sql
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```
to_tsquery creates a tsquery value from querytext, which must consist of single tokens separated by the Boolean operators \& (AND), | (OR) and ! (NOT). These operators can be grouped using parentheses. In other words, the input to to_tsquery must already follow the general rules for tsquery input, as described in Section 8.11. The difference is that while basic tsquery input takes the tokens at face value, to_tsquery normalizes each token to a lexeme using the specified or default configuration, and discards any tokens that are stop words according to the configuration. For example:

```sql
SELECT to_tsquery('english', 'The & Fat & Rats');
to_tsquery
-----------------
'fat' & 'rat'
```

As in basic tsquery input, weight(s) can be attached to each lexeme to restrict it to match only tsvector lexemes of those weight(s). For example:

```sql
SELECT to_tsquery('english', 'Fat | Rats:AB');
to_tsquery
-----------------
'fat' | 'rat':AB
```

to_tsquery can also accept single-quoted phrases. This is primarily useful when the configuration includes a thesaurus dictionary that may trigger on such phrases. In the example below, a thesaurus contains the rule supernovae stars : sn:

```sql
SELECT to_tsquery("'supernovae stars" & !crab");
to_tsquery
--------------
'sn' & '!crab'
```

Without quotes, to_tsquery will generate a syntax error for tokens that are not separated by an AND or OR operator.

```
plainto_tsquery([ config regconfig, ] querytext) returns tsquery
```

plainto_tsquery transforms unformatted text querytext to tsquery. The text is parsed and normalized much as for to_tsvector, then the & (AND) Boolean operator is inserted between surviving words.

Example:

```sql
SELECT plainto_tsquery('english', 'The Fat Rats');
plainto_tsquery
-----------------
'fat' & 'rat'
```

Note that plainto_tsquery cannot recognize either Boolean operators or weight labels in its input:

```sql
SELECT plaintexto_tsquery('english', 'The Fat & Rats:C');
plaintexto_tsquery
---------------------
'fat' & 'rat' & 'c'
```

Here, all the input punctuation was discarded as being space symbols.
12.3.3. Ranking Search Results

Ranking attempts to measure how relevant documents are to a particular query, so that when there are many matches the most relevant ones can be shown first. PostgreSQL provides two predefined ranking functions, which take into account lexical, proximity, and structural information; that is, they consider how often the query terms appear in the document, how close together the terms are in the document, and how important is the part of the document where they occur. However, the concept of relevancy is vague and very application-specific. Different applications might require additional information for ranking, e.g. document modification time. The built-in ranking functions are only examples. You can write your own ranking functions and/or combine their results with additional factors to fit your specific needs.

The two ranking functions currently available are:

\[ \text{ts\_rank}([ \text{weights float4}[]], \text{vector tsvector}, \text{query tsquery}, [, \text{normalization integer}]) \text{ returns float4} \]

Standard ranking function.

\[ \text{ts\_rank\_cd}([ \text{weights float4}[]], \text{vector tsvector}, \text{query tsquery}, [, \text{normalization integer}]) \text{ returns float4} \]

This function computes the cover density ranking for the given document vector and query, as described in Clarke, Cormack, and Tudhope’s "Relevance Ranking for One to Three Term Queries" in the journal "Information Processing and Management", 1999.

This function requires positional information in its input. Therefore it will not work on “stripped” tsvector values — it will always return zero.

For both these functions, the optional weights argument offers the ability to weigh word instances more or less heavily depending on how they are labeled. The weight arrays specify how heavily to weigh each category of word, in the order:

\{D-weight, C-weight, B-weight, A-weight\}

If no weights are provided, then these defaults are used:

\{0.1, 0.2, 0.4, 1.0\}

Typically weights are used to mark words from special areas of the document, like the title or an initial abstract, so that they can be treated as more or less important than words in the document body.

Since a longer document has a greater chance of containing a query term it is reasonable to take into account document size, e.g. a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances. Both ranking functions take an integer normalization option that specifies whether and how a document’s length should impact its rank. The integer option controls several behaviors, so it is a bit mask: you can specify one or more behaviors using \text{\&} (for example, \text{2\&4}).

• 0 (the default) ignores the document length
• 1 divides the rank by \text{1 + the logarithm of the document length}
• 2 divides the rank by the document length
• 4 divides the rank by the mean harmonic distance between extents (this is implemented only by \text{ts\_rank\_cd})
• 8 divides the rank by the number of unique words in document
• 16 divides the rank by \text{1 + the logarithm of the number of unique words in document}
• 32 divides the rank by itself + 1

If more than one flag bit is specified, the transformations are applied in the order listed.

It is important to note that the ranking functions do not use any global information, so it is impossible to produce a fair normalization to 1% or 100% as sometimes desired. Normalization option 32 ($\text{rank}/(\text{rank}+1)$) can be applied to scale all ranks into the range zero to one, but of course this is just a cosmetic change; it will not affect the ordering of the search results.

Here is an example that selects only the ten highest-ranked matches:

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC LIMIT 10;
```

```
title | rank
-----------------------------------------------+----------
Neutrinos in the Sun | 3.1
The Sudbury Neutrino Detector | 2.4
A MACHO View of Galactic Dark Matter | 2.01317
Hot Gas and Dark Matter | 1.91171
The Virgo Cluster: Hot Plasma and Dark Matter | 1.90953
Rafting for Solar Neutrinos | 1.9
NGC 4650A: Strange Galaxy and Dark Matter | 1.85774
Hot Gas and Dark Matter | 1.6123
Ice Fishing for Cosmic Neutrinos | 1.6
Weak Lensing Distorts the Universe | 0.818218
```

This is the same example using normalized ranking:

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC LIMIT 10;
```

```
title | rank
-----------------------------------------------+-------------------
Neutrinos in the Sun | 0.756097569485493
The Sudbury Neutrino Detector | 0.705882361190954
A MACHO View of Galactic Dark Matter | 0.668123210574724
Hot Gas and Dark Matter | 0.6565958650282
The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
Rafting for Solar Neutrinos | 0.655172410958162
NGC 4650A: Strange Galaxy and Dark Matter | 0.65072921219637
Hot Gas and Dark Matter | 0.617195790024749
Ice Fishing for Cosmic Neutrinos | 0.615384618911517
Weak Lensing Distorts the Universe | 0.450010798361481
```

Ranking can be expensive since it requires consulting the tsvector of each matching document, which can be I/O bound and therefore slow. Unfortunately, it is almost impossible to avoid since practical queries often result in large numbers of matches.
12.3.4. Highlighting Results

To present search results it is ideal to show a part of each document and how it is related to the query. Usually, search engines show fragments of the document with marked search terms. PostgreSQL provides a function `ts_headline` that implements this functionality.

```
SELECT ts_headline('english', 'The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.', to_tsquery('query & similarity'));
```

The function `ts_headline` accepts a document along with a query, and returns an excerpt from the document in which terms from the query are highlighted. The configuration to be used to parse the document can be specified by `config`; if `config` is omitted, the `default_text_search_config` configuration is used.

If an `options` string is specified it must consist of a comma-separated list of one or more `option=value` pairs. The available options are:

- **StartSel, StopSel**: the strings with which query words appearing in the document should be delimited to distinguish them from other excerpted words.
- **MaxWords, MinWords**: these numbers determine the longest and shortest headlines to output.
- **ShortWord**: words of this length or less will be dropped at the start and end of a headline. The default value of three eliminates the English articles.
- **HighlightAll**: Boolean flag; if `true` the whole document will be highlighted.

Any unspecified options receive these defaults:

```
StartSel=<b>, StopSel=</b>, MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE
```

For example:

```
SELECT ts_headline('english', 'The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.', to_tsquery('query & similarity'));
```

```
SELECT ts_headline('english', 'The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.', to_tsquery('query & similarity'), 'StartSel = <, StopSel = >');
```

```
SELECT ts_headline('english', 'The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.', to_tsquery('query & similarity'), 'StartSel = <, StopSel = >');
```

```
SELECT ts_headline('english', 'The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.', to_tsquery('query & similarity'), 'StartSel = <, StopSel = >');
```

```
SELECT ts_headline('english', 'The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.', to_tsquery('query & similarity'), 'StartSel = <, StopSel = >');
```

Any unspecified options receive these defaults:

```
StartSel=<b>, StopSel=</b>, MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE
```
ts_headline uses the original document, not a tsvector summary, so it can be slow and should be used with care. A typical mistake is to call ts_headline for every matching document when only ten documents are to be shown. SQL subqueries can help; here is an example:

```sql
SELECT id, ts_headline(body, q), rank
FROM (SELECT id, body, q, ts_rank_cd(ti, q) AS rank
     FROM apod, to_tsquery('stars') q
     WHERE ti @@ q
     ORDER BY rank DESC LIMIT 10) AS foo;
```

12.4. Additional Features

This section describes additional functions and operators that are useful in connection with text search.

12.4.1. Manipulating Documents

Section 12.3.1 showed how raw textual documents can be converted into tsvector values. PostgreSQL also provides functions and operators that can be used to manipulate documents that are already in tsvector form.

```
tsvector || tsvector
```

The tsvector concatenation operator returns a vector which combines the lexemes and positional information of the two vectors given as arguments. Positions and weight labels are retained during the concatenation. Positions appearing in the right-hand vector are offset by the largest position mentioned in the left-hand vector, so that the result is nearly equivalent to the result of performing to_tsvector on the concatenation of the two original document strings. (The equivalence is not exact, because any stop-words removed from the end of the left-hand argument will not affect the result, whereas they would have affected the positions of the lexemes in the right-hand argument if textual concatenation were used.)

One advantage of using concatenation in the vector form, rather than concatenating text before applying to_tsvector, is that you can use different configurations to parse different sections of the document. Also, because the setweight function marks all lexemes of the given vector the same way, it is necessary to parse the text and do setweight before concatenating if you want to label different parts of the document with different weights.

```
setweight(vector tsvector, weight "char") returns tsvector
```

This function returns a copy of the input vector in which every position has been labeled with the given weight, either A, B, C, or D. (D is the default for new vectors and as such is not displayed on output.) These labels are retained when vectors are concatenated, allowing words from different parts of a document to be weighted differently by ranking functions.

Note that weight labels apply to positions, not lexemes. If the input vector has been stripped of positions then setweight does nothing.
Chapter 12. Full Text Search

length(vector tsvector) returns integer

Returns the number of lexemes stored in the vector.

strip(vector tsvector) returns tsvector

Returns a vector which lists the same lexemes as the given vector, but which lacks any position or weight information. While the returned vector is much less useful than an unstripped vector for relevance ranking, it will usually be much smaller.

12.4.2. Manipulating Queries

Section 12.3.2 showed how raw textual queries can be converted into tsquery values. PostgreSQL also provides functions and operators that can be used to manipulate queries that are already in tsquery form.

\[\text{tsquery} \&\& \text{tsquery}\]

Returns the AND-combination of the two given queries.

\[\text{tsquery} || \text{tsquery}\]

Returns the OR-combination of the two given queries.

\[!! \text{tsquery}\]

Returns the negation (NOT) of the given query.

\[\text{numnode(query tsquery) returns integer}\]

Returns the number of nodes (lexemes plus operators) in a tsquery. This function is useful to determine if the query is meaningful (returns > 0), or contains only stop words (returns 0). Examples:

```sql
SELECT numnode(plainto_tsquery('the any'));
NOTICE: query contains only stopword(s) or doesn’t contain lexeme(s), ignored
numnode
---------
0
```

```sql
SELECT numnode('foo & bar':tsquery);
numnode
---------
3
```

\[\text{querytree(query tsquery) returns text}\]

Returns the portion of a tsquery that can be used for searching an index. This function is useful for detecting unindexable queries, for example those containing only stop words or only negated terms. For example:

```sql
SELECT querytree(to_tsquery('!defined'));
querytree
```
12.4.2.1. Query Rewriting

The ts_rewrite family of functions search a given tsquery for occurrences of a target subquery, and replace each occurrence with another substitute subquery. In essence this operation is a tsquery-specific version of substring replacement. A target and substitute combination can be thought of as a query rewrite rule. A collection of such rewrite rules can be a powerful search aid. For example, you can expand the search using synonyms (e.g., new york, big apple, nyc, gotham) or narrow the search to direct the user to some hot topic. There is some overlap in functionality between this feature and thesaurus dictionaries (Section 12.6.4). However, you can modify a set of rewrite rules on-the-fly without reindexing, whereas updating a thesaurus requires reindexing to be effective.

\[
\text{ts_rewrite (query tsquery, target tsquery, substitute tsquery) returns tsquery}
\]

This form of ts_rewrite simply applies a single rewrite rule: target is replaced by substitute wherever it appears in query. For example:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
```

\[
\text{ts_rewrite (query tsquery, select text) returns tsquery}
\]

This form of ts_rewrite accepts a starting query and a SQL select command, which is given as a text string. The select must yield two columns of tsquery type. For each row of the select result, occurrences of the first column value (the target) are replaced by the second column value (the substitute) within the current query value. For example:

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');
SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
```

Note that when multiple rewrite rules are applied in this way, the order of application can be important; so in practice you will want the source query to ORDER BY some ordering key.

Let’s consider a real-life astronomical example. We’ll expand query supernovae using table-driven rewriting rules:

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));
SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
```

We can change the rewriting rules just by updating the table:

```
UPDATE aliases SET s = to_tsquery('supernovae|sn & !nebulae') WHERE t = to_tsquery('supernovae');
```
SELECT ts_rewrite(to_tquery('supernovae & crab'), 'SELECT * FROM aliases');

\texttt{ts_rewrite}

\begin{verbatim}
    'crab' \& \{ 'supernova' | 'sn' \& '!nebula' \}
\end{verbatim}

Rewriting can be slow when there are many rewriting rules, since it checks every rule for a possible hit. To filter out obvious non-candidate rules we can use the containment operators for the \texttt{tsquery} type. In the example below, we select only those rules which might match the original query:

\begin{verbatim}
SELECT ts_rewrite('a \& b':tsquery,
    'SELECT t,s FROM aliases WHERE "a \& b":tsquery @> t');
\end{verbatim}

\texttt{ts_rewrite}

\begin{verbatim}
    'b' \& 'c'
\end{verbatim}

### 12.4.3. Triggers for Automatic Updates

When using a separate column to store the \texttt{tsvector} representation of your documents, it is necessary to create a trigger to update the \texttt{tsvector} column when the document content columns change. Two built-in trigger functions are available for this, or you can write your own.

\begin{verbatim}
    tsvector_update_trigger(tsvector_column_name, config_name, text_column_name [, ... ])
    tsvector_update_trigger_column(tsvector_column_name, config_column_name, text_column_name)
\end{verbatim}

These trigger functions automatically compute a \texttt{tsvector} column from one or more textual columns, under the control of parameters specified in the \texttt{CREATE TRIGGER} command. An example of their use is:

\begin{verbatim}
CREATE TABLE messages {
    title     text,
    body      text,
    tsv         tsvector
};

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE
    tsvector_update_trigger(tsv, 'pg\_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;

\begin{verbatim}
    title | body | tsv
    -----------------------------------------
    title here | the body text is here | 'bodi':4 'text':5 'titl':1
\end{verbatim}

SELECT title, body FROM messages WHERE tsv @@ to_tquery('title \& body');

\begin{verbatim}
    title | body
    -----------------------------------------
    title here | the body text is here
\end{verbatim}

265
Having created this trigger, any change in title or body will automatically be reflected into tsv, without the application having to worry about it.

The first trigger argument must be the name of the tsvector column to be updated. The second argument specifies the text search configuration to be used to perform the conversion. For tsvector_update_trigger, the configuration name is simply given as the second trigger argument. It must be schema-qualified as shown above, so that the trigger behavior will not change with changes in search_path. For tsvector_update_trigger_column, the second trigger argument is the name of another table column, which must be of type regconfig. This allows a per-row selection of configuration to be made. The remaining argument(s) are the names of textual columns (of type text, varchar, or char). These will be included in the document in the order given. NULL values will be skipped (but the other columns will still be indexed).

A limitation of the built-in triggers is that they treat all the input columns alike. To process columns differently — for example, to weight title differently from body — it is necessary to write a custom trigger. Here is an example using PL/pgSQL as the trigger language:

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
    new.tsv :=
        setweight(to_tsvector('pg_catalog.english', coalesce(new.title,"")), 'A') ||
        setweight(to_tsvector('pg_catalog.english', coalesce(new.body,"")), 'D');
    return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE messages_trigger();
```

Keep in mind that it is important to specify the configuration name explicitly when creating tsvector values inside triggers, so that the column’s contents will not be affected by changes to default_text_search_config. Failure to do this is likely to lead to problems such as search results changing after a dump and reload.

### 12.4.4. Gathering Document Statistics

The function ts_stat is useful for checking your configuration and for finding stop-word candidates.

```
ts_stat(sqlquery text, [ weights text, ] OUT word text, OUT ndoc integer, OUT nentry integer)
```

*sqlquery* is a text value containing a SQL query which must return a single tsvector column. ts_stat executes the query and returns statistics about each distinct lexeme (word) contained in the tsvector data. The columns returned are:

- **word text** — the value of a lexeme
- **ndoc integer** — number of documents (tsvectors) the word occurred in
- **nentry integer** — total number of occurrences of the word

If *weights* is supplied, only occurrences having one of those weights are counted.

For example, to find the ten most frequent words in a document collection:
12.5. Parsers

Text search parsers are responsible for splitting raw document text into tokens and identifying each token’s type, where the set of possible types is defined by the parser itself. Note that a parser does not modify the text at all — it simply identifies plausible word boundaries. Because of this limited scope, there is less need for application-specific custom parsers than there is for custom dictionaries.

At present PostgreSQL provides just one built-in parser, which has been found to be useful for a wide range of applications.

The built-in parser is named pg_catalog.default. It recognizes 23 token types:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>asciiword</td>
<td>Word, all ASCII letters</td>
<td>elephant</td>
</tr>
<tr>
<td>word</td>
<td>Word, all letters</td>
<td>mañana</td>
</tr>
<tr>
<td>numword</td>
<td>Word, letters and digits</td>
<td>betal</td>
</tr>
<tr>
<td>asciihword</td>
<td>Hyphenated word, all ASCII</td>
<td>up-to-date</td>
</tr>
<tr>
<td>hword</td>
<td>Hyphenated word, all letters</td>
<td>lógico-matemática</td>
</tr>
<tr>
<td>numhword</td>
<td>Hyphenated word, letters and digits</td>
<td>postgresql-betal</td>
</tr>
<tr>
<td>hword_asciipart</td>
<td>Hyphenated word part, all ASCII</td>
<td>postgresql in the context postgresql-betal</td>
</tr>
<tr>
<td>hword_part</td>
<td>Hyphenated word part, all letters</td>
<td>lógico or matemática in the context lógico-matemática</td>
</tr>
<tr>
<td>hword_numpart</td>
<td>Hyphenated word part, letters and digits</td>
<td>betal in the context postgresql-betal</td>
</tr>
<tr>
<td>email</td>
<td>Email address</td>
<td><a href="mailto:foo@example.com">foo@example.com</a></td>
</tr>
<tr>
<td>protocol</td>
<td>Protocol head</td>
<td>http://</td>
</tr>
<tr>
<td>url</td>
<td>URL</td>
<td>example.com/stuff/index.html</td>
</tr>
<tr>
<td>host</td>
<td>Host</td>
<td>example.com</td>
</tr>
<tr>
<td>url_path</td>
<td>URL path</td>
<td>/stuff/index.html, in the context of a URL</td>
</tr>
<tr>
<td>file</td>
<td>File or path name</td>
<td>/usr/local/foo.txt, if not within a URL</td>
</tr>
</tbody>
</table>
Chapter 12. Full Text Search

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sfloat</td>
<td>Scientific notation</td>
<td>-1.234e56</td>
</tr>
<tr>
<td>float</td>
<td>Decimal notation</td>
<td>-1.234</td>
</tr>
<tr>
<td>int</td>
<td>Signed integer</td>
<td>-1234</td>
</tr>
<tr>
<td>uint</td>
<td>Unsigned integer</td>
<td>1234</td>
</tr>
<tr>
<td>version</td>
<td>Version number</td>
<td>8.3.0</td>
</tr>
<tr>
<td>tag</td>
<td>XML tag</td>
<td>&lt;a href=&quot;dictionaries.html&quot;&gt;</td>
</tr>
<tr>
<td>entity</td>
<td>XML entity</td>
<td>&amp;</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td>(any whitespace or punctuation not otherwise recognized)</td>
</tr>
</tbody>
</table>

Note: The parser’s notion of a “letter” is determined by the server’s locale setting, specifically lc_ctype. Words containing only the basic ASCII letters are reported as a separate token type, since it is sometimes useful to distinguish them. In most European languages, token types word and asciword should always be treated alike.

It is possible for the parser to produce overlapping tokens from the same piece of text. As an example, a hyphenated word will be reported both as the entire word and as each component:

```sql
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
```

<table>
<thead>
<tr>
<th>alias</th>
<th>description</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>numhword</td>
<td>Hyphenated word, letters and digits</td>
<td>foo-bar-beta1</td>
</tr>
<tr>
<td>hword_asciipart</td>
<td>Hyphenated word part, all ASCII</td>
<td>foo</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td>-</td>
</tr>
<tr>
<td>hword_asciipart</td>
<td>Hyphenated word part, all ASCII</td>
<td>bar</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td>-</td>
</tr>
<tr>
<td>hword_numpart</td>
<td>Hyphenated word part, letters and digits</td>
<td>beta</td>
</tr>
</tbody>
</table>

This behavior is desirable since it allows searches to work for both the whole compound word and for components. Here is another instructive example:

```sql
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
```

<table>
<thead>
<tr>
<th>alias</th>
<th>description</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td>Protocol head</td>
<td>http://</td>
</tr>
<tr>
<td>url</td>
<td>URL</td>
<td>example.com/stuff/index.html</td>
</tr>
<tr>
<td>host</td>
<td>Host</td>
<td>example.com</td>
</tr>
<tr>
<td>url_path</td>
<td>URL path</td>
<td>/stuff/index.html</td>
</tr>
</tbody>
</table>

12.6. Dictionaries

Dictionaries are used to eliminate words that should not be considered in a search (stop words), and to normalize words so that different derived forms of the same word will match. A successfully normalized word is called a lexeme. Aside from improving search quality, normalization and removal of
stop words reduce the size of the tsvector representation of a document, thereby improving performance. Normalization does not always have linguistic meaning and usually depends on application semantics.

Some examples of normalization:

- Linguistic - Ispell dictionaries try to reduce input words to a normalized form; stemmer dictionaries remove word endings
- URL locations can be canonicalized to make equivalent URLs match:
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- Color names can be replaced by their hexadecimal values, e.g., red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF
- If indexing numbers, we can remove some fractional digits to reduce the range of possible numbers, so for example 3.14159265359, 3.1415926, 3.14 will be the same after normalization if only two digits are kept after the decimal point.

A dictionary is a program that accepts a token as input and returns:

- an array of lexemes if the input token is known to the dictionary (notice that one token can produce more than one lexeme)
- an empty array if the dictionary knows the token, but it is a stop word
- NULL if the dictionary does not recognize the input token

PostgreSQL provides predefined dictionaries for many languages. There are also several predefined templates that can be used to create new dictionaries with custom parameters. Each predefined dictionary template is described below. If no existing template is suitable, it is possible to create new ones; see the contrib/ area of the PostgreSQL distribution for examples.

A text search configuration binds a parser together with a set of dictionaries to process the parser’s output tokens. For each token type that the parser can return, a separate list of dictionaries is specified by the configuration. When a token of that type is found by the parser, each dictionary in the list is consulted in turn, until some dictionary recognizes it as a known word. If it is identified as a stop word, or if no dictionary recognizes the token, it will be discarded and not indexed or searched for. The general rule for configuring a list of dictionaries is to place first the most narrow, most specific dictionary, then the more general dictionaries, finishing with a very general dictionary, like a Snowball stemmer or simple, which recognizes everything. For example, for an astronomy-specific search (astro_en configuration) one could bind token type asciiword (ASCII word) to a synonym dictionary of astronomical terms, a general English dictionary and a Snowball English stemmer:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

### 12.6.1. Stop Words

Stop words are words that are very common, appear in almost every document, and have no discrimination value. Therefore, they can be ignored in the context of full text searching. For example, every
English text contains words like a and the, so it is useless to store them in an index. However, stop words do affect the positions in tsvector, which in turn affect ranking:

```sql
SELECT to_tsvector('english','in the list of stop words');
```

```
'list':3 'stop':5 'word':6
```

The missing positions 1,2,4 are because of stop words. Ranks calculated for documents with and without stop words are quite different:

```sql
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list & stop'));
```

```
ts_rank_cd
------------
0.05
```

```sql
SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list & stop'));
```

```
ts_rank_cd
------------
0.1
```

It is up to the specific dictionary how it treats stop words. For example, ispell dictionaries first normalize words and then look at the list of stop words, while Snowball stemmers first check the list of stop words. The reason for the different behavior is an attempt to decrease noise.

### 12.6.2. Simple Dictionary

The simple dictionary template operates by converting the input token to lower case and checking it against a file of stop words. If it is found in the file then an empty array is returned, causing the token to be discarded. If not, the lower-cased form of the word is returned as the normalized lexeme. Alternatively, the dictionary can be configured to report non-stop-words as unrecognized, allowing them to be passed on to the next dictionary in the list.

Here is an example of a dictionary definition using the simple template:

```sql
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

Here, english is the base name of a file of stop words. The file’s full name will be $SHAREDIR/tsearch_data/english.stop, where $SHAREDIR means the PostgreSQL installation’s shared-data directory, often /usr/local/share/postgresql (use `pg_config --sharedir` to determine it if you’re not sure). The file format is simply a list of words, one per line. Blank lines and trailing spaces are ignored, and upper case is folded to lower case, but no other processing is done on the file contents.

Now we can test our dictionary:

```sql
SELECT ts_lexize('public.simple_dict','YeS');
```

```
ts_lexize
-----------
{yes}
```
We can also choose to return NULL, instead of the lower-cased word, if it is not found in the stop words file. This behavior is selected by setting the dictionary’s `Accept` parameter to false. Continuing the example:

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );
```

```
SELECT ts_lexize('public.simple_dict','YeS');
```

```
SELECT ts_lexize('public.simple_dict','The');
```

With the default setting of `Accept = true`, it is only useful to place a simple dictionary at the end of a list of dictionaries, since it will never pass on any token to a following dictionary. Conversely, `Accept = false` is only useful when there is at least one following dictionary.

### Caution

Most types of dictionaries rely on configuration files, such as files of stop words. These files must be stored in UTF-8 encoding. They will be translated to the actual database encoding, if that is different, when they are read into the server.

### Caution

Normally, a database session will read a dictionary configuration file only once, when it is first used within the session. If you modify a configuration file and want to force existing sessions to pick up the new contents, issue an `ALTER TEXT SEARCH DICTIONARY` command on the dictionary. This can be a “dummy” update that doesn’t actually change any parameter values.

### 12.6.3. Synonym Dictionary

This dictionary template is used to create dictionaries that replace a word with a synonym. Phrases are not supported (use the thesaurus template (Section 12.6.4) for that). A synonym dictionary can be used to overcome linguistic problems, for example, to prevent an English stemmer dictionary from reducing the word ’Paris’ to ’pari’. It is enough to have a `Paris paris` line in the synonym dictionary and put it before the `english_stem` dictionary:

```
SELECT * FROM ts_debug('english', 'Paris');
```

```
<table>
<thead>
<tr>
<th>alias</th>
<th>description</th>
<th>token</th>
<th>dictionaries</th>
<th>dictionary</th>
<th>lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Chapter 12. Full Text Search

CREATE TEXT SEARCH DICTIONARY my_synonym {
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
};

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');

The only parameter required by the synonym template is SYNONYMS, which is the base name of its configuration file — my_synonyms in the above example. The file’s full name will be $SHAREDIR/tsearch_data/my_synonyms.syn (where $SHAREDIR means the PostgreSQL installation’s shared-data directory). The file format is just one line per word to be substituted, with the word followed by its synonym, separated by white space. Blank lines and trailing spaces are ignored, and upper case is folded to lower case.

12.6.4. Thesaurus Dictionary

A thesaurus dictionary (sometimes abbreviated as TZ) is a collection of words that includes information about the relationships of words and phrases, i.e., broader terms (BT), narrower terms (NT), preferred terms, non-preferred terms, related terms, etc.

Basically a thesaurus dictionary replaces all non-preferred terms by one preferred term and, optionally, preserves the original terms for indexing as well. PostgreSQL’s current implementation of the thesaurus dictionary is an extension of the synonym dictionary with added phrase support. A thesaurus dictionary requires a configuration file of the following format:

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

where the colon (:) symbol acts as a delimiter between a a phrase and its replacement.

A thesaurus dictionary uses a subdictionary (which is specified in the dictionary’s configuration) to normalize the input text before checking for phrase matches. It is only possible to select one subdictionary. An error is reported if the subdictionary fails to recognize a word. In that case, you should remove the use of the word or teach the subdictionary about it. You can place an asterisk (*) at the beginning of an indexed word to skip applying the subdictionary to it, but all sample words must be known to the subdictionary.

The thesaurus dictionary chooses the longest match if there are multiple phrases matching the input, and ties are broken by using the last definition.

Specific stop words recognized by the subdictionary cannot be specified; instead use ? to mark the location where any stop word can appear. For example, assuming that a and the are stop words according to the subdictionary:
matches a one the two and the one a two; both would be replaced by swsw.

Since a thesaurus dictionary has the capability to recognize phrases it must remember its state and interact with the parser. A thesaurus dictionary uses these assignments to check if it should handle the next word or stop accumulation. The thesaurus dictionary must be configured carefully. For example, if the thesaurus dictionary is assigned to handle only the asciiword token, then a thesaurus dictionary definition like one 7 will not work since token type uint is not assigned to the thesaurus dictionary.

**Caution**

Thesauruses are used during indexing so any change in the thesaurus dictionary's parameters requires reindexing. For most other dictionary types, small changes such as adding or removing stopwords does not force reindexing.

### 12.6.4.1. Thesaurus Configuration

To define a new thesaurus dictionary, use the *thesaurus* template. For example:

```sql
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (  
    TEMPLATE = thesaurus,  
    DictFile = mythesaurus,  
    Dictionary = pg_catalog.english_stem
);
```

Here:

- `thesaurus_simple` is the new dictionary's name
- `mythesaurus` is the base name of the thesaurus configuration file. (Its full name will be `$SHAREDIR/tsearch_data/mythesaurus.ths`, where `$SHAREDIR` means the installation shared-data directory.)
- `pg_catalog.english_stem` is the subdictionary (here, a Snowball English stemmer) to use for thesaurus normalization. Notice that the subdictionary will have its own configuration (for example, stop words), which is not shown here.

Now it is possible to bind the thesaurus dictionary `thesaurus_simple` to the desired token types in a configuration, for example:

```sql
ALTER TEXT SEARCH CONFIGURATION russian  
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart WITH thesaurus_simple;
```

### 12.6.4.2. Thesaurus Example

Consider a simple astronomical thesaurus `thesaurus_astro`, which contains some astronomical word combinations:

- supernovae stars : sn
- crab nebulae : crab
Below we create a dictionary and bind some token types to an astronomical thesaurus and English stemmer:

```sql
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (  
    TEMPLATE = thesaurus,  
    DictFile = thesaurus_astro,  
    Dictionary = english_stem
);
```

ALTER TEXT SEARCH CONFIGURATION russian
ALTER MAPPING FOR asciiword, asciihword, hword_asciipart WITH thesaurus_astro, english_stem;

Now we can see how it works. `ts_lexize` is not very useful for testing a thesaurus, because it treats its input as a single token. Instead we can use `plainto_tsquery` and `to_tsvector` which will break their input strings into multiple tokens:

```sql
SELECT plainto_tsquery('supernova star');
plainto_tsquery
-----------------
'sn'

SELECT to_tsvector('supernova star');
to_tsvector
-----------------
'sn':1
```

In principle, one can use `to_tsquery` if you quote the argument:

```sql
SELECT to_tsquery('"supernova star"');
to_tsquery
-----------------
'sn'
```

Notice that `supernova star` matches `supernovae stars` in `thesaurus_astro` because we specified the `english_stem` stemmer in the thesaurus definition. The stemmer removed the `e` and `s`.

To index the original phrase as well as the substitute, just include it in the right-hand part of the definition:

```sql
supernovae stars : sn supernovae stars
```

```sql
SELECT plainto_tsquery('supernova star');
plainto_tsquery
-----------------------------
'sn' & 'supernova' & 'star'
```

### 12.6.5. Ispell Dictionary

The Ispell dictionary template supports morphological dictionaries, which can normalize many different linguistic forms of a word into the same lexeme. For example, an English Ispell dictionary can match all declensions and conjugations of the search term `bank`, e.g. `banking`, `banked`, `banks`, `banks'`, and `bank’s`. 

274
The standard PostgreSQL distribution does not include any Ispell configuration files. Dictionaries for a large number of languages are available from Ispell\(^1\). Also, some more modern dictionary file formats are supported — MySpell\(^2\) (OO < 2.0.1) and Hunspell\(^3\) (OO \(\geq\) 2.0.2). A large list of dictionaries is available on the OpenOffice Wiki\(^4\).

To create an Ispell dictionary, use the built-in `ispell` template and specify several parameters:

```sql
CREATE TEXT SEARCH DICTIONARY english_ispell (
  TEMPLATE = ispell,
  DictFile = english,
  AffFile = english,
  StopWords = english
);
```

Here, `DictFile`, `AffFile`, and `StopWords` specify the base names of the dictionary, affixes, and stop-words files. The stop-words file has the same format explained above for the simple dictionary type. The format of the other files is not specified here but is available from the above-mentioned web sites.

Ispell dictionaries usually recognize a limited set of words, so they should be followed by another broader dictionary, for example, a Snowball dictionary, which recognizes everything.

Ispell dictionaries support splitting compound words. This is a nice feature and PostgreSQL supports it. Notice that the affix file should specify a special flag using the `compoundwords controlled` statement that marks dictionary words that can participate in compound formation:

```sql
compoundwords controlled z
```

Here are some examples for the Norwegian language:

```sql
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
{over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
{sjokoladefabrikk,sjokolade,fabrikk}
```

**Note:** MySpell does not support compound words. Hunspell has sophisticated support for compound words. At present, PostgreSQL implements only the basic compound word operations of Hunspell.

### 12.6.6. Snowball Dictionary

The Snowball dictionary template is based on the project of Martin Porter, inventor of the popular Porter’s stemming algorithm for the English language. Snowball now provides stemming algorithms for many languages (see the Snowball site\(^5\) for more information). Each algorithm understands how to reduce common variant forms of words to a base, or stem, spelling within its language. A Snowball dictionary requires a `language` parameter to identify which stemmer to use, and optionally can specify a `stopword` file name that gives a list of words to eliminate. (PostgreSQL’s standard stopword

---

lists are also provided by the Snowball project.) For example, there is a built-in definition equivalent to

```sql
CREATE TEXT SEARCH DICTIONARY english_stem {
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
};
```

The stopword file format is the same as already explained. A Snowball dictionary recognizes everything, whether or not it is able to simplify the word, so it should be placed at the end of the dictionary list. It is useless to have it before any other dictionary because a token will never pass through it to the next dictionary.

### 12.7. Configuration Example

A text search configuration specifies all options necessary to transform a document into a `tsvector`: the parser to use to break text into tokens, and the dictionaries to use to transform each token into a lexeme. Every call of `to_tsvector` or `to_tsquery` needs a text search configuration to perform its processing. The configuration parameter `default_text_search_config` specifies the name of the default configuration, which is the one used by text search functions if an explicit configuration parameter is omitted. It can be set in `postgresql.conf`, or set for an individual session using the `SET` command.

Several predefined text search configurations are available, and you can create custom configurations easily. To facilitate management of text search objects, a set of SQL commands is available, and there are several `psql` commands that display information about text search objects (Section 12.10).

As an example, we will create a configuration `pg`, starting from a duplicate of the built-in `english` configuration.

```sql
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

We will use a PostgreSQL-specific synonym list and store it in `$SHAREDIR/tsearch_data/pg_dict.syn`. The file contents look like:

```
postgres pg
pgsql pg
postgresql pg
```

We define the synonym dictionary like this:

```sql
CREATE TEXT SEARCH DICTIONARY pg_dict {
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
};
```

Next we register the Ispell dictionary `english_ispell`, which has its own configuration files:

```sql
CREATE TEXT SEARCH DICTIONARY english_ispell {
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
};
```
Now we can set up the mappings for words in configuration pg:

```sql
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart, word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

We choose not to index or search some token types that the built-in configuration does handle:

```sql
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Now we can test our configuration:

```sql
SELECT * FROM ts_debug('public.pg', 'PostgreSQL, the highly scalable, SQL compliant, open source object-relational database management system, is now undergoing beta testing of the next version of our software. ');
```

The next step is to set the session to use the new configuration, which was created in the public schema:

```sql
=> \dF
   List of text search configurations
   Schema    | Name | Description
   ---------+------|----------------
   public   | pg   |

SET default_text_search_config = 'public.pg';
SET SHOW default_text_search_config;
default_text_search_config
-----------------------------
public.pg
```

12.8. Testing and Debugging Text Search

The behavior of a custom text search configuration can easily become complicated enough to be confusing or undesirable. The functions described in this section are useful for testing text search objects. You can test a complete configuration, or test parsers and dictionaries separately.
12.8.1. Configuration Testing

The function `ts_debug` allows easy testing of a text search configuration.

```sql
ts_debug([ config regconfig, ] document text,
   OUT alias text,
   OUT description text,
   OUT token text,
   OUT dictionaries regdictionary[],
   OUT dictionary regdictionary,
   OUT lexemes text[])
returns setof record
```

`ts_debug` displays information about every token of `document` as produced by the parser and processed by the configured dictionaries. It uses the configuration specified by `config`, or `default_text_search_config` if that argument is omitted.

`ts_debug` returns one row for each token identified in the text by the parser. The columns returned are

- `alias text` — short name of the token type
- `description text` — description of the token type
- `token text` — text of the token
- `dictionaries regdictionary[]` — the dictionaries selected by the configuration for this token type
- `dictionary regdictionary` — the dictionary that recognized the token, or NULL if none did
- `lexemes text[]` — the lexeme(s) produced by the dictionary that recognized the token, or NULL if none did; an empty array (``) means it was recognized as a stop word

Here is a simple example:

```sql
SELECT * FROM ts_debug('english','a fat cat sat on a mat - it ate a fat rats');
```

<table>
<thead>
<tr>
<th>alias</th>
<th>description</th>
<th>token</th>
<th>dictionaries</th>
<th>dictionary</th>
<th>lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>a</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>fat</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{fat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>cat</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{cat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>sat</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{sat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>on</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>a</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>mat</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{mat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>it</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>ate</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{ate}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>a</td>
<td>{english_stem}</td>
<td>english_stem</td>
<td>{}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>{}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For a more extensive demonstration, we first create a public.english configuration and Ispell dictionary for the English language:

CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell {
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
};

ALTER TEXT SEARCH CONFIGURATION public.english
ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;

SELECT * FROM ts_debug('public.english','The Brightest supernovaes');

<table>
<thead>
<tr>
<th>alias</th>
<th>description</th>
<th>token</th>
<th>dictionaries</th>
<th>dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>The</td>
<td>{english_ispell,english_stem}</td>
<td>english_ispell</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>Brightest</td>
<td>{english_ispell,english_stem}</td>
<td>english_ispell</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Word, all ASCII</td>
<td>supernovaes</td>
<td>{english_ispell,english_stem}</td>
<td>english_stem</td>
</tr>
</tbody>
</table>

In this example, the word **Brightest** was recognized by the parser as an ASCII word (alias asciiword). For this token type the dictionary list is english_ispell and english_stem. The word was recognized by english_ispell, which reduced it to the noun **bright**. The word supernovaes is unknown to the english_ispell dictionary so it was passed to the next dictionary, and, fortunately, was recognized (in fact, english_stem is a Snowball dictionary which recognizes everything; that is why it was placed at the end of the dictionary list).

The word **The** was recognized by the english_ispell dictionary as a stop word (Section 12.6.1) and will not be indexed. The spaces are discarded too, since the configuration provides no dictionaries at all for them.

You can reduce the volume of output by explicitly specifying which columns you want to see:

SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english','The Brightest supernovaes');

<table>
<thead>
<tr>
<th>alias</th>
<th>token</th>
<th>dictionary</th>
<th>lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asciiword</td>
<td>The</td>
<td>english_ispell</td>
<td>{}</td>
</tr>
<tr>
<td>blank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>Brightest</td>
<td>english_ispell</td>
<td>{bright}</td>
</tr>
<tr>
<td>blank</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>asciiword</td>
<td>supernovaes</td>
<td>english_stem</td>
<td>{supernova}</td>
</tr>
</tbody>
</table>

279
12.8.2. Parser Testing

The following functions allow direct testing of a text search parser.

```sql
ts_parse(parser_name text, document text, OUT tokid integer, OUT token text) returns setof record
```

```sql
ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text) returns setof record
```

`ts_parse` parses the given `document` and returns a series of records, one for each token produced by parsing. Each record includes a `tokid` showing the assigned token type and a `token` which is the text of the token. For example:

```sql
SELECT * FROM ts_parse('default', '123 - a number');
```

<table>
<thead>
<tr>
<th>tokid</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>123</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>number</td>
</tr>
</tbody>
</table>

```sql
```

```sql
```

```sql
```

```sql
```

```sql
```

```sql
```

```sql
```

```sql
```
12.8.3. Dictionary Testing

The `ts_lexize` function facilitates dictionary testing.

\[ \text{ts_lexize}(\text{dict \ regdictionary, token text}) \text{ returns text[]} \]

`ts_lexize` returns an array of lexemes if the input `token` is known to the dictionary, or an empty array if the token is known to the dictionary but it is a stop word, or `NULL` if it is an unknown word.

Examples:

```sql
SELECT ts_lexize('english_stem', 'stars');
```

```
{star}
```

```sql
SELECT ts_lexize('english_stem', 'a');
```

```
{}
```

**Note:** The `ts_lexize` function expects a single `token`, not text. Here is a case where this can be confusing:

```sql
SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;
```

```
t
```

The thesaurus dictionary `thesaurus_astro` does know the phrase `supernovae stars`, but `ts_lexize` fails since it does not parse the input text but treats it as a single token. Use `plainto_tsquery` or `to_tsvector` to test thesaurus dictionaries, for example:

```sql
SELECT plainto_tsquery('supernovae stars');
```

```
'sn'
```
12.9. GiST and GIN Index Types

There are two kinds of indexes that can be used to speed up full text searches. Note that indexes are not mandatory for full text searching, but in cases where a column is searched on a regular basis, an index will usually be desirable.

```
CREATE INDEX name ON table USING gist(column);
```

Creates a GiST (Generalized Search Tree)-based index. The column can be of tsvector or tsquery type.

```
CREATE INDEX name ON table USING gin(column);
```

Creates a GIN (Generalized Inverted Index)-based index. The column must be of tsvector type.

There are substantial performance differences between the two index types, so it is important to understand which to use.

A GiST index is lossy, meaning that the index may produce false matches, and it is necessary to check the actual table row to eliminate such false matches. PostgreSQL does this automatically; for example, in the query plan below, the Filter: line indicates the index output will be rechecked:

```
EXPLAIN SELECT * FROM apod WHERE textsearch @@ to_tsquery('supernovae');
```

```
QUERY PLAN
-------------------------------------------------------------------------
Index Scan using textsearch_gidx on apod (cost=0.00..12.29 rows=2 width=1469)
  Index Cond: (textsearch @@ to_tsquery('supernovae'))
  Filter: (textsearch @@ to_tsquery('supernovae'))
-------------------------------------------------------------------------
```

GiST indexes are lossy because each document is represented in the index by a fixed-length signature. The signature is generated by hashing each word into a random bit in an n-bit string, with all these bits OR-ed together to produce an n-bit document signature. When two words hash to the same bit position there will be a false match. If all words in the query have matches (real or false) then the table row must be retrieved to see if the match is correct.

Lossiness causes performance degradation due to useless fetches of table records that turn out to be false matches. Since random access to table records is slow, this limits the usefulness of GiST indexes. The likelihood of false matches depends on several factors, in particular the number of unique words, so using dictionaries to reduce this number is recommended.

GIN indexes are not lossy but their performance depends logarithmically on the number of unique words.

Actually, GIN indexes store only the words (lexemes) of tsvector values, and not their weight labels. Thus, while a GIN index can be considered non-lossy for a query that does not specify weights, it is lossy for one that does. Thus a table row recheck is needed when using a query that involves weights. Unfortunately, in the current design of PostgreSQL, whether a recheck is needed is a static property of a particular operator, and not something that can be enabled or disabled on-the-fly depending on the values given to the operator. To deal with this situation without imposing the overhead of rechecks on queries that do not need them, the following approach has been adopted:

- The standard text match operator @@ is marked as non-lossy for GIN indexes.
• An additional match operator @@@ is provided, and marked as lossy for GIN indexes. This operator behaves exactly like @@ otherwise.
• When a GIN index search is initiated with the @@ operator, the index support code will throw an error if the query specifies any weights. This protects against giving wrong answers due to failure to recheck the weights.

In short, you must use @@@ rather than @@ to perform GIN index searches on queries that involve weight restrictions. For queries that do not have weight restrictions, either operator will work, but @@ will be faster. This awkwardness will probably be addressed in a future release of PostgreSQL.

In choosing which index type to use, GiST or GIN, consider these performance differences:

• GIN index lookups are about three times faster than GiST
• GIN indexes take about three times longer to build than GiST
• GIN indexes are about ten times slower to update than GiST
• GIN indexes are two-to-three times larger than GiST

As a rule of thumb, GIN indexes are best for static data because lookups are faster. For dynamic data, GiST indexes are faster to update. Specifically, GiST indexes are very good for dynamic data and fast if the number of unique words (lexemes) is under 100,000, while GIN indexes will handle 100,000+ lexemes better but are slower to update.

Note that GIN index build time can often be improved by increasing maintenance_work_mem, while GiST index build time is not sensitive to that parameter.

Partitioning of big collections and the proper use of GiST and GIN indexes allows the implementation of very fast searches with online update. Partitioning can be done at the database level using table inheritance and constraint_exclusion, or by distributing documents over servers and collecting search results using the contrib/dblink extension module. The latter is possible because ranking functions use only local information.

12.10. psql Support

Information about text search configuration objects can be obtained in psql using a set of commands:

```
\dF{d,p,t} [+][PATTERN]
```

An optional + produces more details.

The optional parameter PATTERN should be the name of a text search object, optionally schema-qualified. If PATTERN is omitted then information about all visible objects will be displayed. PATTERN can be a regular expression and can provide separate patterns for the schema and object names. The following examples illustrate this:

```
=> \dF *fulltext*
  List of text search configurations
  Schema | Name            | Description
  --------+------------------+------------------
  public  | fulltext_cfg     |                 

=> \dF *.fulltext*
  List of text search configurations
  Schema | Name            | Description
```

283
Chapter 12. Full Text Search

The available commands are:

```
\dF[+][PATTERN]
```

List text search configurations (add + for more detail).

```bash
=> \dF russian
```

List of text search configurations

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>russian</td>
<td>configuration for russian language</td>
</tr>
</tbody>
</table>

```bash
=> \dF+ russian
```

Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"

<table>
<thead>
<tr>
<th>Token</th>
<th>Dictionaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>asciihword</td>
<td>english_stem</td>
</tr>
<tr>
<td>asciiword</td>
<td>english_stem</td>
</tr>
<tr>
<td>email</td>
<td>simple</td>
</tr>
<tr>
<td>file</td>
<td>simple</td>
</tr>
<tr>
<td>float</td>
<td>simple</td>
</tr>
<tr>
<td>host</td>
<td>simple</td>
</tr>
<tr>
<td>hword</td>
<td>russian_stem</td>
</tr>
<tr>
<td>hword_asciipart</td>
<td>english_stem</td>
</tr>
<tr>
<td>hword_numpart</td>
<td>simple</td>
</tr>
<tr>
<td>hword_part</td>
<td>russian_stem</td>
</tr>
<tr>
<td>int</td>
<td>simple</td>
</tr>
<tr>
<td>numhword</td>
<td>simple</td>
</tr>
<tr>
<td>numword</td>
<td>simple</td>
</tr>
<tr>
<td>sfloat</td>
<td>simple</td>
</tr>
<tr>
<td>uint</td>
<td>simple</td>
</tr>
<tr>
<td>url</td>
<td>simple</td>
</tr>
<tr>
<td>url_path</td>
<td>simple</td>
</tr>
<tr>
<td>version</td>
<td>simple</td>
</tr>
<tr>
<td>word</td>
<td>russian_stem</td>
</tr>
</tbody>
</table>

```
\dFd[+][PATTERN]
```

List text search dictionaries (add + for more detail).

```bash
=> \dFd
```

List of text search dictionaries

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>danish_stem</td>
<td>snowball stemmer for danish language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>dutch_stem</td>
<td>snowball stemmer for dutch language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>english_stem</td>
<td>snowball stemmer for english language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>finnish_stem</td>
<td>snowball stemmer for finnish language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>french_stem</td>
<td>snowball stemmer for french language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>german_stem</td>
<td>snowball stemmer for german language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>hungarian_stem</td>
<td>snowball stemmer for hungarian language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>italian_stem</td>
<td>snowball stemmer for italian language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>norwegian_stem</td>
<td>snowball stemmer for norwegian language</td>
</tr>
</tbody>
</table>
Chapter 12. Full Text Search

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>portuguese_stem</td>
<td>snowball stemmer for portuguese language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>romanian_stem</td>
<td>snowball stemmer for romanian language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>russian_stem</td>
<td>snowball stemmer for russian language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>simple</td>
<td>simple dictionary: just lower case and check for stopwords</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>spanish_stem</td>
<td>snowball stemmer for spanish language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>swedish_stem</td>
<td>snowball stemmer for swedish language</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>turkish_stem</td>
<td>snowball stemmer for turkish language</td>
</tr>
</tbody>
</table>

\dFp[+] [PATTERN]

List text search parsers (add + for more detail).

```
=> \dFp
```

<table>
<thead>
<tr>
<th>List of text search parsers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>pg_catalog</td>
</tr>
</tbody>
</table>

```
=> \dFp+
```

Text search parser "pg_catalog.default"

```
<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start parse</td>
<td>prsd_start</td>
<td></td>
</tr>
<tr>
<td>Get next token</td>
<td>prsd_nexttoken</td>
<td></td>
</tr>
<tr>
<td>End parse</td>
<td>prsd_end</td>
<td></td>
</tr>
<tr>
<td>Get headline</td>
<td>prsd_headline</td>
<td></td>
</tr>
<tr>
<td>Get token types</td>
<td>prsd_lextype</td>
<td></td>
</tr>
</tbody>
</table>

Token types for parser "pg_catalog.default"

```
<table>
<thead>
<tr>
<th>Token name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asciihword</td>
<td>Hyphenated word, all ASCII</td>
</tr>
<tr>
<td>asciiwword</td>
<td>Word, all ASCII</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
</tr>
<tr>
<td>email</td>
<td>Email address</td>
</tr>
<tr>
<td>entity</td>
<td>XML entity</td>
</tr>
<tr>
<td>file</td>
<td>File or path name</td>
</tr>
<tr>
<td>float</td>
<td>Decimal notation</td>
</tr>
<tr>
<td>host</td>
<td>Host</td>
</tr>
<tr>
<td>hword</td>
<td>Hyphenated word, all letters</td>
</tr>
<tr>
<td>hword_ascipart</td>
<td>Hyphenated word part, all ASCII</td>
</tr>
<tr>
<td>hword_numpart</td>
<td>Hyphenated word part, letters and digits</td>
</tr>
<tr>
<td>hword_part</td>
<td>Hyphenated word part, all letters</td>
</tr>
<tr>
<td>int</td>
<td>Signed integer</td>
</tr>
<tr>
<td>numhword</td>
<td>Hyphenated word, letters and digits</td>
</tr>
<tr>
<td>numword</td>
<td>Word, letters and digits</td>
</tr>
<tr>
<td>protocol</td>
<td>Protocol head</td>
</tr>
<tr>
<td>sfloat</td>
<td>Scientific notation</td>
</tr>
<tr>
<td>tag</td>
<td>XML tag</td>
</tr>
<tr>
<td>uint</td>
<td>Unsigned integer</td>
</tr>
<tr>
<td>url</td>
<td>URL</td>
</tr>
<tr>
<td>url_path</td>
<td>URL path</td>
</tr>
<tr>
<td>version</td>
<td>Version number</td>
</tr>
<tr>
<td>word</td>
<td>Word, all letters</td>
</tr>
</tbody>
</table>

(23 rows)
List text search templates (add `+` for more detail).

```
=> \dFt[+][PATTERN]
```

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>ispell</td>
<td>ispell dictionary</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>simple</td>
<td>simple dictionary: just lower case and check for stopword</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>snowball</td>
<td>snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>synonym</td>
<td>synonym dictionary: replace word by its synonym</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>thesaurus</td>
<td>thesaurus dictionary: phrase by phrase substitution</td>
</tr>
</tbody>
</table>

### 12.11. Limitations

The current limitations of PostgreSQL’s text search features are:

- The length of each lexeme must be less than 2K bytes
- The length of a `tsvector` (lexemes + positions) must be less than 1 megabyte
- The number of lexemes must be less than $2^{64}$
- Position values in `tsvector` must be greater than 0 and no more than 16,383
- No more than 256 positions per lexeme
- The number of nodes (lexemes + operators) in a `tsquery` must be less than 32,768

For comparison, the PostgreSQL 8.1 documentation contained 10,441 unique words, a total of 335,420 words, and the most frequent word “postgresql” was mentioned 6,127 times in 655 documents.

Another example — the PostgreSQL mailing list archives contained 910,989 unique words with 57,491,343 lexemes in 461,020 messages.

### 12.12. Migration from Pre-8.3 Text Search

Applications that used the `contrib/tsearch2` add-on module for text searching will need some adjustments to work with the built-in features:

- Some functions have been renamed or had small adjustments in their argument lists, and all of them are now in the `pg_catalog` schema, whereas in a previous installation they would have been in `public` or another non-system schema. There is a new version of `contrib/tsearch2` (see Section F.31) that provides a compatibility layer to solve most problems in this area.
- The old `contrib/tsearch2` functions and other objects must be suppressed when loading `pg_dump` output from a pre-8.3 database. While many of them won’t load anyway, a few will and then cause problems. One simple way to deal with this is to load the new `contrib/tsearch2` module before restoring the dump; then it will block the old objects from being loaded.
- Text search configuration setup is completely different now. Instead of manually inserting rows into configuration tables, search is configured through the specialized SQL commands shown earlier.
in this chapter. There is not currently any automated support for converting an existing custom configuration for 8.3; you’re on your own here.

- Most types of dictionaries rely on some outside-the-database configuration files. These are largely compatible with pre-8.3 usage, but note the following differences:
  - Configuration files now must be placed in a single specified directory (\$SHAREDIR/tsearch_data), and must have a specific extension depending on the type of file, as noted previously in the descriptions of the various dictionary types. This restriction was added to forestall security problems.
  - Configuration files must be encoded in UTF-8 encoding, regardless of what database encoding is used.
  - In thesaurus configuration files, stop words must be marked with?.
Chapter 13. Concurrency Control

This chapter describes the behavior of the PostgreSQL database system when two or more sessions try to access the same data at the same time. The goals in that situation are to allow efficient access for all sessions while maintaining strict data integrity. Every developer of database applications should be familiar with the topics covered in this chapter.

13.1. Introduction

PostgreSQL provides a rich set of tools for developers to manage concurrent access to data. Internally, data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC). This means that while querying a database each transaction sees a snapshot of data (a database version) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing transaction isolation for each database session. MVCC, by eschewing explicit locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments.

The main advantage to using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading.

Table- and row-level locking facilities are also available in PostgreSQL for applications that cannot adapt easily to MVCC behavior. However, proper use of MVCC will generally provide better performance than locks. In addition, application-defined advisory locks provide a mechanism for acquiring locks that are not tied to a single transaction.

13.2. Transaction Isolation

The SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented between concurrent transactions. These undesirable phenomena are:

- **dirty read**
  A transaction reads data written by a concurrent uncommitted transaction.

- **nonrepeatable read**
  A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

- **phantom read**
  A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

The four transaction isolation levels and the corresponding behaviors are described in Table 13-1.

Table 13-1. SQL Transaction Isolation Levels
Chapter 13. Concurrency Control

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read committed</td>
<td>Not possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
</tbody>
</table>

In PostgreSQL, you can request any of the four standard transaction isolation levels. But internally, there are only two distinct isolation levels, which correspond to the levels Read Committed and Serializable. When you select the level Read Uncommitted you really get Read Committed, and when you select Repeatable Read you really get Serializable, so the actual isolation level might be stricter than what you select. This is permitted by the SQL standard: the four isolation levels only define which phenomena must not happen, they do not define which phenomena must happen. The reason that PostgreSQL only provides two isolation levels is that this is the only sensible way to map the standard isolation levels to the multiversion concurrency control architecture. The behavior of the available isolation levels is detailed in the following subsections.

To set the transaction isolation level of a transaction, use the command `SET TRANSACTION`.

13.2.1. Read Committed Isolation Level

*Read Committed* is the default isolation level in PostgreSQL. When a transaction runs on this isolation level, a `SELECT` query sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions. (However, the `SELECT` does see the effects of previous updates executed within its own transaction, even though they are not yet committed.) In effect, a `SELECT` query sees a snapshot of the database as of the instant that that query begins to run. Notice that two successive `SELECT` commands can see different data, even though they are within a single transaction, if other transactions commit changes during execution of the first `SELECT`.

`UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they will only find target rows that were committed as of the command start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the would-be updaters will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the second updater can proceed with updating the originally found row. If the first updater commits, the second updater will ignore the row if the first updater deleted it, otherwise it will attempt to apply its operation to the updated version of the row. The search condition of the command (the `WHERE` clause) is re-evaluated to see if the updated version of the row still matches the search condition. If so, the second updater proceeds with its operation, starting from the updated version of the row. (In the case of `SELECT FOR UPDATE` and `SELECT FOR SHARE`, that means it is the updated version of the row that is locked and returned to the client.)

Because of the above rule, it is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands that affected the same rows it is trying to update, but it does not see effects of those commands on other rows in the database. This behavior makes Read Committed mode unsuitable for commands that involve complex search conditions. However, it is just right for simpler cases. For example, consider updating bank balances with transactions like

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```
Chapter 13. Concurrency Control

If two such transactions concurrently try to change the balance of account 12345, we clearly want the second transaction to start from the updated version of the account’s row. Because each command is affecting only a predetermined row, letting it see the updated version of the row does not create any troublesome inconsistency.

Since in Read Committed mode each new command starts with a new snapshot that includes all transactions committed up to that instant, subsequent commands in the same transaction will see the effects of the committed concurrent transaction in any case. The point at issue here is whether or not within a single command we see an absolutely consistent view of the database.

The partial transaction isolation provided by Read Committed mode is adequate for many applications, and this mode is fast and simple to use. However, for applications that do complex queries and updates, it might be necessary to guarantee a more rigorously consistent view of the database than the Read Committed mode provides.

13.2.2. Serializable Isolation Level

The level Serializable provides the strictest transaction isolation. This level emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. However, applications using this level must be prepared to retry transactions due to serialization failures.

When a transaction is on the serializable level, a SELECT query sees only data committed before the transaction began; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed.) This is different from Read Committed in that the SELECT sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction. Thus, successive SELECT commands within a single transaction always see the same data.

UPDATE, DELETE, SELECT FOR UPDATE, and SELECT FOR SHARE commands behave the same as SELECT in terms of searching for target rows: they will only find target rows that were committed as of the transaction start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the serializable transaction will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the serializable transaction can proceed with updating the originally found row. But if the first updater commits (and actually updated or deleted the row, not just locked it) then the serializable transaction will be rolled back with the message

ERROR: could not serialize access due to concurrent update

because a serializable transaction cannot modify or lock rows changed by other transactions after the serializable transaction began.

When the application receives this error message, it should abort the current transaction and then retry the whole transaction from the beginning. The second time through, the transaction sees the previously-committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction’s update.

Note that only updating transactions might need to be retried; read-only transactions will never have serialization conflicts.

The Serializable mode provides a rigorous guarantee that each transaction sees a wholly consistent view of the database. However, the application has to be prepared to retry transactions when concurrent updates make it impossible to sustain the illusion of serial execution. Since the cost of redoing
complex transactions might be significant, this mode is recommended only when updating transactions contain logic sufficiently complex that they might give wrong answers in Read Committed mode. Most commonly, Serializable mode is necessary when a transaction executes several successive commands that must see identical views of the database.

### 13.2.2.1. Serializable Isolation versus True Serializability

The intuitive meaning (and mathematical definition) of “serializable” execution is that any two successfully committed concurrent transactions will appear to have executed strictly serially, one after the other — although which one appeared to occur first might not be predictable in advance. It is important to realize that forbidding the undesirable behaviors listed in Table 13-1 is not sufficient to guarantee true serializability, and in fact PostgreSQL’s Serializable mode does not guarantee serializable execution in this sense. As an example, consider a table `mytab`, initially containing

<table>
<thead>
<tr>
<th>class</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
</tr>
</tbody>
</table>

Suppose that serializable transaction A computes

```sql
SELECT SUM(value) FROM mytab WHERE class = 1;
```

and then inserts the result (30) as the value in a new row with `class = 2`. Concurrently, serializable transaction B computes

```sql
SELECT SUM(value) FROM mytab WHERE class = 2;
```

and obtains the result 300, which it inserts in a new row with `class = 1`. Then both transactions commit. None of the listed undesirable behaviors have occurred, yet we have a result that could not have occurred in either order serially. If A had executed before B, B would have computed the sum 330, not 300, and similarly the other order would have resulted in a different sum computed by A.

To guarantee true mathematical serializability, it is necessary for a database system to enforce predicate locking, which means that a transaction cannot insert or modify a row that would have matched the `WHERE` condition of a query in another concurrent transaction. For example, once transaction A has executed the query `SELECT ... WHERE class = 1`, a predicate-locking system would forbid transaction B from inserting any new row with class 1 until A has committed. Such a locking system is complex to implement and extremely expensive in execution, since every session must be aware of the details of every query executed by every concurrent transaction. And this large expense is mostly wasted, since in practice most applications do not do the sorts of things that could result in problems. (Certainly the example above is rather contrived and unlikely to represent real software.) For these reasons, PostgreSQL does not implement predicate locking.

In those cases where the possibility of nonserializable execution is a real hazard, problems can be prevented by appropriate use of explicit locking. Further discussion appears in the following sections.

---

1. Essentially, a predicate-locking system prevents phantom reads by restricting what is written, whereas MVCC prevents them by restricting what is read.
Chapter 13. Concurrency Control

13.3. Explicit Locking

PostgreSQL provides various lock modes to control concurrent access to data in tables. These modes can be used for application-controlled locking in situations where MVCC does not give the desired behavior. Also, most PostgreSQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. (For example, ALTER TABLE cannot safely be executed concurrently with other operations on the same table, so it obtains an exclusive lock on the table to enforce that.)

To examine a list of the currently outstanding locks in a database server, use the pg_locks system view. For more information on monitoring the status of the lock manager subsystem, refer to Chapter 26.

13.3.1. Table-Level Locks

The list below shows the available lock modes and the contexts in which they are used automatically by PostgreSQL. You can also acquire any of these locks explicitly with the command LOCK. Remember that all of these lock modes are table-level locks, even if the name contains the word “row”; the names of the lock modes are historical. To some extent the names reflect the typical usage of each lock mode — but the semantics are all the same. The only real difference between one lock mode and another is the set of lock modes with which each conflicts (see Table 13-2). Two transactions cannot hold locks of conflicting modes on the same table at the same time. (However, a transaction never conflicts with itself. For example, it might acquire ACCESS EXCLUSIVE lock and later acquire ACCESS SHARE lock on the same table.) Non-conflicting lock modes can be held concurrently by many transactions. Notice in particular that some lock modes are self-conflicting (for example, an ACCESS EXCLUSIVE lock cannot be held by more than one transaction at a time) while others are not self-conflicting (for example, an ACCESS SHARE lock can be held by multiple transactions).

Table-level lock modes

ACCESS SHARE

Conflicts with the ACCESS EXCLUSIVE lock mode only.

The SELECT command acquires a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.

ROW SHARE

Conflicts with the EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

The SELECT FOR UPDATE and SELECT FOR SHARE commands acquire a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR UPDATE/FOR SHARE).

ROW EXCLUSIVE

Conflicts with the SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

The commands UPDATE, DELETE, and INSERT acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables). In general, this lock mode will be acquired by any command that modifies the data in a table.

SHARE UPDATE EXCLUSIVE

Conflicts with the SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against
concurrent schema changes and VACUUM runs.

Acquired by VACUUM (without FULL), ANALYZE, and CREATE INDEX CONCURRENTLY.

SHARE

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes.

Acquired by CREATE INDEX (without CONCURRENTLY).

SHARE ROW EXCLUSIVE

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

This lock mode is not automatically acquired by any PostgreSQL command.

EXCLUSIVE

Conflicts with the ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode allows only concurrent ACCESS SHARE locks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

This lock mode is not automatically acquired on user tables by any PostgreSQL command. However it is acquired on certain system catalogs in some operations.

ACCESS EXCLUSIVE

Conflicts with locks of all modes (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE). This mode guarantees that the holder is the only transaction accessing the table in any way.

Acquired by the ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, and VACUUM FULL commands. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

Tip: Only an ACCESS EXCLUSIVE lock blocks a SELECT (without FOR UPDATE/SHARE) statement.

Once acquired, a lock is normally held till end of transaction. But if a lock is acquired after establishing a savepoint, the lock is released immediately if the savepoint is rolled back to. This is consistent with the principle that ROLLBACK cancels all effects of the commands since the savepoint. The same holds for locks acquired within a PL/pgSQL exception block: an error escape from the block releases locks acquired within it.

Table 13-2. Conflicting lock modes

<table>
<thead>
<tr>
<th>Requested Lock Mode</th>
<th>Current Lock Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS SHARE</td>
<td>ROW SHARE</td>
</tr>
<tr>
<td>ROW SHARE</td>
<td>ROW EXCLUSIVE</td>
</tr>
<tr>
<td>SHARE UPDATE</td>
<td>SHARE UPDATE EXCLUSIVE</td>
</tr>
<tr>
<td>SHARE</td>
<td>SHARE ROW EXCLUSIVE</td>
</tr>
<tr>
<td>SHARE ROW EXCLUSIVE</td>
<td>EXCLUSIVE</td>
</tr>
<tr>
<td>ACCESS EXCLUSIVE</td>
<td></td>
</tr>
</tbody>
</table>
### 13.3.2. Row-Level Locks

In addition to table-level locks, there are row-level locks, which can be exclusive or shared locks. An exclusive row-level lock on a specific row is automatically acquired when the row is updated or deleted. The lock is held until the transaction commits or rolls back, in just the same way as for table-level locks. Row-level locks do not affect data querying; they block *writers to the same row only.*

To acquire an exclusive row-level lock on a row without actually modifying the row, select the row with `SELECT FOR UPDATE`. Note that once the row-level lock is acquired, the transaction can update the row multiple times without fear of conflicts.

To acquire a shared row-level lock on a row, select the row with `SELECT FOR SHARE`. A shared lock does not prevent other transactions from acquiring the same shared lock. However, no transaction is allowed to update, delete, or exclusively lock a row on which any other transaction holds a shared lock. Any attempt to do so will block until the shared lock(s) have been released.
Chapter 13. Concurrency Control

PostgreSQL doesn’t remember any information about modified rows in memory, so it has no limit to the number of rows locked at one time. However, locking a row might cause a disk write; thus, for example, SELECT FOR UPDATE will modify selected rows to mark them locked, and so will result in disk writes.

In addition to table and row locks, page-level share/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a row is fetched or updated. Application developers normally need not be concerned with page-level locks, but we mention them for completeness.

13.3.3. Deadlocks

The use of explicit locking can increase the likelihood of deadlocks, wherein two (or more) transactions each hold locks that the other wants. For example, if transaction 1 acquires an exclusive lock on table A and then tries to acquire an exclusive lock on table B, while transaction 2 has already exclusive-locked table B and now wants an exclusive lock on table A, then neither one can proceed. PostgreSQL automatically detects deadlock situations and resolves them by aborting one of the transactions involved, allowing the other(s) to complete. (Exactly which transaction will be aborted is difficult to predict and should not be relied on.)

Note that deadlocks can also occur as the result of row-level locks (and thus, they can occur even if explicit locking is not used). Consider the case in which there are two concurrent transactions modifying a table. The first transaction executes:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

This acquires a row-level lock on the row with the specified account number. Then, the second transaction executes:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

The first UPDATE statement successfully acquires a row-level lock on the specified row, so it succeeds in updating that row. However, the second UPDATE statement finds that the row it is attempting to update has already been locked, so it waits for the transaction that acquired the lock to complete. Transaction two is now waiting on transaction one to complete before it continues execution. Now, transaction one executes:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

Transaction one attempts to acquire a row-level lock on the specified row, but it cannot: transaction two already holds such a lock. So it waits for transaction two to complete. Thus, transaction one is blocked on transaction two, and transaction two is blocked on transaction one: a deadlock condition. PostgreSQL will detect this situation and abort one of the transactions.

The best defense against deadlocks is generally to avoid them by being certain that all applications using a database acquire locks on multiple objects in a consistent order. In the example above, if both transactions had updated the rows in the same order, no deadlock would have occurred. One should also ensure that the first lock acquired on an object in a transaction is the highest mode that will be needed for that object. If it is not feasible to verify this in advance, then deadlocks can be handled on-the-fly by retrying transactions that are aborted due to deadlock.

So long as no deadlock situation is detected, a transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. This means it is a bad idea for applications to hold transactions open for long periods of time (e.g., while waiting for user input).
Chapter 13. Concurrency Control

13.3.4. Advisory Locks

PostgreSQL provides a means for creating locks that have application-defined meanings. These are called advisory locks, because the system does not enforce their use — it is up to the application to use them correctly. Advisory locks can be useful for locking strategies that are an awkward fit for the MVCC model. Once acquired, an advisory lock is held until explicitly released or the session ends. Unlike standard locks, advisory locks do not honor transaction semantics: a lock acquired during a transaction that is later rolled back will still be held following the rollback, and likewise an unlock is effective even if the calling transaction fails later. The same lock can be acquired multiple times by its owning process; for each lock request there must be a corresponding unlock request before the lock is actually released. (If a session already holds a given lock, additional requests will always succeed, even if other sessions are awaiting the lock.) Like all locks in PostgreSQL, a complete list of advisory locks currently held by any session can be found in the pg_locks system view.

Advisory locks are allocated out of a shared memory pool whose size is defined by the configuration variables max_locks_per_transaction and max_connections. Care must be taken not to exhaust this memory or the server will not be able to grant any locks at all. This imposes an upper limit on the number of advisory locks grantable by the server, typically in the tens to hundreds of thousands depending on how the server is configured.

A common use of advisory locks is to emulate pessimistic locking strategies typical of so called “flat file” data management systems. While a flag stored in a table could be used for the same purpose, advisory locks are faster, avoid MVCC bloat, and are automatically cleaned up by the server at the end of the session. In certain cases using this method, especially in queries involving explicit ordering and LIMIT clauses, care must be taken to control the locks acquired because of the order in which SQL expressions are evaluated. For example:

```sql
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!
SELECT pg_advisory_lock(q.id) FROM
  (SELECT id FROM foo WHERE id > 12345 LIMIT 100;
   q;
  ) q; -- ok
```

In the above queries, the second form is dangerous because the LIMIT is not guaranteed to be applied before the locking function is executed. This might cause some locks to be acquired that the application was not expecting, and hence would fail to release (until it ends the session). From the point of view of the application, such locks would be dangling, although still viewable in pg_locks.

The functions provided to manipulate advisory locks are described in Table 9-56.

13.4. Data Consistency Checks at the Application Level

Because readers in PostgreSQL do not lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another concurrent transaction. In other words, if a row is returned by SELECT it doesn’t mean that the row is still current at the instant it is returned (i.e., sometime after the current query began). The row might have been modified or deleted by an already-committed transaction that committed after this one started. Even if the row is still valid “now”, it could be changed or deleted before the current transaction does a commit or rollback.

Another way to think about it is that each transaction sees a snapshot of the database contents, and concurrently executing transactions might very well see different snapshots. So the whole concept of “now” is somewhat ill-defined anyway. This is not normally a big problem if the client applications...
are isolated from each other, but if the clients can communicate via channels outside the database then serious confusion might ensue.

To ensure the current validity of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE`, `SELECT FOR SHARE`, or an appropriate `LOCK TABLE` statement. (`SELECT FOR UPDATE` or `SELECT FOR SHARE` locks just the returned rows against concurrent updates, while `LOCK TABLE` locks the whole table.) This should be taken into account when porting applications to PostgreSQL from other environments.

Global validity checks require extra thought under MVCC. For example, a banking application might wish to check that the sum of all credits in one table equals the sum of debits in another table, when both tables are being actively updated. Comparing the results of two successive `SELECT sum(...)` commands will not work reliably under Read Committed mode, since the second query will likely include the results of transactions not counted by the first. Doing the two sums in a single serializable transaction will give an accurate picture of the effects of transactions that committed before the serializable transaction started — but one might legitimately wonder whether the answer is still relevant by the time it is delivered. If the serializable transaction itself applied some changes before trying to make the consistency check, the usefulness of the check becomes even more debatable, since now it includes some but not all post-transaction-start changes. In such cases a careful person might wish to lock all tables needed for the check, in order to get an indisputable picture of current reality. A `SHARE` mode (or higher) lock guarantees that there are no uncommitted changes in the locked table, other than those of the current transaction.

Note also that if one is relying on explicit locking to prevent concurrent changes, one should use Read Committed mode, or in Serializable mode be careful to obtain the lock(s) before performing queries. A lock obtained by a serializable transaction guarantees that no other transactions modifying the table are still running, but if the snapshot seen by the transaction predates obtaining the lock, it might predate some now-committed changes in the table. A serializable transaction’s snapshot is actually frozen at the start of its first query or data-modification command (`SELECT`, `INSERT`, `UPDATE`, or `DELETE`), so it’s possible to obtain locks explicitly before the snapshot is frozen.

### 13.5. Locking and Indexes

Though PostgreSQL provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in PostgreSQL. The various index types are handled as follows:

**B-tree and GiST indexes**

Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. These index types provide the highest concurrency without deadlock conditions.

**Hash indexes**

Share/exclusive hash-bucket-level locks are used for read/write access. Locks are released after the whole bucket is processed. Bucket-level locks provide better concurrency than index-level ones, but deadlock is possible since the locks are held longer than one index operation.

**GIN indexes**

Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. But note that a GIN-indexed value insertion usually produces several index key insertions per row, so GIN might do substantial work for a single value’s insertion.
Currently, B-tree indexes offer the best performance for concurrent applications; since they also have more features than hash indexes, they are the recommended index type for concurrent applications that need to index scalar data. When dealing with non-scalar data, B-trees are not useful, and GiST or GIN indexes should be used instead.
Chapter 14. Performance Tips

Query performance can be affected by many things. Some of these can be manipulated by the user, while others are fundamental to the underlying design of the system. This chapter provides some hints about understanding and tuning PostgreSQL performance.

14.1. Using EXPLAIN

PostgreSQL devises a query plan for each query it is given. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex planner that tries to select good plans. You can use the EXPLAIN command to see what query plan the planner creates for any query. Plan-reading is an art that deserves an extensive tutorial, which this is not; but here is some basic information.

The structure of a query plan is a tree of plan nodes. Nodes at the bottom level are table scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes “atop” the scan nodes to perform these operations. Again, there is usually more than one possible way to do these operations, so different node types can appear here too. The output of EXPLAIN has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. The first line (topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.

Here is a trivial example, just to show what the output looks like.¹

EXPLAIN SELECT * FROM tenk1;

QUERY PLAN

-------------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)

The numbers that are quoted by EXPLAIN are:

- Estimated start-up cost (Time expended before output scan can start, e.g., time to do the sorting in a sort node.)
- Estimated total cost (If all rows were to be retrieved, though they might not be: for example, a query with a LIMIT clause will stop short of paying the total cost of the Limit plan node’s input node.)
- Estimated number of rows output by this plan node (Again, only if executed to completion.)
- Estimated average width (in bytes) of rows output by this plan node

The costs are measured in arbitrary units determined by the planner’s cost parameters (see Section 18.6.2). Traditional practice is to measure the costs in units of disk page fetches; that is, seq_page_cost is conventionally set to 1.0 and the other cost parameters are set relative to that. The examples in this section are run with the default cost parameters.

¹ Examples in this section are drawn from the regression test database after doing a VACUUM ANALYZE, using 8.2 development sources. You should be able to get similar results if you try the examples yourself, but your estimated costs and row counts will probably vary slightly because ANALYZE’s statistics are random samples rather than being exact.
It’s important to note that the cost of an upper-level node includes the cost of all its child nodes. It’s also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the true elapsed time; but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same row set, we trust.)

Rows output is a little tricky because it is not the number of rows processed or scanned by the plan node. It is usually less, reflecting the estimated selectivity of any WHERE-clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Returning to our example:

```
EXPLAIN SELECT * FROM tenk1;
```

```
QUERY PLAN

Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

This is about as straightforward as it gets. If you do:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

you will find out that tenk1 has 358 disk pages and 10000 rows. The estimated cost is (disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost). By default, seq_page_cost is 1.0 and cpu_tuple_cost is 0.01. So the estimated cost is (358 * 1.0) + (10000 * 0.01) = 458.

Now let’s modify the query to add a WHERE condition:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;
```

```
QUERY PLAN

Seq Scan on tenk1 (cost=0.00..483.00 rows=7033 width=244)
  Filter: (unique1 < 7000)
```

Notice that the EXPLAIN output shows the WHERE clause being applied as a “filter” condition; this means that the plan node checks the condition for each row it scans, and outputs only the ones that pass the condition. The estimate of output rows has gone down because of the WHERE clause. However, the scan will still have to visit all 10000 rows, so the cost hasn’t decreased; in fact it has gone up a bit (by 10000 * cpu_operator_cost, to be exact) to reflect the extra CPU time spent checking the WHERE condition.

The actual number of rows this query would select is 7000, but the rows estimate is only approximate. If you try to duplicate this experiment, you will probably get a slightly different estimate; moreover, it will change after each ANALYZE command, because the statistics produced by ANALYZE are taken from a randomized sample of the table.

Now, let’s make the condition more restrictive:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;
```

```
QUERY PLAN

Bitmap Heap Scan on tenk1 (cost=2.37..232.35 rows=106 width=244)
  Recheck Cond: (unique1 < 100)
  -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0)
```
Chapter 14. Performance Tips

Here the planner has decided to use a two-step plan: the bottom plan node visits an index to find the locations of rows matching the index condition, and then the upper plan node actually fetches those rows from the table itself. Fetching the rows separately is much more expensive than sequentially reading them, but because not all the pages of the table have to be visited, this is still cheaper than a sequential scan. (The reason for using two levels of plan is that the upper plan node sorts the row locations identified by the index into physical order before reading them, so as to minimize the costs of the separate fetches. The “bitmap” mentioned in the node names is the mechanism that does the sorting.)

If the WHERE condition is selective enough, the planner might switch to a “simple” index scan plan:

```sql
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3;
```

```sql
QUERY PLAN
------------------------------------------------------------------------------
Index Scan using tenk1_unique1 on tenk1 (cost=0.00..10.00 rows=2 width=244)
  Index Cond: (unique1 < 3)
In this case the table rows are fetched in index order, which makes them even more expensive to read, but there are so few that the extra cost of sorting the row locations is not worth it. You’ll most often see this plan type for queries that fetch just a single row, and for queries that request an ORDER BY condition that matches the index order.

Add another condition to the WHERE clause:

```sql
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3 AND stringu1 = 'xxx';
```

```sql
QUERY PLAN
------------------------------------------------------------------------------
Index Scan using tenk1_unique1 on tenk1 (cost=0.00..10.01 rows=1 width=244)
  Index Cond: (unique1 < 3)
  Filter: (stringu1 = 'xxx'::name)
The added condition stringu1 = 'xxx' reduces the output-rows estimate, but not the cost because we still have to visit the same set of rows. Notice that the stringu1 clause cannot be applied as an index condition (since this index is only on the unique1 column). Instead it is applied as a filter on the rows retrieved by the index. Thus the cost has actually gone up a little bit to reflect this extra checking.

If there are indexes on several columns used in WHERE, the planner might choose to use an AND or OR combination of the indexes:

```sql
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;
```

```sql
QUERY PLAN
------------------------------------------------------------------------------
Bitmap Heap Scan on tenk1 (cost=11.27..49.11 rows=11 width=244)
  Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
  -> BitmapAnd (cost=11.27..11.27 rows=11 width=0)
    -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0)
      Index Cond: (unique1 < 100)
    -> Bitmap Index Scan on tenk1_unique2 (cost=0.00..8.65 rows=1042 width=0)
      Index Cond: (unique2 > 9000)
Chapter 14. Performance Tips

But this requires visiting both indexes, so it’s not necessarily a win compared to using just one index and treating the other condition as a filter. If you vary the ranges involved you’ll see the plan change accordingly.

Let’s try joining two tables, using the columns we have been discussing:

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

**QUERY PLAN**

```
--------------------------------------------------------------------------------------
| Nested Loop (cost=2.37..553.11 rows=106 width=488) |
| -> Bitmap Heap Scan on tenk1 t1 (cost=2.37..232.35 rows=106 width=244) |
| | Recheck Cond: (unique1 < 100) |
| | -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0) |
| | | Index Cond: (unique1 < 100) |
| | -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.00..3.01 rows=1 width=244) |
| | | Index Cond: (t2.unique2 = t1.unique2) |
--------------------------------------------------------------------------------------
```

In this nested-loop join, the outer scan is the same bitmap index scan we saw earlier, and so its cost and row count are the same because we are applying the `WHERE` clause `unique1 < 100` at that node. The `t1.unique2 = t2.unique2` clause is not relevant yet, so it doesn’t affect row count of the outer scan. For the inner scan, the `unique2` value of the current outer-scan row is plugged into the inner index scan to produce an index condition like `t2.unique2 = constant`. So we get the same inner-scan plan and costs that we’d get from, say, `EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42`. The costs of the loop node are then set on the basis of the cost of the outer scan, plus one repetition of the inner scan for each outer row (106 * 3.01, here), plus a little CPU time for join processing.

In this example the join’s output row count is the same as the product of the two scans’ row counts, but that’s not true in general, because in general you can have `WHERE` clauses that mention both tables and so can only be applied at the join point, not to either input scan. For example, if we added `WHERE ... AND t1.hundred < t2.hundred`, that would decrease the output row count of the join node, but not change either input scan.

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the winner, using the enable/disable flags described in Section 18.6.1. (This is a crude tool, but useful. See also Section 14.3.)

```
SET enable_nestloop = off;
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

**QUERY PLAN**

```
------------------------------------------------------------------------------------------
| Hash Join (cost=232.61..741.67 rows=106 width=488) |
| Hash Cond: (t2.unique2 = t1.unique2) |
| -> Seq Scan on tenk2 t2 (cost=0.00..458.00 rows=10000 width=244) |
| -> Hash (cost=232.35..232.35 rows=106 width=244) |
| | -> Bitmap Heap Scan on tenk1 t1 (cost=2.37..232.35 rows=106 width=244) |
| | | Recheck Cond: (unique1 < 100) |
| | | -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0) |
| | | | Index Cond: (unique1 < 100) |
------------------------------------------------------------------------------------------
```

This plan proposes to extract the 100 interesting rows of `tenk1` using that same old index scan, stash them into an in-memory hash table, and then do a sequential scan of `tenk2`, probing into the hash table for possible matches of `t1.unique2 = t2.unique2` at each `tenk2` row. The cost to read...
tenk1 and set up the hash table is entirely start-up cost for the hash join, since we won’t get any rows out until we can start reading tenk2. The total time estimate for the join also includes a hefty charge for the CPU time to probe the hash table 10000 times. Note, however, that we are not charging 10000 times 232.35; the hash table setup is only done once in this plan type.

It is possible to check on the accuracy of the planner’s estimated costs by using EXPLAIN ANALYZE. This command actually executes the query, and then displays the true run time accumulated within each plan node along with the same estimated costs that a plain EXPLAIN shows. For example, we might get a result like this:

```sql
EXPLAIN ANALYZE SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

```
QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------
Nested Loop  (cost=2.37..553.11 rows=106 width=488) (actual time=1.392..12.700 rows=100 loops=1)
  ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244) (actual time=0.878..2.367 rows=100 loops=1)
      Recheck Cond: (unique1 < 100)
  ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0) (actual time=0.546..0.546 rows=100 loops=1)
      Index Cond: (unique1 < 100)
  ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=244) (actual time=0.067..0.078 rows=1 loops=100)
      Index Cond: (t2.unique2 = t1.unique2)
Total runtime: 14.452 ms
```

Note that the “actual time” values are in milliseconds of real time, whereas the “cost” estimates are expressed in arbitrary units; so they are unlikely to match up. The thing to pay attention to is whether the ratios of actual time and estimated costs are consistent.

In some query plans, it is possible for a subplan node to be executed more than once. For example, the inner index scan is executed once per outer row in the above nested-loop plan. In such cases, the “loops” value reports the total number of executions of the node, and the actual time and rows values shown are averages per-execution. This is done to make the numbers comparable with the way that the cost estimates are shown. Multiply by the “loops” value to get the total time actually spent in the node.

The Total runtime shown by EXPLAIN ANALYZE includes executor start-up and shut-down time, as well as time spent processing the result rows. It does not include parsing, rewriting, or planning time. For a SELECT query, the total run time will normally be just a little larger than the total time reported for the top-level plan node. For INSERT, UPDATE, and DELETE commands, the total run time might be considerably larger, because it includes the time spent processing the result rows. In these commands, the time for the top plan node essentially is the time spent computing the new rows and/or locating the old ones, but it doesn’t include the time spent applying the changes. Time spent firing triggers, if any, is also outside the top plan node, and is shown separately for each trigger.

It is worth noting that EXPLAIN results should not be extrapolated to situations other than the one you are actually testing; for example, results on a toy-sized table cannot be assumed to apply to large tables. The planner’s cost estimates are not linear and so it might choose a different plan for a larger or smaller table. An extreme example is that on a table that only occupies one disk page, you’ll nearly always get a sequential scan plan whether indexes are available or not. The planner realizes that it’s going to take one disk page read to process the table in any case, so there’s no value in expending additional page reads to look at an index.
14.2. Statistics Used by the Planner

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. This section provides a quick look at the statistics that the system uses for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table `pg_class`, in the columns `reltuples` and `relpages`. We can look at it with queries similar to this one:

```
SELECT relname, relkind, reltuples, relpages FROM pg_class WHERE relname LIKE 'tenk1%';
```

<table>
<thead>
<tr>
<th>relname</th>
<th>relkind</th>
<th>reltuples</th>
<th>relpages</th>
</tr>
</thead>
<tbody>
<tr>
<td>tenk1</td>
<td>r</td>
<td>10000</td>
<td>358</td>
</tr>
<tr>
<td>tenk1_hundred</td>
<td>i</td>
<td>10000</td>
<td>30</td>
</tr>
<tr>
<td>tenk1_thous_tenthous</td>
<td>i</td>
<td>10000</td>
<td>30</td>
</tr>
<tr>
<td>tenk1_unique1</td>
<td>i</td>
<td>10000</td>
<td>30</td>
</tr>
<tr>
<td>tenk1_unique2</td>
<td>i</td>
<td>10000</td>
<td>30</td>
</tr>
</tbody>
</table>

Here we can see that `tenk1` contains 10000 rows, as do its indexes, but the indexes are (unsurprisingly) much smaller than the table.

For efficiency reasons, `reltuples` and `relpages` are not updated on-the-fly, and so they usually contain somewhat out-of-date values. They are updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`. A stand-alone `ANALYZE`, that is one not part of `VACUUM`, generates an approximate `reltuples` value since it does not read every row of the table. The planner will scale the values it finds in `pg_class` to match the current physical table size, thus obtaining a closer approximation.

Most queries retrieve only a fraction of the rows in a table, due to having `WHERE` clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the selectivity of `WHERE` clauses, that is, the fraction of rows that match each condition in the `WHERE` clause. The information used for this task is stored in the `pg_statistic` system catalog. Entries in `pg_statistic` are updated by the `ANALYZE` and `VACUUM ANALYZE` commands, and are always approximate even when freshly updated.

Rather than look at `pg_statistic` directly, it’s better to look at its view `pg_stats` when examining the statistics manually. `pg_stats` is designed to be more easily readable. Furthermore, `pg_stats` is readable by all, whereas `pg_statistic` is only readable by a superuser. (This prevents unprivileged users from learning something about the contents of other people’s tables from the statistics. The `pg_stats` view is restricted to show only rows about tables that the current user can read.) For example, we might do:

```
SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE tablename = 'road';
```

<table>
<thead>
<tr>
<th>attname</th>
<th>n_distinct</th>
<th>most_common_vals</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>-0.467008</td>
<td>(&quot;I- 580 Ramp&quot;,&quot;I- 880 Ramp&quot;,&quot;I- 580&quot;,&quot;I- 680 Ramp&quot;,&quot;I- 80 Ramp&quot;,&quot;14th St&quot;,&quot;5th St&quot;,&quot;Mission Blvd&quot;,&quot;I- 880&quot;)</td>
</tr>
<tr>
<td>thepath</td>
<td>20</td>
<td>(&quot;([-122.089,37.71),(-122.088,37.711)]&quot;)</td>
</tr>
</tbody>
</table>

(2 rows)
The amount of information stored in `pg_statistic` by `ANALYZE`, in particular the maximum number of entries in the `most_common_vals` and `histogram_bounds` arrays for each column, can be set on a column-by-column basis using the `ALTER TABLE SET STATISTICS` command, or globally by setting the `default_statistics_target` configuration variable. The default limit is presently 10 entries. Raising the limit might allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in `pg_statistic` and slightly more time to compute the estimates. Conversely, a lower limit might be sufficient for columns with simple data distributions.

Further details about the planner’s use of statistics can be found in Chapter 55.

### 14.3. Controlling the Planner with Explicit `JOIN` Clauses

It is possible to control the query planner to some extent by using the explicit `JOIN` syntax. To see why this matters, we first need some background.

In a simple join query, such as:

```sql
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

the planner is free to join the given tables in any order. For example, it could generate a query plan that joins A to B, using the `WHERE` condition `a.id = b.id`, and then joins C to this joined table, using the other `WHERE` condition. Or it could join B to C and then join A to that result. Or it could join A to C and then join them with B — but that would be inefficient, since the full Cartesian product of A and C would have to be formed, there being no applicable condition in the `WHERE` clause to allow optimization of the join. (All joins in the PostgreSQL executor happen between two input tables, so it’s necessary to build up the result in one or another of these fashions.) The important point is that these different join possibilities give semantically equivalent results but might have hugely different execution costs. Therefore, the planner will explore all of them to try to find the most efficient query plan.

When a query only involves two or three tables, there aren’t many join orders to worry about. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it’s no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning might take an annoyingly long time. When there are too many input tables, the PostgreSQL planner will switch from exhaustive search to a genetic probabilistic search through a limited number of possibilities. (The switch-over threshold is set by the `geqo_threshold` run-time parameter.) The genetic search takes less time, but it won’t necessarily find the best possible plan.

When the query involves outer joins, the planner has less freedom than it does for plain (inner) joins. For example, consider:

```sql
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Although this query’s restrictions are superficially similar to the previous example, the semantics are different because a row must be emitted for each row of A that has no matching row in the join of B and C. Therefore the planner has no choice of join order here: it must join B to C and then join A to that result. Accordingly, this query takes less time to plan than the previous query. In other cases, the planner might be able to determine that more than one join order is safe. For example, given:

```sql
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

it is valid to join A to either B or C first. Currently, only `FULL JOIN` completely constrains the join order. Most practical cases involving `LEFT JOIN` or `RIGHT JOIN` can be rearranged to some extent.
Explicit inner join syntax (INNER JOIN, CROSS JOIN, or unadorned JOIN) is semantically the same as listing the input relations in FROM, so it does not constrain the join order.

Even though most kinds of JOIN don’t completely constrain the join order, it is possible to instruct the PostgreSQL query planner to treat all JOIN clauses as constraining the join order anyway. For example, these three queries are logically equivalent:

```sql
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

But if we tell the planner to honor the JOIN order, the second and third take less time to plan than the first. This effect is not worth worrying about for only three tables, but it can be a lifesaver with many tables.

To force the planner to follow the join order laid out by explicit JOINs, set the join_collapse_limit run-time parameter to 1. (Other possible values are discussed below.)

You do not need to constrain the join order completely in order to cut search time, because it’s OK to use JOIN operators within items of a plain FROM list. For example, consider:

```sql
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...
```

With join_collapse_limit = 1, this forces the planner to join A to B before joining them to other tables, but doesn’t constrain its choices otherwise. In this example, the number of possible join orders is reduced by a factor of 5.

Constraining the planner’s search in this way is a useful technique both for reducing planning time and for directing the planner to a good query plan. If the planner chooses a bad join order by default, you can force it to choose a better order via JOIN syntax — assuming that you know of a better order, that is. Experimentation is recommended.

A closely related issue that affects planning time is collapsing of subqueries into their parent query. For example, consider:

```sql
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

This situation might arise from use of a view that contains a join; the view’s SELECT rule will be inserted in place of the view reference, yielding a query much like the above. Normally, the planner will try to collapse the subquery into the parent, yielding:

```sql
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

This usually results in a better plan than planning the subquery separately. (For example, the outer WHERE conditions might be such that joining X to A first eliminates many rows of A, thus avoiding the need to form the full logical output of the subquery.) But at the same time, we have increased the planning time; here, we have a five-way join problem replacing two separate three-way join problems. Because of the exponential growth of the number of possibilities, this makes a big difference. The planner tries to avoid getting stuck in huge join search problems by not collapsing a subquery if more than fromCollapse_limit FROM items would result in the parent query. You can trade off planning time against quality of plan by adjusting this run-time parameter up or down.

fromCollapse_limit and join_collapse_limit are similarly named because they do almost the same thing: one controls when the planner will “flatten out” subqueries, and the other controls when it will flatten out explicit joins. Typically you would either set join_collapse_limit
equal to `fromCollapseLimit` (so that explicit joins and subqueries act similarly) or set `join CollapseLimit` to 1 (if you want to control join order with explicit joins). But you might set them differently if you are trying to fine-tune the trade-off between planning time and run time.

### 14.4. Populating a Database

One might need to insert a large amount of data when first populating a database. This section contains some suggestions on how to make this process as efficient as possible.

#### 14.4.1. Disable Autocommit

Turn off autocommit and just do one commit at the end. (In plain SQL, this means issuing `BEGIN` at the start and `COMMIT` at the end. Some client libraries might do this behind your back, in which case you need to make sure the library does it when you want it done.) If you allow each insertion to be committed separately, PostgreSQL is doing a lot of work for each row that is added. An additional benefit of doing all insertions in one transaction is that if the insertion of one row were to fail then the insertion of all rows inserted up to that point would be rolled back, so you won’t be stuck with partially loaded data.

#### 14.4.2. Use `COPY`

Use `COPY` to load all the rows in one command, instead of using a series of `INSERT` commands. The `COPY` command is optimized for loading large numbers of rows; it is less flexible than `INSERT`, but incurs significantly less overhead for large data loads. Since `COPY` is a single command, there is no need to disable autocommit if you use this method to populate a table.

If you cannot use `COPY`, it might help to use `PREPARE` to create a prepared `INSERT` statement, and then use `EXECUTE` as many times as required. This avoids some of the overhead of repeatedly parsing and planning `INSERT`. Different interfaces provide this facility in different ways; look for “prepared statements” in the interface documentation.

Note that loading a large number of rows using `COPY` is almost always faster than using `INSERT`, even if `PREPARE` is used and multiple insertions are batched into a single transaction.

`COPY` is fastest when used within the same transaction as an earlier `CREATE TABLE` or `TRUNCATE` command. In such cases no WAL needs to be written, because in case of an error, the files containing the newly loaded data will be removed anyway. However, this consideration does not apply when `archive_mode` is set, as all commands must write WAL in that case.

#### 14.4.3. Remove Indexes

If you are loading a freshly created table, the fastest way is to create the table, bulk load the table’s data using `COPY`, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each row is loaded.

If you are adding large amounts of data to an existing table, it might be a win to drop the index, load the table, and then recreate the index. Of course, the database performance for other users might be adversely affected during the time that the index is missing. One should also think twice before dropping unique indexes, since the error checking afforded by the unique constraint will be lost while the index is missing.
14.4.4. Remove Foreign Key Constraints

Just as with indexes, a foreign key constraint can be checked “in bulk” more efficiently than row-by-row. So it might be useful to drop foreign key constraints, load data, and re-create the constraints. Again, there is a trade-off between data load speed and loss of error checking while the constraint is missing.

14.4.5. Increase maintenance_work_mem

Temporarily increasing the maintenance_work_mem configuration variable when loading large amounts of data can lead to improved performance. This will help to speed up CREATE INDEX commands and ALTER TABLE ADD FOREIGN KEY commands. It won’t do much for COPY itself, so this advice is only useful when you are using one or both of the above techniques.

14.4.6. Increase checkpoint_segments

Temporarily increasing the checkpoint_segments configuration variable can also make large data loads faster. This is because loading a large amount of data into PostgreSQL will cause checkpoints to occur more often than the normal checkpoint frequency (specified by the checkpoint_timeout configuration variable). Whenever a checkpoint occurs, all dirty pages must be flushed to disk. By increasing checkpoint_segments temporarily during bulk data loads, the number of checkpoints that are required can be reduced.

14.4.7. Turn off archive_mode

When loading large amounts of data into an installation that uses WAL archiving, you might want to disable archiving (turn off the archive_mode configuration variable) while loading. It might be faster to take a new base backup after the load has completed than to process a large amount of incremental WAL data. But note that turning archive_mode on or off requires a server restart.

Aside from avoiding the time for the archiver to process the WAL data, doing this will actually make certain commands faster, because they are designed not to write WAL at all if archive_mode is off. (They can guarantee crash safety more cheaply by doing an fsync at the end than by writing WAL.) This applies to the following commands:

• CREATE TABLE AS SELECT
• CREATE INDEX (and variants such as ALTER TABLE ADD PRIMARY KEY)
• ALTER TABLE SET TABLESPACE
• CLUSTER
• COPY FROM, when the target table has been created or truncated earlier in the same transaction

14.4.8. Run ANALYZE Afterwards

Whenever you have significantly altered the distribution of data within a table, running ANALYZE is strongly recommended. This includes bulk loading large amounts of data into the table. Running ANALYZE (or VACUUM ANALYZE) ensures that the planner has up-to-date statistics about the table.
Chapter 14. Performance Tips

With no statistics or obsolete statistics, the planner might make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics.

14.4.9. Some Notes About pg_dump

Dump scripts generated by pg_dump automatically apply several, but not all, of the above guidelines. To reload a pg_dump dump as quickly as possible, you need to do a few extra things manually. (Note that these points apply while restoring a dump, not while creating it. The same points apply when using pg_restore to load from a pg_dump archive file.)

By default, pg_dump uses COPY, and when it is generating a complete schema-and-data dump, it is careful to load data before creating indexes and foreign keys. So in this case several guidelines are handled automatically. What is left for you to do is to:

- Set appropriate (i.e., larger than normal) values for maintenance_work_mem and checkpoint_segments.
- If using WAL archiving, consider disabling it during the restore. To do that, turn off archive_mode before loading the dump script, and afterwards turn it back on and take a fresh base backup.
- Consider whether the whole dump should be restored as a single transaction. To do that, pass the -1 or --single-transaction command-line option to psql or pg_restore. When using this mode, even the smallest of errors will rollback the entire restore, possibly discarding many hours of processing. Depending on how interrelated the data is, that might seem preferable to manual cleanup, or not. COPY commands will run fastest if you use a single transaction and have WAL archiving turned off.
- Run ANALYZE afterwards.

A data-only dump will still use COPY, but it does not drop or recreate indexes, and it does not normally touch foreign keys. So when loading a data-only dump, it is up to you to drop and recreate indexes and foreign keys if you wish to use those techniques. It’s still useful to increase checkpoint_segments while loading the data, but don’t bother increasing maintenance_work_mem; rather, you’d do that while manually recreating indexes and foreign keys afterwards. And don’t forget to ANALYZE when you’re done.

2. You can get the effect of disabling foreign keys by using the --disable-triggers option — but realize that that eliminates, rather than just postpones, foreign key validation, and so it is possible to insert bad data if you use it.
III. Server Administration

This part covers topics that are of interest to a PostgreSQL database administrator. This includes installation of the software, set up and configuration of the server, management of users and databases, and maintenance tasks. Anyone who runs a PostgreSQL server, even for personal use, but especially in production, should be familiar with the topics covered in this part.

The information in this part is arranged approximately in the order in which a new user should read it. But the chapters are self-contained and can be read individually as desired. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should look into Part VI.

The first few chapters are written so that they can be understood without prerequisite knowledge, so that new users who need to set up their own server can begin their exploration with this part. The rest of this part is about tuning and management; that material assumes that the reader is familiar with the general use of the PostgreSQL database system. Readers are encouraged to look at Part I and Part II for additional information.
Chapter 15. Installation Instructions

This chapter describes the installation of PostgreSQL from the source code distribution. (If you are installing a pre-packaged distribution, such as an RPM or Debian package, ignore this chapter and read the packager’s instructions instead.)

15.1. Short Version

./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test

The long version is the rest of this chapter.

15.2. Requirements

In general, a modern Unix-compatible platform should be able to run PostgreSQL. The platforms that had received specific testing at the time of release are listed in Section 15.7 below. In the doc subdirectory of the distribution there are several platform-specific FAQ documents you might wish to consult if you are having trouble.

The following software packages are required for building PostgreSQL:

- GNU make is required; other make programs will not work. GNU make is often installed under the name gmake; this document will always refer to it by that name. (On some systems GNU make is the default tool with the name make.) To test for GNU make enter

  gmake --version

  It is recommended to use version 3.76.1 or later.

- You need an ISO/ANSI C compiler. Recent versions of GCC are recommendable, but PostgreSQL is known to build with a wide variety of compilers from different vendors.

- tar is required to unpack the source distribution in the first place, in addition to either gzip or bzip2.

- The GNU Readline library (for simple line editing and command history retrieval) is used by default. If you don’t want to use it then you must specify the --without-readline option for configure. As an alternative, you can often use the BSD-licensed libedit library, originally developed on NetBSD. The libedit library is GNU Readline-compatible and is used if libreadline is not found, or if --with-libedit-preferred is used as an option to configure. If you are using a package-based Linux distribution, be aware that you need both the readline and readline-devel packages, if those are separate in your distribution.
Chapter 15. Installation Instructions

- The zlib compression library will be used by default. If you don’t want to use it then you must specify the `--without-zlib` option for `configure`. Using this option disables support for compressed archives in `pg_dump` and `pg_restore`.

The following packages are optional. They are not required in the default configuration, but they are needed when certain build options are enabled, as explained below.

- To build the server programming language PL/Perl you need a full Perl installation, including the `libperl` library and the header files. Since PL/Perl will be a shared library, the `libperl` library must be a shared library also on most platforms. This appears to be the default in recent Perl versions, but it was not in earlier versions, and in any case it is the choice of whomever installed Perl at your site. If you intend to make more than incidental use of PL/Perl, you should ensure that the Perl installation was built with the `usemultiplicity` option enabled (`perl -V` will show whether this is the case).

  If you don’t have the shared library but you need one, a message like this will appear during the build to point out this fact:

  ```
  *** Cannot build PL/Perl because libperl is not a shared library.
  *** You might have to rebuild your Perl installation. Refer to
  *** the documentation for details.
  (If you don’t follow the on-screen output you will merely notice that the PL/Perl library object, `piperl.so` or similar, will not be installed.) If you see this, you will have to rebuild and install Perl manually to be able to build PL/Perl. During the configuration process for Perl, request a shared library.
  ```

- To build the PL/Python server programming language, you need a Python installation with the header files and the distutils module. The distutils module is included by default with Python 1.6 and later; users of earlier versions of Python will need to install it. Since PL/Python will be a shared library, the `libpython` library must be a shared library also on most platforms. This is not the case in a default Python installation. If after building and installing you have a file called `plpython.so` (possibly a different extension), then everything went well. Otherwise you should have seen a notice like this flying by:

  ```
  *** Cannot build PL/Python because libpython is not a shared library.
  *** You might have to rebuild your Python installation. Refer to
  *** the documentation for details.
  That means you have to rebuild (part of) your Python installation to supply this shared library.
  ```

  If you have problems, run Python 2.3 or later’s `configure` using the `--enable-shared` flag. On some operating systems you don’t have to build a shared library, but you will have to convince the PostgreSQL build system of this. Consult the `Makefile` in the `src/pl/plpython` directory for details.

- If you want to build the PL/Tcl procedural language, you of course need a Tcl installation. If you are using a pre-8.4 release of Tcl, ensure that it was built without multithreading support.

- To enable Native Language Support (NLS), that is, the ability to display a program’s messages in a language other than English, you need an implementation of the Gettext API. Some operating systems have this built-in (e.g., Linux, NetBSD, Solaris), for other systems you can download an add-on package from `http://developer.postgresql.org/~petere/bsd-gettext/`. If you are using the Gettext implementation in the GNU C library then you will additionally need the GNU Gettext package for some utility programs. For any of the other implementations you will not need it.

- Kerberos, OpenSSL, OpenLDAP, and/or PAM, if you want to support authentication or encryption using these services.
If you are building from a Git tree instead of using a released source package, or if you want to do server development, you also need the following packages:

- GNU Flex and Bison are needed to build from a Git checkout, or if you changed the actual scanner and parser definition files. If you need them, be sure to get Flex 2.5.4 or later and Bison 1.875 or later. Other yacc programs can sometimes be used, but doing so requires extra effort and is not recommended. Other lex programs will definitely not work.

If you need to get a GNU package, you can find it at your local GNU mirror site (see http://www.gnu.org/order/ftp.html for a list) or at ftp://ftp.gnu.org/gnu/.

Also check that you have sufficient disk space. You will need about 65 MB for the source tree during compilation and about 15 MB for the installation directory. An empty database cluster takes about 25 MB, databases take about five times the amount of space that a flat text file with the same data would take. If you are going to run the regression tests you will temporarily need up to an extra 90 MB. Use the `df` command to check free disk space.

### 15.3. Getting The Source

The PostgreSQL 8.3.23 sources can be obtained by anonymous FTP from ftp://ftp.postgresql.org/pub/source/v8.3.23/postgresql-8.3.23.tar.gz. Other download options can be found on our website: http://www.postgresql.org/download/. After you have obtained the file, unpack it:

```
gunzip postgresql-8.3.23.tar.gz
```

```
tar xf postgresql-8.3.23.tar
```

This will create a directory `postgresql-8.3.23` under the current directory with the PostgreSQL sources. Change into that directory for the rest of the installation procedure.

### 15.4. Upgrading

These instructions assume that your existing installation is under the `/usr/local/pgsql` directory, and that the data area is in `/usr/local/pgsql/data`. Substitute your paths appropriately.

The internal data storage format typically changes in every major release of PostgreSQL. Therefore, if you are upgrading an existing installation that does not have a version number of “8.3.x”, you must back up and restore your data. If you are upgrading from PostgreSQL “8.3.x”, the new version can use your current data files so you should skip the backup and restore steps below because they are unnecessary.

1. If making a backup, make sure that your database is not being updated. This does not affect the integrity of the backup, but the changed data would of course not be included. If necessary, edit the permissions in the file `/usr/local/pgsql/data/pg_hba.conf` (or equivalent) to disallow access from everyone except you.

To back up your database installation, type:

```
pg_dumpall > outputfile
```
Chapter 15. Installation Instructions

If you need to preserve OIDs (such as when using them as foreign keys), then use the -o option when running pg_dumpall.

To make the backup, you can use the pg_dumpall command from the version you are currently running. For best results, however, try to use the pg_dumpall command from PostgreSQL 8.3.23, since this version contains bug fixes and improvements over older versions. While this advice might seem idiosyncratic since you haven’t installed the new version yet, it is advisable to follow it if you plan to install the new version in parallel with the old version. In that case you can complete the installation normally and transfer the data later. This will also decrease the downtime.

2. Shut down the old server:
   
   **pg_ctl stop**
   
   On systems that have PostgreSQL started at boot time, there is probably a start-up file that will accomplish the same thing. For example, on a Red Hat Linux system one might find that
   
   `/etc/rc.d/init.d/postgresql stop`
   
   works.

3. If restoring from backup, rename or delete the old installation directory. It is a good idea to rename the directory, rather than delete it, in case you have trouble and need to revert to it. Keep in mind the directory might consume significant disk space. To rename the directory, use a command like this:

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

4. Install the new version of PostgreSQL as outlined in Section 15.5.

5. Create a new database cluster if needed. Remember that you must execute these commands while logged in to the special database user account (which you already have if you are upgrading).

   ```
   /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
   ```

6. Restore your previous `pg_hba.conf` and any `postgresql.conf` modifications.

7. Start the database server, again from the special database user account:

   ```
   /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
   ```

8. Finally, restore your data from backup with

   ```
   /usr/local/pgsql/bin/psql -d postgres -f outputfile
   ```

   using the new `psql`.

Further discussion appears in Section 24.5, including instructions on how the previous installation can continue running while the new installation is installed.

### 15.5. Installation Procedure

1. Configuration

   The first step of the installation procedure is to configure the source tree for your system and choose the options you would like. This is done by running the `configure` script. For a default installation simply enter

   ```
   ./configure
   ```

   This script will run a number of tests to guess values for various system dependent variables and detect some quirks of your operating system, and finally will create several files in the build tree.
Chapter 15. Installation Instructions

to record what it found. (You can also run configure in a directory outside the source tree if you want to keep the build directory separate.)

The default configuration will build the server and utilities, as well as all client applications and interfaces that require only a C compiler. All files will be installed under /usr/local/pgsql by default.

You can customize the build and installation process by supplying one or more of the following command line options to configure:

```
--prefix=PREFIX
```

Install all files under the directory PREFIX instead of /usr/local/pgsql. The actual files will be installed into various subdirectories; no files will ever be installed directly into the PREFIX directory.

If you have special needs, you can also customize the individual subdirectories with the following options. However, if you leave these with their defaults, the installation will be relocatable, meaning you can move the directory after installation. (The man and doc locations are not affected by this.)

For relocatable installs, you might want to use configure’s --disable-rpath option. Also, you will need to tell the operating system how to find the shared libraries.

```
--exec-prefix=EXEC-PREFIX
```

You can install architecture-dependent files under a different prefix, EXEC-PREFIX, than what PREFIX was set to. This can be useful to share architecture-independent files between hosts. If you omit this, then EXEC-PREFIX is set equal to PREFIX and both architecture-dependent and independent files will be installed under the same tree, which is probably what you want.

```
--bindir=DIRECTORY
```

Specifies the directory for executable programs. The default is EXEC-PREFIX/bin, which normally means /usr/local/pgsql/bin.

```
--datadir=DIRECTORY
```

Sets the directory for read-only data files used by the installed programs. The default is PREFIX/share. Note that this has nothing to do with where your database files will be placed.

```
--sysconfdir=DIRECTORY
```

The directory for various configuration files, PREFIX/etc by default.

```
--libdir=DIRECTORY
```

The location to install libraries and dynamically loadable modules. The default is EXEC-PREFIX/lib.

```
--includedir=DIRECTORY
```

The directory for installing C and C++ header files. The default is PREFIX/include.

```
--mandir=DIRECTORY
```

The man pages that come with PostgreSQL will be installed under this directory, in their respective manx subdirectories. The default is PREFIX/man.
**--with-docdir=DIRECTORY**

Documentation files, except "man" pages, will be installed into this directory. The default is `PREFIX/doc`. If the option `--without-docdir` is specified, the documentation will not be installed by `make install`. This is intended for packaging scripts that have special methods for installing documentation.

**Note:** Care has been taken to make it possible to install PostgreSQL into shared installation locations (such as `/usr/local/include`) without interfering with the namespace of the rest of the system. First, the string "/postgresql" is automatically appended to `datadir`, `sysconfdir`, and `docdir`, unless the fully expanded directory name already contains the string "postgresql" or "pgsql". For example, if you choose `/usr/local` as prefix, the documentation will be installed in `/usr/local/doc/postgresql`, but if the prefix is `/opt/postgres`, then it will be in `/opt/postgres/doc`. The public C header files of the client interfaces are installed into `includedir` and are namespace-clean. The internal header files and the server header files are installed into private directories under `includedir`. See the documentation of each interface for information about how to get at the its header files. Finally, a private subdirectory will also be created, if appropriate, under `libdir` for dynamically loadable modules.

**--with-includes=DIRECTORIES**

`DIRECTORIES` is a colon-separated list of directories that will be added to the list the compiler searches for header files. If you have optional packages (such as GNU Readline) installed in a non-standard location, you have to use this option and probably also the corresponding `--with-libraries` option.

Example: `--with-includes=/opt/gnu/include:/usr/sup/include`.

**--with-libraries=DIRECTORIES**

`DIRECTORIES` is a colon-separated list of directories to search for libraries. You will probably have to use this option (and the corresponding `--with-includes` option) if you have packages installed in non-standard locations.

Example: `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

**--enable-nls[=LANGUAGES]**

Enables Native Language Support (NLS), that is, the ability to display a program’s messages in a language other than English. `LANGUAGES` is a space-separated list of codes of the languages that you want supported, for example `--enable-nls='de fr'`. (The intersection between your list and the set of actually provided translations will be computed automatically.) If you do not specify a list, then all available translations are installed.

To use this option, you will need an implementation of the Gettext API; see above.

**--with-pgport=NUMBER**

Set `NUMBER` as the default port number for server and clients. The default is 5432. The port can always be changed later on, but if you specify it here then both server and clients will have the same default compiled in, which can be very convenient. Usually the only good reason to select a non-default value is if you intend to run multiple PostgreSQL servers on the same machine.
Chapter 15. Installation Instructions

--with-perl
    Build the PL/Perl server-side language.

--with-python
    Build the PL/Python server-side language.

--with-tcl
    Build the PL/Tcl server-side language.

--with-tclconfig=DIRECTORY
    Tcl installs the file tclConfig.sh, which contains configuration information needed to
    build modules interfacing to Tcl. This file is normally found automatically at a well-known
    location, but if you want to use a different version of Tcl you can specify the directory in
    which to look for it.

--with-gssapi
    Build with support for GSSAPI authentication. On many systems, the GSSAPI (usually
    a part of the Kerberos installation) system is not installed in a location that is searched by
    default (e.g., /usr/include, /usr/lib), so you must use the options --with-includes
    and --with-libraries in addition to this option. configure will check for the required
    header files and libraries to make sure that your GSSAPI installation is sufficient before
    proceeding.

--with-krb5
    Build with support for Kerberos 5 authentication. On many systems, the Kerberos system is
    not installed in a location that is searched by default (e.g., /usr/include, /usr/lib), so
    you must use the options --with-includes and --with-libraries in addition to this
    option. configure will check for the required header files and libraries to make sure that
    your Kerberos installation is sufficient before proceeding.

--with-krb-srvnam=NAME
    The default name of the Kerberos service principal (also used by GSSAPI). postgres is the
    default. There’s usually no reason to change this unless you have a Windows environment,
    in which case it must be set to uppercase POSTGRES.

--with-openssl
    Build with support for SSL (encrypted) connections. This requires the OpenSSL package to
    be installed. configure will check for the required header files and libraries to make sure
    that your OpenSSL installation is sufficient before proceeding.

--with-pam
    Build with PAM (Pluggable Authentication Modules) support.

--with-ldap
    Build with LDAP support for authentication and connection parameter lookup (see Section
    30.15 and Section 21.2.7 for more information). On Unix, this requires the OpenLDAP
    package to be installed. configure will check for the required header files and libraries to
    make sure that your OpenLDAP installation is sufficient before proceeding. On Windows,
    the default WinLDAP library is used.

--without-readline
    Prevents use of the Readline library (and libedit as well). This option disables command-line
    editing and history in psql, so it is not recommended.
--with-libedit-preferred
Favors the use of the BSD-licensed libedit library rather than GPL-licensed Readline. This option is significant only if you have both libraries installed; the default in that case is to use Readline.

--with-bonjour
Build with Bonjour support. This requires Bonjour support in your operating system. Recommended on Mac OS X.

--with-ossp-uuid
Use the OSSP UUID library\(^1\) when building contrib/uuid-oss. The library provides functions to generate UUIDs.

--with-libxml
Build with libxml (enables SQL/XML support). Libxml version 2.6.23 or later is required for this feature.

Libxml installs a program xml2-config that can be used to detect the required compiler and linker options. PostgreSQL will use it automatically if found. To specify a libxml installation at an unusual location, you can either set the environment variable XML2_CONFIG to point to the xml2-config program belonging to the installation, or use the options --with-includes and --with-libraries.

--with-libxslt
Use libxslt when building contrib/xml2. contrib/xml2 relies on this library to perform XSL transformations of XML.

--enable-integer-datetimes
Use 64-bit integer storage for datetimes and intervals, rather than the default floating-point storage. This reduces the range of representable values but guarantees microsecond precision across the full range (see Section 8.5 for more information).

--disable-spinlocks
Allow the build to succeed even if PostgreSQL has no CPU spinlock support for the platform. The lack of spinlock support will result in poor performance; therefore, this option should only be used if the build aborts and informs you that the platform lacks spinlock support. If this option is required to build PostgreSQL on your platform, please report the problem to the PostgreSQL developers.

--enable-thread-safety
Make the client libraries thread-safe. This allows concurrent threads in libpq and ECPG programs to safely control their private connection handles. This option requires adequate threading support in your operating system.

--with-system-tzdata=DIRECTORY
PostgreSQL includes its own time zone database, which it requires for date and time operations. This time zone database is in fact compatible with the \textit{zic} time zone database provided by many operating systems such as FreeBSD, Linux, and Solaris, so it would be redundant to install it again. When this option is used, the system-supplied time zone database in \textit{DIRECTORY} is used instead of the one included in the PostgreSQL source distribution. \textit{DIRECTORY} must be specified as an absolute path. /usr/share/zoneinfo is a likely directory on some operating systems. Note that the installation routine will not detect

\(^1\) http://www.ossp.org/pkg/lib/uuid/
Chapter 15. Installation Instructions

mismatching or erroneous time zone data. If you use this option, you are advised to run the regression tests to verify that the time zone data you have pointed to works correctly with PostgreSQL.

This option is mainly aimed at binary package distributors who know their target operating system well. The main advantage of using this option is that the PostgreSQL package won’t need to be upgraded whenever any of the many local daylight-saving time rules change. Another advantage is that PostgreSQL can be cross-compiled more straightforwardly if the time zone database files do not need to be built during the installation.

--without-zlib
Prevents use of the Zlib library. This disables support for compressed archives in pg_dump and pg_restore. This option is only intended for those rare systems where this library is not available.

--enable-debug
Compiles all programs and libraries with debugging symbols. This means that you can run the programs through a debugger to analyze problems. This enlarges the size of the installed executables considerably, and on non-GCC compilers it usually also disables compiler optimization, causing slowdowns. However, having the symbols available is extremely helpful for dealing with any problems that might arise. Currently, this option is recommended for production installations only if you use GCC. But you should always have it on if you are doing development work or running a beta version.

--enable-profiling
If using GCC, all programs and libraries are compiled so they can be profiled. On backend exit, a subdirectory will be created that contains the gmon.out file for use in profiling. This option is for use only with GCC and when doing development work.

--enable-cassert
 Enables assertion checks in the server, which test for many “cannot happen” conditions. This is invaluable for code development purposes, but the tests can slow down the server significantly. Also, having the tests turned on won’t necessarily enhance the stability of your server! The assertion checks are not categorized for severity, and so what might be a relatively harmless bug will still lead to server restarts if it triggers an assertion failure. This option is not recommended for production use, but you should have it on for development work or when running a beta version.

--enable-depend
Enables automatic dependency tracking. With this option, the makefiles are set up so that all affected object files will be rebuilt when any header file is changed. This is useful if you are doing development work, but is just wasted overhead if you intend only to compile once and install. At present, this option will work only if you use GCC.

--enable-dtrace
Compiles with support for the dynamic tracing tool DTrace. Operating system support for DTrace is currently only available in Solaris.

To point to the dtrace program, the environment variable DTRACE can be set. This will often be necessary because dtrace is typically installed under /usr/sbin, which might not be in the path. Additional command-line options for the dtrace program can be specified in the environment variable DTRACEFLAGS.

To include DTrace support in a 64-bit binary, specify DTRACEFLAGS="-64" to configure. For example, using the GCC compiler:
Chapter 15. Installation Instructions

./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='--64' ...

Using Sun’s compiler:

./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace DTRACEFLAGS='--64' ...

If you prefer a C compiler different from the one configure picks, you can set the environment variable CC to the program of your choice. By default, configure will pick gcc if available, else the platform’s default (usually cc). Similarly, you can override the default compiler flags if needed with the CFLAGS variable.

You can specify environment variables on the configure command line, for example:

./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'

Here is a list of the significant variables that can be set in this manner:

CC
   C compiler

CFLAGS
   options to pass to the C compiler

CPP
   C preprocessor

CPPFLAGS
   options to pass to the C preprocessor

DTRACE
   location of the dtrace program

DTRACEFLAGS
   options to pass to the dtrace program

LDFLAGS
   options to pass to the link editor

LDFLAGS_SL
   linker options for shared library linking

MSGFMT
   msgfmt program for native language support

PERL
   Full path to the Perl interpreter. This will be used to determine the dependencies for building PL/Perl.

PYTHON
   Full path to the Python interpreter. This will be used to determine the dependencies for building PL/Python.

TCLSH
   Full path to the Tcl interpreter. This will be used to determine the dependencies for building PL/Tcl.
Chapter 15. Installation Instructions

XML2_CONFIG

xml2-config program used to locate the libxml installation.

YACC

Yacc program (bison -y if using Bison)

2. Build

To start the build, type

```
gmake
```

(Remember to use GNU make.) The build will take a few minutes depending on your hardware. The last line displayed should be

```
All of PostgreSQL is successfully made. Ready to install.
```

3. Regression Tests

If you want to test the newly built server before you install it, you can run the regression tests at this point. The regression tests are a test suite to verify that PostgreSQL runs on your machine in the way the developers expected it to. Type

```
gmake check
```

(This won’t work as root; do it as an unprivileged user.) Chapter 29 contains detailed information about interpreting the test results. You can repeat this test at any later time by issuing the same command.

4. Installing The Files

**Note:** If you are upgrading an existing system and are going to install the new files over the old ones, be sure to back up your data and shut down the old server before proceeding, as explained in Section 15.4 above.

To install PostgreSQL enter

```
gmake install
```

This will install files into the directories that were specified in step 1. Make sure that you have appropriate permissions to write into that area. Normally you need to do this step as root. Alternatively, you could create the target directories in advance and arrange for appropriate permissions to be granted.

You can use `gmake install-strip` instead of `gmake install` to strip the executable files and libraries as they are installed. This will save some space. If you built with debugging support, stripping will effectively remove the debugging support, so it should only be done if debugging is no longer needed. `install-strip` tries to do a reasonable job saving space, but it does not have perfect knowledge of how to strip every unneeded byte from an executable file, so if you want to save all the disk space you possibly can, you will have to do manual work.

The standard installation provides all the header files needed for client application development as well as for server-side program development, such as custom functions or data types written in C. (Prior to PostgreSQL 8.0, a separate `gmake install-all-headers` command was needed for the latter, but this step has been folded into the standard install.)

**Client-only installation:** If you want to install only the client applications and interface libraries, then you can use these commands:

```
gmake -C src/bin install
```

322
Chapter 15. Installation Instructions

```
gmake -C src/include install

gmake -C src/interfaces install

gmake -C doc install
```

src/bin has a few binaries for server-only use, but they are small.

Registering eventlog on Windows: To register a Windows eventlog library with the operating system, issue this command after installation:

```
regsvr32 pgsql_library_directory/pgevent.dll
```

This creates registry entries used by the event viewer.

Uninstallation: To undo the installation use the command `gmake uninstall`. However, this will not remove any created directories.

Cleaning: After the installation you can make room by removing the built files from the source tree with the command `gmake clean`. This will preserve the files made by the `configure` program, so that you can rebuild everything with `gmake` later on. To reset the source tree to the state in which it was distributed, use `gmake distclean`. If you are going to build for several platforms within the same source tree you must do this and re-configure for each build. (Alternatively, use a separate build tree for each platform, so that the source tree remains unmodified.)

If you perform a build and then discover that your `configure` options were wrong, or if you change anything that `configure` investigates (for example, software upgrades), then it’s a good idea to do `gmake distclean` before reconfiguring and rebuilding. Without this, your changes in configuration choices might not propagate everywhere they need to.

### 15.6. Post-Installation Setup

15.6.1. Shared Libraries

On some systems that have shared libraries (which most systems do) you need to tell your system how to find the newly installed shared libraries. The systems on which this is not necessary include BSD/OS, FreeBSD, HP-UX, IRIX, Linux, NetBSD, OpenBSD, Tru64 UNIX (formerly Digital UNIX), and Solaris.

The method to set the shared library search path varies between platforms, but the most widely usable method is to set the environment variable `LD_LIBRARY_PATH` like so: In Bourne shells (`sh`, `ksh`, `bash`, `zsh`):

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

or in `csh` or `tcsh`:

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Replace `/usr/local/pgsql/lib` with whatever you set `--libdir` to in step 1. You should put these commands into a shell start-up file such as `/etc/profile` or `~/.bash_profile`. Some good information about the caveats associated with this method can be found at http://www.visi.com/~barr/ldpath.html.

On some systems it might be preferable to set the environment variable `LD_RUN_PATH` before building.
Chapter 15. Installation Instructions

On Cygwin, put the library directory in the PATH or move the .dll files into the bin directory.

If in doubt, refer to the manual pages of your system (perhaps ld.so or rld). If you later on get a message like

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

then this step was necessary. Simply take care of it then.

If you are on BSD/OS, Linux, or SunOS 4 and you have root access you can run:

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(or equivalent directory) after installation to enable the run-time linker to find the shared libraries faster. Refer to the manual page of ldconfig for more information. On FreeBSD, NetBSD, and OpenBSD the command is:

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

instead. Other systems are not known to have an equivalent command.

15.6.2. Environment Variables

If you installed into /usr/local/pgsql or some other location that is not searched for programs by default, you should add /usr/local/pgsql/bin (or whatever you set --bindir to in step 1) into your PATH. Strictly speaking, this is not necessary, but it will make the use of PostgreSQL much more convenient.

To do this, add the following to your shell start-up file, such as ~/.bash_profile (or /etc/profile, if you want it to affect every user):

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

If you are using csh or tcsh, then use this command:

```
set path = ( /usr/local/pgsql/bin $path )
```

To enable your system to find the man documentation, you need to add lines like the following to a shell start-up file unless you installed into a location that is searched by default:

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

The environment variables PGHOST and PGPORT specify to client applications the host and port of the database server, overriding the compiled-in defaults. If you are going to run client applications remotely then it is convenient if every user that plans to use the database sets PGHOST. This is not required, however: the settings can be communicated via command line options to most client programs.
15.7. Supported Platforms

A platform (that is, a CPU architecture and operating system combination) is considered supported by the PostgreSQL development community if the code contains provisions to work on that platform and it has recently been verified to build and pass its regression tests on that platform. Currently, most testing of platform compatibility is done automatically by test machines in the PostgreSQL Build Farm\(^2\). If you are interested in using PostgreSQL on a platform that is not represented in the build farm, but on which the code works or can be made to work, you are strongly encouraged to set up a build farm member machine so that continued compatibility can be assured.

In general, PostgreSQL can be expected to work on these CPU architectures: x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, Alpha, ARM, MIPS, MIPSEL, M68K, and PA-RISC. Code support exists for M32R, NS32K, and VAX, but these architectures are not known to have been tested recently. It is often possible to build on an unsupported CPU type by configuring with `--disable-spinlocks`, but performance will be poor.

PostgreSQL can be expected to work on these operating systems: Linux (all recent distributions), Windows (Win2000 SP4 and later), FreeBSD, OpenBSD, NetBSD, Mac OS X, AIX, HP/UX, IRIX, Solaris, Tru64 Unix, and UnixWare. Other Unix-like systems may also work but are not currently being tested. In most cases, all CPU architectures supported by a given operating system will work. Look in the `doc/` directory of the source distribution to see if there is a FAQ document specific to your operating system, particularly if using an older system.

If you have installation problems on a platform that is known to be supported according to recent build farm results, please report it to `<pgsql-bugs@postgresql.org>`. If you are interested in porting PostgreSQL to a new platform, `<pgsql-hackers@postgresql.org>` is the appropriate place to discuss that.

\(^2\) [http://buildfarm.postgresql.org/](http://buildfarm.postgresql.org/)
Chapter 16. Installation on Windows

It is recommended that most users download the binary distribution for Windows, available as a Windows Installer package from the PostgreSQL website. Building from source is only intended for people developing PostgreSQL or extensions.

There are several different ways of building PostgreSQL on Windows. The complete system can be built using MinGW or Visual C++ 2005. It can also be built for older versions of Windows using Cygwin. Finally, the client access library (libpq) can be built using Visual C++ 7.1 or Borland C++ for compatibility with statically linked applications built using these tools.

Building using MinGW or Cygwin uses the normal build system, see Chapter 15 and the FAQs in doc/FAQ_MINGW and do/FAQ_CYGWIN. Note that Cygwin is not recommended, and should only be used for older versions of Windows where the native build does not work, such as Windows 98.


The tools for building using Visual C++ 2005, are in the src/tools/msvc directory. When building, make sure there are no tools from MinGW or Cygwin present in your system PATH. Also, make sure you have all the required Visual C++ tools available in the PATH, usually by starting a Visual Studio Command Prompt and running the commands from there. All commands should be run from the src\tools\msvc directory.

Before you build, edit the file config.pl to reflect the configuration options you want set, including the paths to libraries used. If you need to set any other environment variables, create a file called buildenv.pl and put the required commands there. For example, to add the path for bison when it’s not in the PATH, create a file containing:

```
$ENV{PATH}=$ENV{PATH} . ';c:\some\where\bison\bin';
```

16.1.1. Requirements

PostgreSQL will build using either the professional versions (any edition) or the free Express edition of Visual Studio 2005. The following additional products are required to build the complete package. Use the config.pl file to specify which directories the libraries are available in.

ActiveState Perl

ActiveState Perl is required to run the build generation scripts. MinGW or Cygwin Perl will not work. It must also be present in the PATH. Binaries can be downloaded from http://www.activestate.com (Note: version 5.8 or later is required, the free Standard Distribution is sufficient).

ActiveState TCL

Required for building PL/TCL (Note: version 8.4 is required, the free Standard Distribution is sufficient).
Chapter 16. Installation on Windows

Bison and Flex

Bison and Flex are required to build from Git, but not required when building from a release file. Note that only Bison 1.875 or versions 2.2 and later will work. Bison and Flex can be downloaded from http://gnuwin32.sourceforge.net.

Diff

Diff is required to run the regression tests, and can be downloaded from http://gnuwin32.sourceforge.net.

Gettext

Gettext is required to build with NLS support, and can be downloaded from http://gnuwin32.sourceforge.net. Note that binaries, dependencies and developer files are all needed.

Microsoft Platform SDK

It is recommended that you upgrade to the latest available version of the Microsoft Platform SDK, available for download from http://www.microsoft.com/downloads/.

MIT Kerberos


libxml2 and libxslt

Required for XML support. Binaries can be downloaded from http://zlatkovic.com/pub/libxml or source from http://xmlsoft.org. Note that libxml2 requires iconv, which is available from the same download location.

openssl


ossp-uuid

Required for UUID-OSSP support (contrib only). Source can be downloaded from http://www.ossp.org/pkg/lib/uuid/.

Python


zlib

Required for compression support in pg_dump and pg_restore. Binaries can be downloaded from http://www.zlib.net.

16.1.2. Building

To build all of PostgreSQL in release configuration (the default), run the command:

\texttt{build}

To build all of PostgreSQL in debug configuration, run the command:
Chapter 16. Installation on Windows

To build just a single project, for example psql, run the commands:

```bash
build psql
build DEBUG psql
```

To change the default build configuration to debug, put the following in the `buildenv.pl` file:

```perl
$ENV{CONFIG}="Debug";
```

It is also possible to build from inside the Visual Studio GUI. In this case, you need to run:

```bash
perl mkvcbuild.pl
```

from the command prompt, and then open the generated `pgsql.sln` (in the root directory of the source tree) in Visual Studio.

### 16.1.3. Cleaning and installing

Most of the time, the automatic dependency tracking in Visual Studio will handle changed files. But if there have been large changes, you may need to clean the installation. To do this, simply run the `clean.bat` command, which will automatically clean out all generated files. You can also run it with the `dist` parameter, in which case it will behave like `make distclean` and remove the flex/bison output files as well.

By default, all files are written into a subdirectory of the `debug` or `release` directories. To install these files using the standard layout, and also generate the files required to initialize and use the database, run the command:

```bash
perl install.pl c:\destination\directory
```

### 16.1.4. Running the regression tests

To run the regression tests, make sure you have completed the build of all required parts first. Also, make sure that the DLLs required to load all parts of the system (such as the Perl and Python DLLs for the procedural languages) are present in the system path. If they are not, set it through the
Chapter 16. Installation on Windows

buildenv.pl file. To run the tests, run one of the following commands from the src\tools\msvc directory:

- vcregress check
- vcregress installcheck
- vcregress plcheck
- vcregress contribcheck

To change the schedule used (default is the parallel), append it to the command line like:

- vcregress check serial

For more information about the regression tests, see Chapter 29.

16.1.5. Building the documentation

Building the PostgreSQL documentation in HTML format requires several tools and files. Create a root directory for all these files, and store them in the subdirectories in the list below.

OpenJade 1.3.1-2

Download from http://sourceforge.net/project/downloading.php?groupname=openjade&filename=openjade-1_3_1-2-bin.zip and uncompress in the subdirectory openjade-1.3.1.

DocBook DTD 4.2


DocBook DSSSL 1.79


ISO character entities


Edit the buildenv.pl file, and add a variable for the location of the root directory, for example:

$ENV{DOCROOT}='c:\docbook';

To build the documentation, run the command builddoc.bat. Note that this will actually run the build twice, in order to generate the indexes. The generated HTML files will be in doc\src\sgml.
16.2. Building libpq with Visual C++ or Borland C++

Using Visual C++ 7.1-8.0 or Borland C++ to build libpq is only recommended if you need a version with different debug/release flags, or if you need a static library to link into an application. For normal use the MinGW or Visual Studio 2005 version is recommended.

To build the libpq client library using Visual Studio 7.1 or later, change into the src directory and type the command

```
nmake /f win32.mak
```

To build a 64-bit version of the libpq client library using Visual Studio 8.0 or later, change into the src directory and type in the command

```
nmake /f win32.mak CPU=AMD64
```

See the win32.mak file for further details about supported variables.

To build the libpq client library using Borland C++, change into the src directory and type the command

```
make -N -DCFG=Release /f bcc32.mak
```

16.2.1. Generated files

The following files will be built:

```
interfaces\libpq\Release\libpq.dll
```

The dynamically linkable frontend library

```
interfaces\libpq\Release\libpqdll.lib
```

Import library to link your programs to libpq.dll

```
interfaces\libpq\Release\libpq.lib
```

Static version of the frontend library

Normally you do not need to install any of the client files. You should place the libpq.dll file in the same directory as your applications executable file. Do not install libpq.dll into your Windows, System or System32 directory unless absolutely necessary. If this file is installed using a setup program, it should be installed with version checking using the VERSIONINFO resource included in the file, to ensure that a newer version of the library is not overwritten.

If you are planning to do development using libpq on this machine, you will have to add the src\include and src\interfaces\libpq subdirectories of the source tree to the include path in your compiler’s settings.

To use the library, you must add the libpqdll.lib file to your project. (In Visual C++, just right-click on the project and choose to add it.)
Chapter 17. Operating System Environment

This chapter discusses how to set up and run the database server and its interactions with the operating system.

17.1. The PostgreSQL User Account

As with any other server daemon that is accessible to the outside world, it is advisable to run PostgreSQL under a separate user account. This user account should only own the data that is managed by the server, and should not be shared with other daemons. (For example, using the user nobody is a bad idea.) It is not advisable to install executables owned by this user because compromised systems could then modify their own binaries.

To add a Unix user account to your system, look for a command useradd or adduser. The user name postgres is often used, and is assumed throughout this book, but you can use another name if you like.

17.2. Creating a Database Cluster

Before you can do anything, you must initialize a database storage area on disk. We call this a database cluster. (SQL uses the term catalog cluster.) A database cluster is a collection of databases that is managed by a single instance of a running database server. After initialization, a database cluster will contain a database named postgres, which is meant as a default database for use by utilities, users and third party applications. The database server itself does not require the postgres database to exist, but many external utility programs assume it exists. Another database created within each cluster during initialization is called template1. As the name suggests, this will be used as a template for subsequently created databases; it should not be used for actual work. (See Chapter 20 for information about creating new databases within a cluster.)

In file system terms, a database cluster will be a single directory under which all data will be stored. We call this the data directory or data area. It is completely up to you where you choose to store your data. There is no default, although locations such as /usr/local/pgsql/data or /var/lib/pgsql/data are popular. To initialize a database cluster, use the command initdb, which is installed with PostgreSQL. The desired file system location of your database cluster is indicated by the -D option, for example

$ initdb -D /usr/local/pgsql/data

Note that you must execute this command while logged into the PostgreSQL user account, which is described in the previous section.

Tip: As an alternative to the -D option, you can set the environment variable PGDATA.

initdb will attempt to create the directory you specify if it does not already exist. It is likely that it will not have the permission to do so (if you followed our advice and created an unprivileged account). In that case you should create the directory yourself (as root) and change the owner to be the PostgreSQL user. Here is how this might be done:

root# mkdir /usr/local/pgsql/data
Chapter 17. Operating System Environment

root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data

initdb will refuse to run if the data directory looks like it has already been initialized.

Because the data directory contains all the data stored in the database, it is essential that it be secured from unauthorized access. initdb therefore revokes access permissions from everyone but the PostgreSQL user.

However, while the directory contents are secure, the default client authentication setup allows any local user to connect to the database and even become the database superuser. If you do not trust other local users, we recommend you use one of initdb’s -W, --pwprompt or --pwfile options to assign a password to the database superuser. Also, specify -A md5 or -A password so that the default trust authentication mode is not used; or modify the generated pg_hba.conf file after running initdb, before you start the server for the first time. (Other reasonable approaches include using ident authentication or file system permissions to restrict connections. See Chapter 21 for more information.)

initdb also initializes the default locale for the database cluster. Normally, it will just take the locale settings in the environment and apply them to the initialized database. It is possible to specify a different locale for the database; more information about that can be found in Section 22.1. The sort order used within a particular database cluster is set by initdb and cannot be changed later, short of dumping all data, rerunning initdb, and reloading the data. There is also a performance impact for using locales other than C or POSIX. Therefore, it is important to make this choice correctly the first time.

initdb also sets the default character set encoding for the database cluster. Normally this should be chosen to match the locale setting. For details see Section 22.2.

17.2.1. Network File Systems

Many installations create database clusters on network file systems. Sometimes this is done directly via NFS, or by using a Network Attached Storage (NAS) device that uses NFS internally. PostgreSQL does nothing special for NFS file systems, meaning it assumes NFS behaves exactly like locally-connected drives (DAS, Direct Attached Storage). If client and server NFS implementations have non-standard semantics, this can cause reliability problems (see http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html). Specifically, delayed (asynchronous) writes to the NFS server can cause reliability problems; if possible, mount NFS file systems synchronously (without caching) to avoid this. (Storage Area Networks (SAN) use a low-level communication protocol rather than NFS.)

17.3. Starting the Database Server

Before anyone can access the database, you must start the database server. The database server program is called postgres. The postgres program must know where to find the data it is supposed to use. This is done with the -D option. Thus, the simplest way to start the server is:

$ postgres -D /usr/local/pgsql/data
which will leave the server running in the foreground. This must be done while logged into the Post- 
greSQL user account. Without -D, the server will try to use the data directory named by the environ-
ment variable PGDATA. If that variable is not provided either, it will fail.

Normally it is better to start `postgres` in the background. For this, use the usual shell syntax:

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

It is important to store the server’s stdout and stderr output somewhere, as shown above. It will help 
for auditing purposes and to diagnose problems. (See Section 23.3 for a more thorough discussion of 
log file handling.)

The `postgres` program also takes a number of other command-line options. For more information, 
see the `postgres` reference page and Chapter 18 below.

This shell syntax can get tedious quickly. Therefore the wrapper program `pg_ctl` is provided to sim-
plify some tasks. For example:

```
pg_ctl start -l logfile
```

will start the server in the background and put the output into the named log file. The -D option has 
the same meaning here as for `postgres`. `pg_ctl` is also capable of stopping the server.

Normally, you will want to start the database server when the computer boots. Autostart 
scripts are operating-system-specific. There are a few distributed with PostgreSQL in the 
`contrib/start-scripts` directory. Installing one will require root privileges.

Different systems have different conventions for starting up daemons at boot time. Many systems have 
a file `/etc/rc.local` or `/etc/rc.d/rc.local`. Others use `rc.d` directories. Whatever you do, 
the server must be run by the PostgreSQL user account and not by root or any other user. Therefore 
you probably should form your commands using `su postgres -c '...'. For example:

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

Here are a few more operating-system-specific suggestions. (In each case be sure to use the proper 
installation directory and user name where we show generic values.)

- For FreeBSD, look at the file `contrib/start-scripts/freebsd` in the PostgreSQL source 
distribution.
- On OpenBSD, add the following lines to the file `/etc/rc.local`:

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
  su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -D /usr/local/pgsql/data'
  echo -n ' postgresql'
  fi
  ```
- On Linux systems either add

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data 
  ```
  
  to `/etc/rc.d/rc.local` or look at the file `contrib/start-scripts/linux` in the Post-
  greSQL source distribution.
- On NetBSD, use either the FreeBSD or Linux start scripts, depending on preference.
- On Solaris, create a file called `/etc/init.d/postgresql` that contains the following line:

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data"
  ```

  Then, create a symbolic link to it in `/etc/rc3.d/S99postgresql`. 


While the server is running, its PID is stored in the file `postmaster.pid` in the data directory. This is used to prevent multiple server instances from running in the same data directory and can also be used for shutting down the server.

### 17.3.1. Server Start-up Failures

There are several common reasons the server might fail to start. Check the server’s log file, or start it by hand (without redirecting standard output or standard error) and see what error messages appear. Below we explain some of the most common error messages in more detail.

**LOG:** could not bind IPv4 socket: Address already in use  
**HINT:** Is another postmaster already running on port 5432? If not, wait a few seconds and retry.  
**FATAL:** could not create TCP/IP listen socket

This usually means just what it suggests: you tried to start another server on the same port where one is already running. However, if the kernel error message is not `Address already in use` or some variant of that, there might be a different problem. For example, you might try to start a server on a reserved port number:

```
$ postgres -p 666  
```

**LOG:** could not bind IPv4 socket: Permission denied  
**HINT:** Is another postmaster already running on port 666? If not, wait a few seconds and retry.  
**FATAL:** could not create TCP/IP listen socket

A message like

**FATAL:** could not create shared memory segment: Invalid argument  
**DETAIL:** Failed system call was `shmget(key=5440001, size=4011376640, 03600)`.

probably means your kernel’s limit on the size of shared memory is smaller than the work area PostgreSQL is trying to create (4011376640 bytes in this example). Or it could mean that you do not have System-V-style shared memory support configured into your kernel at all. As a temporary workaround, you can try starting the server with a smaller-than-normal number of buffers (shared_buffers). You will eventually want to reconfigure your kernel to increase the allowed shared memory size. You might also see this message when trying to start multiple servers on the same machine, if their total space requested exceeds the kernel limit.

An error like

**FATAL:** could not create semaphores: No space left on device  
**DETAIL:** Failed system call was `semget(5440126, 17, 03600)`.

does not mean you’ve run out of disk space. It means your kernel’s limit on the number of System V semaphores is smaller than the number PostgreSQL wants to create. As above, you might be able to work around the problem by starting the server with a reduced number of allowed connections (max_connections), but you’ll eventually want to increase the kernel limit.

If you get an “illegal system call” error, it is likely that shared memory or semaphores are not supported in your kernel at all. In that case your only option is to reconfigure the kernel to enable these features.

Details about configuring System V IPC facilities are given in Section 17.4.1.
17.3.2. Client Connection Problems

Although the error conditions possible on the client side are quite varied and application-dependent, a few of them might be directly related to how the server was started up. Conditions other than those shown below should be documented with the respective client application.

```
psql: could not connect to server: Connection refused
    Is the server running on host "server.joe.com" and accepting
    TCP/IP connections on port 5432?
```

This is the generic “I couldn’t find a server to talk to” failure. It looks like the above when TCP/IP communication is attempted. A common mistake is to forget to configure the server to allow TCP/IP connections.

Alternatively, you’ll get this when attempting Unix-domain socket communication to a local server:

```
psql: could not connect to server: No such file or directory
    Is the server running locally and accepting
    connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

The last line is useful in verifying that the client is trying to connect to the right place. If there is in fact no server running there, the kernel error message will typically be either Connection refused or No such file or directory, as illustrated. (It is important to realize that Connection refused in this context does not mean that the server got your connection request and rejected it. That case will produce a different message, as shown in Section 21.3.) Other error messages such as Connection timed out might indicate more fundamental problems, like lack of network connectivity.

17.4. Managing Kernel Resources

A large PostgreSQL installation can quickly exhaust various operating system resource limits. (On some systems, the factory defaults are so low that you don’t even need a really “large” installation.) If you have encountered this kind of problem, keep reading.

17.4.1. Shared Memory and Semaphores

Shared memory and semaphores are collectively referred to as “System V IPC” (together with message queues, which are not relevant for PostgreSQL). Almost all modern operating systems provide these features, but not all of them have them turned on or sufficiently sized by default, especially systems with BSD heritage. (On Windows, PostgreSQL provides its own replacement implementation of these facilities, and so most of this section can be disregarded.)

The complete lack of these facilities is usually manifested by an Illegal system call error upon server start. In that case there’s nothing left to do but to reconfigure your kernel. PostgreSQL won’t work without them.

When PostgreSQL exceeds one of the various hard IPC limits, the server will refuse to start and should leave an instructive error message describing the problem encountered and what to do about it. (See also Section 17.3.1.) The relevant kernel parameters are named consistently across different systems; Table 17-1 gives an overview. The methods to set them, however, vary. Suggestions for some
platforms are given below. Be warned that it is often necessary to reboot your machine, and possibly even recompile the kernel, to change these settings.

Table 17-1. System V IPC parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Reasonable values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMMAX</td>
<td>Maximum size of shared memory segment (bytes)</td>
<td>at least several megabytes (see text)</td>
</tr>
<tr>
<td>SHMMIN</td>
<td>Minimum size of shared memory segment (bytes)</td>
<td>1</td>
</tr>
<tr>
<td>SHMALL</td>
<td>Total amount of shared memory available (bytes or pages)</td>
<td>if bytes, same as SHMMAX; if pages, ceil(SHMMAX/PAGE_SIZE)</td>
</tr>
<tr>
<td>SHMSEG</td>
<td>Maximum number of shared memory segments per process</td>
<td>only 1 segment is needed, but the default is much higher</td>
</tr>
<tr>
<td>SHMMNI</td>
<td>Maximum number of shared memory segments system-wide</td>
<td>like SHMSEG plus room for other applications</td>
</tr>
<tr>
<td>SEMMNI</td>
<td>Maximum number of semaphore identifiers (i.e., sets)</td>
<td>at least ceil((max_connections + autovacuum_max_workers) / 16)</td>
</tr>
<tr>
<td>SEMMNS</td>
<td>Maximum number of semaphores system-wide</td>
<td>ceil((max_connections + autovacuum_max_workers) / 16) * 17 plus room for other applications</td>
</tr>
<tr>
<td>SEMMSL</td>
<td>Maximum number of semaphores per set</td>
<td>at least 17</td>
</tr>
<tr>
<td>SEMMAP</td>
<td>Number of entries in semaphore map</td>
<td>see text</td>
</tr>
<tr>
<td>SEMVMX</td>
<td>Maximum value of semaphore</td>
<td>at least 1000 (The default is often 32767, don’t change unless forced to)</td>
</tr>
</tbody>
</table>

The most important shared memory parameter is SHMMAX, the maximum size, in bytes, of a shared memory segment. If you get an error message from `shmget` like Invalid argument, it is likely that this limit has been exceeded. The size of the required shared memory segment varies depending on several PostgreSQL configuration parameters, as shown in Table 17-2. (Any error message you might get will include the exact size of the failed allocation request.) You can, as a temporary solution, lower some of those settings to avoid the failure. While it is possible to get PostgreSQL to run with SHMMAX as small as 2 MB, you need considerably more for acceptable performance. Desirable settings are in the tens to hundreds of megabytes.

Some systems also have a limit on the total amount of shared memory in the system (SHMALL). Make sure this is large enough for PostgreSQL plus any other applications that are using shared memory segments. (Caution: SHMALL is measured in pages rather than bytes on many systems.)

Less likely to cause problems is the minimum size for shared memory segments (SHMMIN), which should be at most approximately 500 kB for PostgreSQL (it is usually just 1). The maximum number of segments system-wide (SHMMNI) or per-process (SHMSEG) are unlikely to cause a problem unless your system has them set to zero.

PostgreSQL uses one semaphore per allowed connection (max_connections) and allowed autovac-
uum worker process (autovacuum_max_workers), in sets of 16. Each such set will also contain a 17th semaphore which contains a “magic number”, to detect collision with semaphore sets used by other applications. The maximum number of semaphores in the system is set by SEMMNS, which consequently must be at least as high as max_connections plus autovacuum_max_workers, plus one extra for each 16 allowed connections plus workers (see the formula in Table 17-1). The parameter SEMMNI determines the limit on the number of semaphore sets that can exist on the system at one time. Hence this parameter must be at least \(\lceil(\text{max_connections} + \text{autovacuum_max_workers}) / 16\rceil\). Lowering the number of allowed connections is a temporary workaround for failures, which are usually confusingly worded No space left on device, from the function semget.

In some cases it might also be necessary to increase SEMMAP to be at least on the order of SEMMNS. This parameter defines the size of the semaphore resource map, in which each contiguous block of available semaphores needs an entry. When a semaphore set is freed it is either added to an existing entry that is adjacent to the freed block or it is registered under a new map entry. If the map is full, the freed semaphores get lost (until reboot). Fragmentation of the semaphore space could over time lead to fewer available semaphores than there should be.

The SEMMSL parameter, which determines how many semaphores can be in a set, must be at least 17 for PostgreSQL.

Various other settings related to “semaphore undo”, such as SEMMNU and SEMUME, are not of concern for PostgreSQL.

AIX

At least as of version 5.1, it should not be necessary to do any special configuration for such parameters as SHMMAX, as it appears this is configured to allow all memory to be used as shared memory. That is the sort of configuration commonly used for other databases such as DB/2.

It might, however, be necessary to modify the global ulimit information in /etc/security/limits, as the default hard limits for file sizes (fsize) and numbers of files (nofiles) might be too low.

BSD/OS

Shared Memory. By default, only 4 MB of shared memory is supported. Keep in mind that shared memory is not pageable; it is locked in RAM. To increase the amount of shared memory supported by your system, add something like the following to your kernel configuration file:

```bash
options "SHMALL=8192"
options "SHMMAX=\(\text{SHMALL} \times \text{PAGE_SIZE}\)"
```

SHMALL is measured in 4 kB pages, so a value of 1024 represents 4 MB of shared memory. Therefore the above increases the maximum shared memory area to 32 MB. For those running 4.3 or later, you will probably also need to increase KERNEL_VIRTUAL_MB above the default 248. Once all changes have been made, recompile the kernel, and reboot.

For those running 4.0 and earlier releases, use bpatch to find the sysptsize value in the current kernel. This is computed dynamically at boot time.

```bash
$ bpatch -r sysptsize
0x9 = 9
```

Next, add SYSPTSIZE as a hard-coded value in the kernel configuration file. Increase the value you found using bpatch. Add 1 for every additional 4 MB of shared memory you desire.

```bash
options "SYSPTSIZE=16"
sysptsize cannot be changed by sysctl.
```

Semaphores. You will probably want to increase the number of semaphores as well; the default system total of 60 will only allow about 50 PostgreSQL connections. Set the values you want in your kernel configuration file, e.g.:
options "SEMMNI=40"
options "SEMMNS=240"

FreeBSD

The default settings are only suitable for small installations (for example, default SHMMAX is 32 MB). Changes can be made via the sysctl or loader interfaces. The following parameters can be set using sysctl:

$$
\text{sysctl -w kern.ipc.shmall=32768} \\
\text{sysctl -w kern.ipc.shmmax=134217728} \\
\text{sysctl -w kern.ipc.semmap=256}
$$

To have these settings persist over reboots, modify /etc/sysctl.conf.

The remaining semaphore settings are read-only as far as sysctl is concerned, but can be changed before boot using the loader prompt:

(loader) \text{set kern.ipc.semmni=256}
(loader) \text{set kern.ipc.semmns=512}
(loader) \text{set kern.ipc.semmnu=256}

Similarly these can be saved between reboots in /boot/loader.conf.

You might also want to configure your kernel to lock shared memory into RAM and prevent it from being paged out to swap. This can be accomplished using the sysctl setting kern.ipc.shm_use_phys.

If running in FreeBSD jails by enabling sysctl’s security.jail.sysvipc_allowed, postmasters running in different jails should be run by different operating system users. This improves security because it prevents non-root users from interfering with shared memory or semaphores in a different jail, and it allows the PostgreSQL IPC cleanup code to function properly. (In FreeBSD 6.0 and later the IPC cleanup code doesn’t properly detect processes in other jails, preventing the running of postmasters on the same port in different jails.)

FreeBSD versions before 4.0 work like OpenBSD (see below).

NetBSD

In NetBSD 5.0 and later, IPC parameters can be adjusted using sysctl, for example:

$$
\text{sysctl -w kern.ipc.shmmax=16777216}
$$

To have these settings persist over reboots, modify /etc/sysctl.conf.

You might also want to configure your kernel to lock shared memory into RAM and prevent it from being paged out to swap. This can be accomplished using the sysctl setting kern.ipc.shm_use_phys.

NetBSD versions before 5.0 work like OpenBSD (see below), except that parameters should be set with the keyword options not option.

OpenBSD

The options SYSVSHM and SYSVSEM need to be enabled when the kernel is compiled. (They are by default.) The maximum size of shared memory is determined by the option SHMMAXPGS (in pages). The following shows an example of how to set the various parameters:

```
option SYSVSHM
option SHMMAXPGS=4096
option SHMSEG=256

option SYSVSEM
option SEMMNI=256
option SEMMNS=512
option SEMMNU=256
```
option SEMMAP=256

You might also want to configure your kernel to lock shared memory into RAM and prevent it from beingpaged out to swap. This can be accomplished using the sysctl setting kern.ipc.shm_use_phys.

HP-UX

The default settings tend to suffice for normal installations. On HP-UX 10, the factory default for SEMMNS is 128, which might be too low for larger database sites.

IPC parameters can be set in the System Administration Manager (SAM) under Kernel Configuration → Configurable Parameters. Hit Create A New Kernel when you’re done.

Linux

The default maximum segment size is 32 MB, which is only adequate for small PostgreSQL installations. However, the remaining defaults are quite generously sized, and usually do not require changes. The maximum shared memory segment size can be changed via the sysctl interface. For example, to allow 128 MB, and explicitly set the maximum total shared memory size to 2097152 pages (the default):

```
$ sysctl -w kernel.shmmax=134217728
$ sysctl -w kernel.shmall=2097152
```

In addition these settings can be saved between reboots in /etc/sysctl.conf.

Older distributions might not have the sysctl program, but equivalent changes can be made by manipulating the /proc file system:

```
$ echo 134217728 > /proc/sys/kernel/shmmax
$ echo 2097152 > /proc/sys/kernel/shmall
```

MacOS X

In OS X 10.2 and earlier, edit the file /System/Library/StartupItems/SystemTuning/SystemTuning and change the values in the following commands:

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

In OS X 10.3 and later, these commands have been moved to /etc/rc and must be edited there. Note that /etc/rc is usually overwritten by OS X updates (such as 10.3.6 to 10.3.7) so you should expect to have to redo your editing after each update.

In OS X 10.3.9 and later, instead of editing /etc/rc you can create a file named /etc/sysctl.conf, containing variable assignments such as:

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

This method is better than editing /etc/rc because your changes will be preserved across system updates. Note that all five shared-memory parameters must be set in /etc/sysctl.conf, else the values will be ignored.

Beware that recent releases of OS X ignore attempts to set SHMMAX to a value that isn’t an exact multiple of 4096.

SHMALL is measured in 4 kB pages on this platform.
In all OS X versions, you’ll need to reboot to make changes in the shared memory parameters take effect.

SCO OpenServer

In the default configuration, only 512 kB of shared memory per segment is allowed. To increase the setting, first change to the directory /etc/conf/cf.d. To display the current value of SHMMAX, run:

```
./configure -y SHMMAX
```

To set a new value for SHMMAX, run:

```
./configure SHMMAX=value
```

where value is the new value you want to use (in bytes). After setting SHMMAX, rebuild the kernel:

```
./link_unix
```

and reboot.

Solaris

At least in version 2.6, the default maximum size of a shared memory segment is too low for PostgreSQL. The relevant settings can be changed in /etc/system, for example:

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmin=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

You need to reboot for the changes to take effect.

See also http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html for information on shared memory under Solaris.

UnixWare

On UnixWare 7, the maximum size for shared memory segments is only 512 kB in the default configuration. To display the current value of SHMMAX, run:

```
/etc/conf/bin/idtune -g SHMMAX
```

which displays the current, default, minimum, and maximum values. To set a new value for SHMMAX, run:

```
/etc/conf/bin/idtune SHMMAX value
```

where value is the new value you want to use (in bytes). After setting SHMMAX, rebuild the kernel:

```
/etc/conf/bin/idbuild -B
```

and reboot.

Table 17-2. Configuration parameters affecting PostgreSQL’s shared memory usage

<table>
<thead>
<tr>
<th>Name</th>
<th>Approximate multiplier (bytes per increment) as of 8.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_connections</td>
<td>1800 + 270 * max_locks_per_transaction</td>
</tr>
<tr>
<td>autovacuum_max_workers</td>
<td>1800 + 270 * max_locks_per_transaction</td>
</tr>
<tr>
<td>max_prepared_transactions</td>
<td>770 + 270 * max_locks_per_transaction</td>
</tr>
<tr>
<td>Name</td>
<td>Approximate multiplier (bytes per increment) as of 8.3</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>shared_buffers</td>
<td>8400 (assuming 8 kB BLCKSZ)</td>
</tr>
<tr>
<td>wal_buffers</td>
<td>8200 (assuming 8 kB XLOG_BLCKSZ)</td>
</tr>
<tr>
<td>max_fsm_relations</td>
<td>70</td>
</tr>
<tr>
<td>max_fsm_pages</td>
<td>6</td>
</tr>
<tr>
<td>Fixed space requirements</td>
<td>770 kB</td>
</tr>
</tbody>
</table>

### 17.4.2. Resource Limits

Unix-like operating systems enforce various kinds of resource limits that might interfere with the operation of your PostgreSQL server. Of particular importance are limits on the number of processes per user, the number of open files per process, and the amount of memory available to each process. Each of these have a “hard” and a “soft” limit. The soft limit is what actually counts but it can be changed by the user up to the hard limit. The hard limit can only be changed by the root user. The system call `setrlimit` is responsible for setting these parameters. The shell’s built-in command `ulimit` (Bourne shells) or `limit` (csh) is used to control the resource limits from the command line. On BSD-derived systems the file `/etc/login.conf` controls the various resource limits set during login. See the operating system documentation for details. The relevant parameters are `maxproc`, `openfiles`, and `datasize`. For example:

```plaintext
default:\  ...
  :datasize-cur=256M:\
  :maxproc-cur=256:\
  :openfiles-cur=256:\
  ...  
```

(\-cur is the soft limit. Append \-max to set the hard limit.)

Kernels can also have system-wide limits on some resources.

- On Linux `/proc/sys/fs/file-max` determines the maximum number of open files that the kernel will support. It can be changed by writing a different number into the file or by adding an assignment in `/etc/sysctl.conf`. The maximum limit of files per process is fixed at the time the kernel is compiled; see `/usr/src/linux/Documentation/proc.txt` for more information.

The PostgreSQL server uses one process per connection so you should provide for at least as many processes as allowed connections, in addition to what you need for the rest of your system. This is usually not a problem but if you run several servers on one machine things might get tight.

The factory default limit on open files is often set to “socially friendly” values that allow many users to coexist on a machine without using an inappropriate fraction of the system resources. If you run many servers on a machine this is perhaps what you want, but on dedicated servers you might want to raise this limit.

On the other side of the coin, some systems allow individual processes to open large numbers of files; if more than a few processes do so then the system-wide limit can easily be exceeded. If you find this happening, and you do not want to alter the system-wide limit, you can set PostgreSQL’s `max_files_per_process` configuration parameter to limit the consumption of open files.
17.4.3. Linux Memory Overcommit

In Linux 2.4 and later, the default virtual memory behavior is not optimal for PostgreSQL. Because of the way that the kernel implements memory overcommit, the kernel might terminate the PostgreSQL server (the master server process) if the memory demands of another process cause the system to run out of virtual memory.

If this happens, you will see a kernel message that looks like this (consult your system documentation and configuration on where to look for such a message):

Out of Memory: Killed process 12345 (postgres).

This indicates that the postgres process has been terminated due to memory pressure. Although existing database connections will continue to function normally, no new connections will be accepted. To recover, PostgreSQL will need to be restarted.

One way to avoid this problem is to run PostgreSQL on a machine where you can be sure that other processes will not run the machine out of memory. If memory is tight, increasing the swap space of the operating system can help avoiding the problem, because the out-of-memory (OOM) killer is invoked whenever physical memory and swap space are exhausted.

On Linux 2.6 and later, an additional measure is to modify the kernel’s behavior so that it will not “overcommit” memory. Although this setting will not prevent the OOM killer from being invoked altogether, it will lower the chances significantly and will therefore lead to more robust system behavior. This is done by selecting strict overcommit mode via `sysctl`:

```bash
sysctl -w vm.overcommit_memory=2
```

or placing an equivalent entry in `/etc/sysctl.conf`. You might also wish to modify the related setting `vm.overcommit_ratio`. For details see the kernel documentation file `Documentation/vm/overcommit-accounting`.

Some vendors’ Linux 2.4 kernels are reported to have early versions of the 2.6 overcommit `sysctl` parameter. However, setting `vm.overcommit_memory` to 2 on a kernel that does not have the relevant code will make things worse not better. It is recommended that you inspect the actual kernel source code (see the function `vm_enough_memory` in the file `mm/mmap.c`) to verify what is supported in your copy before you try this in a 2.4 installation. The presence of the `overcommit-accounting` documentation file should not be taken as evidence that the feature is there. If in any doubt, consult a kernel expert or your kernel vendor.

17.5. Shutting Down the Server

There are several ways to shut down the database server. You control the type of shutdown by sending different signals to the master `postgres` process.

SIGTERM

After receiving SIGTERM, the server disallows new connections, but lets existing sessions end their work normally. It shuts down only after all of the sessions terminate normally. This is the Smart Shutdown.

1. http://lwn.net/Articles/104179/
Chapter 17. Operating System Environment

SIGINT

The server disallows new connections and sends all existing server processes SIGTERM, which will cause them to abort their current transactions and exit promptly. It then waits for the server processes to exit and finally shuts down. This is the Fast Shutdown.

SIGQUIT

This is the Immediate Shutdown, which will cause the master `postgres` process to send a SIGQUIT to all child processes and exit immediately, without properly shutting itself down. The child processes likewise exit immediately upon receiving SIGQUIT. This will lead to recovery (by replaying the WAL log) upon next start-up. This is recommended only in emergencies.

The `pg_ctl` program provides a convenient interface for sending these signals to shut down the server. Alternatively, you can send the signal directly using `kill` on non-Windows systems. The PID of the `postgres` process can be found using the `ps` program, or from the file `postmaster.pid` in the data directory. For example, to do a fast shutdown:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

**Important:** It is best not to use SIGKILL to shut down the server. Doing so will prevent the server from releasing shared memory and semaphores, which might then have to be done manually before a new server can be started. Furthermore, SIGKILL kills the `postgres` process without letting it relay the signal to its subprocesses, so it will be necessary to kill the individual subprocesses by hand as well.

17.6. Preventing Server Spoofing

While the server is running, it is not possible for a malicious user to interfere with client/server communications. However, when the server is down it is possible for a local user to spoof the normal server by starting their own server. The spoof server could read passwords and queries sent by clients, but could not return any data because the `PGDATA` directory would still be secure because of directory permissions. Spoofing is possible because any user can start a database server; a client cannot identify an invalid server unless it is specially configured.

The simplest way to prevent invalid servers for local connections is to use a Unix domain socket directory (`unix_socket_directory`) that has write permission only for a trusted local user. This prevents a malicious user from creating their own socket file in that directory. If you are concerned that some applications might still reference `/tmp` for the socket file and hence be vulnerable to spoofing, during operating system startup create symbolic link `/tmp/.s.PGSQL.5432` that points to the relocated socket file. You also might need to modify your `/tmp` cleanup script to preserve the symbolic link.

For TCP connections the server must accept only `hostssl` connections (Section 21.1) and have SSL `server.key` (key) and `server.crt` (certificate) files (Section 17.8). The TCP client must connect using `sslmode='require'` (Section 30.1) and have a `~/.postgresql/root.crt` SSL certificate (Section 30.16).
Chapter 17. Operating System Environment

17.7. Encryption Options

PostgreSQL offers encryption at several levels, and provides flexibility in protecting data from disclosure due to database server theft, unscrupulous administrators, and insecure networks. Encryption might also be required to secure sensitive data such as medical records or financial transactions.

Password Storage Encryption

By default, database user passwords are stored as MD5 hashes, so the administrator cannot determine the actual password assigned to the user. If MD5 encryption is used for client authentication, the unencrypted password is never even temporarily present on the server because the client MD5 encrypts it before being sent across the network.

Encryption For Specific Columns

The contrib function library pgcrypto allows certain fields to be stored encrypted. This is useful if only some of the data is sensitive. The client supplies the decryption key and the data is decrypted on the server and then sent to the client.

The decrypted data and the decryption key are present on the server for a brief time while it is being decrypted and communicated between the client and server. This presents a brief moment where the data and keys can be intercepted by someone with complete access to the database server, such as the system administrator.

Data Partition Encryption

On Linux, encryption can be layered on top of a file system mount using a “loopback device”. This allows an entire file system partition be encrypted on disk, and decrypted by the operating system. On FreeBSD, the equivalent facility is called GEOM Based Disk Encryption, or gbde.

This mechanism prevents unencrypted data from being read from the drives if the drives or the entire computer is stolen. This does not protect against attacks while the file system is mounted, because when mounted, the operating system provides an unencrypted view of the data. However, to mount the file system, you need some way for the encryption key to be passed to the operating system, and sometimes the key is stored somewhere on the host that mounts the disk.

Encrypting Passwords Across A Network

The MD5 authentication method double-encrypts the password on the client before sending it to the server. It first MD5 encrypts it based on the user name, and then encrypts it based on a random salt sent by the server when the database connection was made. It is this double-encrypted value that is sent over the network to the server. Double-encryption not only prevents the password from being discovered, it also prevents another connection from using the same encrypted password to connect to the database server at a later time.

Encrypting Data Across A Network

SSL connections encrypt all data sent across the network: the password, the queries, and the data returned. The pg_hba.conf file allows administrators to specify which hosts can use non-encrypted connections (host) and which require SSL-encrypted connections (hostssl). Also, clients can specify that they connect to servers only via SSL. Stunnel or SSH can also be used to encrypt transmissions.

SSL Host Authentication

It is possible for both the client and server to provide SSL keys or certificates to each other. It takes some extra configuration on each side, but this provides stronger verification of identity than the mere use of passwords. It prevents a computer from pretending to be the server just long enough to read the password send by the client. It also helps prevent “man in the middle” attacks...
where a computer between the client and server pretends to be the server and reads and passes all data between the client and server.

Client-Side Encryption

If the system administrator cannot be trusted, it is necessary for the client to encrypt the data; this way, unencrypted data never appears on the database server. Data is encrypted on the client before being sent to the server, and database results have to be decrypted on the client before being used.

**17.8. Secure TCP/IP Connections with SSL**

PostgreSQL has native support for using SSL connections to encrypt client/server communications for increased security. This requires that OpenSSL is installed on both client and server systems and that support in PostgreSQL is enabled at build time (see Chapter 15).

With SSL support compiled in, the PostgreSQL server can be started with SSL enabled by setting the parameter `ssl` to `on` in `postgresql.conf`. The server will listen for both standard and SSL connections on the same TCP port, and will negotiate with any connecting client on whether to use SSL. By default, this is at the client’s option; see Section 21.1 about how to set up the server to require use of SSL for some or all connections.

PostgreSQL reads the system-wide OpenSSL configuration file. By default, this file is named `openssl.cnf` and is located in the directory reported by `openssl version -d`. This default can be overridden by setting environment variable `OPENSSL_CONF` to the name of the desired configuration file.

OpenSSL supports a wide range of ciphers and authentication algorithms, of varying strength. While a list of ciphers can be specified in the OpenSSL configuration file, you can specify ciphers specifically for use by the database server by modifying `ssl_ciphers` in `postgresql.conf`.

**Note:** It is possible to have authentication without encryption overhead by using `NULL-SHA` or `NULL-MD5` ciphers. However, a man-in-the-middle could read and pass communications between client and server. Also, encryption overhead is minimal compared to the overhead of authentication. For these reasons NULL ciphers are not recommended.

To start in SSL mode, the files `server.crt` and `server.key` must exist in the server’s data directory. These files should contain the server certificate and private key, respectively. If the private key is protected with a passphrase, the server will prompt for the passphrase and will not start until it has been entered.

To require the client to supply a trusted certificate, place certificates of the certificate authorities (CA) you trust in the file `root.crt` in the data directory. A certificate will then be requested from the client during SSL connection startup. (See Section 30.16 for a description of how to set up client certificates.) The server will verify that the client’s certificate is signed by one of the trusted certificate authorities. Certificate Revocation List (CRL) entries are also checked if the file `root.crl` exists. (See http://h71000.www7.hp.com/DOC/83final/BA554_90007/ch04s02.html for diagrams showing SSL certificate usage.)

If the `root.crt` file is not present, client certificates will not be requested or checked. In this mode, SSL provides encrypted communication but not authentication.

The files `server.key`, `server.crt`, `root.crt`, and `root.crl` are only examined during server start; so you must restart the server for changes in them to take effect.
Table 17-3. SSL Server File Usage

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>server.crt</td>
<td>server certificate</td>
<td>requested by client</td>
</tr>
<tr>
<td>server.key</td>
<td>server private key</td>
<td>proves server certificate sent by owner; does not indicate certificate owner is trustworthy</td>
</tr>
<tr>
<td>root.crt</td>
<td>trusted certificate authorities</td>
<td>requests client certificate; checks certificate is signed by a trusted certificate authority</td>
</tr>
<tr>
<td>root.crl</td>
<td>certificates revoked by certificate authorities</td>
<td>client certificate must not be on this list</td>
</tr>
</tbody>
</table>

17.8.1. Creating a Self-Signed Certificate

To create a quick self-signed certificate for the server, use the following OpenSSL command:

openssl req -new -text -out server.req

Fill out the information that openssl asks for. Make sure you enter the local host name as “Common Name”; the challenge password can be left blank. The program will generate a key that is passphrase protected; it will not accept a passphrase that is less than four characters long. To remove the passphrase (as you must if you want automatic start-up of the server), run the commands:

openssl rsa -in privkey.pem -out server.key
rm privkey.pem

Enter the old passphrase to unlock the existing key. Now do:

openssl req -x509 -in server.req -text -key server.key -out server.crt
chmod og-rwx server.key

to turn the certificate into a self-signed certificate and to copy the key and certificate to where the server will look for them. For more details on how to create your server private key and certificate, refer to the OpenSSL documentation.

A self-signed certificate can be used for testing, but a certificate signed by a certificate authority (CA) (either one of the global CAs or a local one) should be used in production so the client can verify the server’s identity.

17.9. Secure TCP/IP Connections with SSH Tunnels

One can use SSH to encrypt the network connection between clients and a PostgreSQL server. Done properly, this provides an adequately secure network connection, even for non-SSL-capable clients.

First make sure that an SSH server is running properly on the same machine as the PostgreSQL server and that you can log in using `ssh` as some user. Then you can establish a secure tunnel with a command like this from the client machine:

```
ssh -L 3333:foo.com:5432 joe@foo.com
```
The first number in the `-L` argument, 3333, is the port number of your end of the tunnel; it can be chosen freely. The second number, 5432, is the remote end of the tunnel: the port number your server is using. The name or IP address between the port numbers is the host with the database server you are going to connect to. In order to connect to the database server using this tunnel, you connect to port 3333 on the local machine:

```
psql -h localhost -p 3333 postgres
```

To the database server it will then look as though you are really user joe@foo.com and it will use whatever authentication procedure was configured for connections from this user and host. Note that the server will not think the connection is SSL-encrypted, since in fact it is not encrypted between the SSH server and the PostgreSQL server. This should not pose any extra security risk as long as they are on the same machine.

In order for the tunnel setup to succeed you must be allowed to connect via `ssh` as joe@foo.com, just as if you had attempted to use `ssh` to set up a terminal session.

**Tip:** Several other applications exist that can provide secure tunnels using a procedure similar in concept to the one just described.
Chapter 18. Server Configuration

There are many configuration parameters that affect the behavior of the database system. In the first section of this chapter, we describe how to set configuration parameters. The subsequent sections discuss each parameter in detail.

18.1. Setting Parameters

All parameter names are case-insensitive. Every parameter takes a value of one of four types: Boolean, integer, floating point, or string. Boolean values can be written as ON, OFF, TRUE, FALSE, YES, NO, 1, 0 (all case-insensitive) or any unambiguous prefix of these.

Some settings specify a memory or time value. Each of these has an implicit unit, which is either kilobytes, blocks (typically eight kilobytes), milliseconds, seconds, or minutes. Default units can be queried by referencing pg_settings.unit. For convenience, a different unit can also be specified explicitly. Valid memory units are kB (kilobytes), MB (megabytes), and GB (gigabytes); valid time units are ms (milliseconds), s (seconds), min (minutes), h (hours), and d (days). Note that the multiplier for memory units is 1024, not 1000.

One way to set these parameters is to edit the file postgresql.conf, which is normally kept in the data directory. (initdb installs a default copy there.) An example of what this file might look like is:

```plaintext
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '$user', public'
shared_buffers = 128MB
```

One parameter is specified per line. The equal sign between name and value is optional. Whitespace is insignificant and blank lines are ignored. Hash marks (#) introduce comments anywhere. Parameter values that are not simple identifiers or numbers must be single-quoted. To embed a single quote in a parameter value, write either two quotes (preferred) or backslash-quote.

In addition to parameter settings, the postgresql.conf file can contain include directives, which specify another file to read and process as if it were inserted into the configuration file at this point. Include directives simply look like:

```plaintext
include 'filename'
```

If the file name is not an absolute path, it is taken as relative to the directory containing the referencing configuration file. Inclusions can be nested.

The configuration file is reread whenever the main server process receives a SIGHUP signal (which is most easily sent by means of pg_ctl reload). The main server process also propagates this signal to all currently running server processes so that existing sessions also get the new value. Alternatively, you can send the signal to a single server process directly. Some parameters can only be set at server start; any changes to their entries in the configuration file will be ignored until the server is restarted.

A second way to set these configuration parameters is to give them as a command-line option to the postgres command, such as:

```plaintext
postgres -c log_connections=yes -c log_destination='syslog'
```
Chapter 18. Server Configuration

Command-line options override any conflicting settings in `postgresql.conf`. Note that this means you won’t be able to change the value on-the-fly by editing `postgresql.conf`, so while the command-line method might be convenient, it can cost you flexibility later.

Occasionally it is useful to give a command line option to one particular session only. The environment variable `PGOPTIONS` can be used for this purpose on the client side:

```bash
eval PGOPTIONS=''-c geqo=off'' psql
```

(This works for any `libpq`-based client application, not just `psql`.) Note that this won’t work for parameters that are fixed when the server is started or that must be specified in `postgresql.conf`.

Furthermore, it is possible to assign a set of parameter settings to a user or a database. Whenever a session is started, the default settings for the user and database involved are loaded. The commands `ALTER USER` and `ALTER DATABASE`, respectively, are used to configure these settings. Per-database settings override anything received from the `postgres` command-line or the configuration file, and in turn are overridden by per-user settings; both are overridden by per-session settings.

Some parameters can be changed in individual SQL sessions with the `SET` command, for example:

```sql
SET ENABLE_SEQSCAN TO OFF;
```

If `SET` is allowed, it overrides all other sources of values for the parameter. Some parameters cannot be changed via `SET`: for example, if they control behavior that cannot be changed without restarting the entire PostgreSQL server. Also, some parameters can be modified via `SET` or `ALTER` by superusers, but not by ordinary users.

The `SHOW` command allows inspection of the current values of all parameters.

The virtual table `pg_settings` (described in Section 44.51) also allows displaying and updating session run-time parameters. It is equivalent to `SHOW` and `SET`, but can be more convenient to use because it can be joined with other tables, or selected from using any desired selection condition.

18.2. File Locations

In addition to the `postgresql.conf` file already mentioned, PostgreSQL uses two other manually-edited configuration files, which control client authentication (their use is discussed in Chapter 21). By default, all three configuration files are stored in the database cluster’s data directory. The parameters described in this section allow the configuration files to be placed elsewhere. (Doing so can ease administration. In particular it is often easier to ensure that the configuration files are properly backed-up when they are kept separate.)

`data_directory` (string)

Specifies the directory to use for data storage. This parameter can only be set at server start.

`config_file` (string)

Specifies the main server configuration file (customarily called `postgresql.conf`). This parameter can only be set on the `postgres` command line.

`hba_file` (string)

Specifies the configuration file for host-based authentication (customarily called `pg_hba.conf`). This parameter can only be set at server start.
ident_file(string)

Specifies the configuration file for ident authentication (customarily called pg_ident.conf). This parameter can only be set at server start.

external_pid_file(string)

Specifies the name of an additional process-id (PID) file that the server should create for use by server administration programs. This parameter can only be set at server start.

In a default installation, none of the above parameters are set explicitly. Instead, the data directory is specified by the -D command-line option or the PGDATA environment variable, and the configuration files are all found within the data directory.

If you wish to keep the configuration files elsewhere than the data directory, the postgres -D command-line option or PGDATA environment variable must point to the directory containing the configuration files, and the data_directory parameter must be set in postgresql.conf (or on the command line) to show where the data directory is actually located. Notice that data_directory overrides -D and PGDATA for the location of the data directory, but not for the location of the configuration files.

If you wish, you can specify the configuration file names and locations individually using the parameters config_file, hba_file and/or ident_file. config_file can only be specified on the postgres command line, but the others can be set within the main configuration file. If all three parameters plus data_directory are explicitly set, then it is not necessary to specify -D or PGDATA.

When setting any of these parameters, a relative path will be interpreted with respect to the directory in which postgres is started.

Chapter 18. Server Configuration

18.3. Connections and Authentication

18.3.1. Connection Settings

listen_addresses(string)

Specifies the TCP/IP address(es) on which the server is to listen for connections from client applications. The value takes the form of a comma-separated list of host names and/or numeric IP addresses. The special entry * corresponds to all available IP interfaces. If the list is empty, the server does not listen on any IP interface at all, in which case only Unix-domain sockets can be used to connect to it. The default value is localhost, which allows only local “loopback” connections to be made. This parameter can only be set at server start.

port(integer)

The TCP port the server listens on; 5432 by default. Note that the same port number is used for all IP addresses the server listens on. This parameter can only be set at server start.

max_connections(integer)

Determines the maximum number of concurrent connections to the database server. The default is typically 100 connections, but might be less if your kernel settings will not support it (as determined during initdb). This parameter can only be set at server start.

Increasing this parameter might cause PostgreSQL to request more System V shared memory or semaphores than your operating system’s default configuration allows. See Section 17.4.1 for information on how to adjust those parameters, if necessary.
superuser_reserved_connections (integer)

Determines the number of connection “slots” that are reserved for connections by PostgreSQL superusers. At most max_connections connections can ever be active simultaneously. Whenever the number of active concurrent connections is at least max_connections minus superuser_reserved_connections, new connections will be accepted only for superusers.

The default value is three connections. The value must be less than the value of max_connections. This parameter can only be set at server start.

unix_socket_directory (string)

Specifies the directory of the Unix-domain socket on which the server is to listen for connections from client applications. The default is normally /tmp, but can be changed at build time. This parameter can only be set at server start.

unix_socket_group (string)

Sets the owning group of the Unix-domain socket. (The owning user of the socket is always the user that starts the server.) In combination with the parameter unix_socket_permissions this can be used as an additional access control mechanism for Unix-domain connections. By default this is the empty string, which selects the default group for the current user. This parameter can only be set at server start.

unix_socket_permissions (integer)

Sets the access permissions of the Unix-domain socket. Unix-domain sockets use the usual Unix file system permission set. The parameter value is expected to be a numeric mode specification in the form accepted by the chmod and umask system calls. (To use the customary octal format the number must start with a 0 (zero).)

The default permissions are 0777, meaning anyone can connect. Reasonable alternatives are 0770 (only user and group, see also unix_socket_group) and 0700 (only user). (Note that for a Unix-domain socket, only write permission matters and so there is no point in setting or revoking read or execute permissions.)

This access control mechanism is independent of the one described in Chapter 21.

This parameter can only be set at server start.

bonjour_name (string)

Specifies the Bonjour broadcast name. The computer name is used if this parameter is set to the empty string “” (which is the default). This parameter is ignored if the server was not compiled with Bonjour support. This parameter can only be set at server start.

tcp_keepalives_idle (integer)

On systems that support the TCP_KEEPIDLE socket option, specifies the number of seconds between sending keepalives on an otherwise idle connection. A value of zero uses the system default. If TCP_KEEPIDLE is not supported, this parameter must be zero. This parameter is ignored for connections made via a Unix-domain socket.

tcp_keepalives_interval (integer)

On systems that support the TCP_KEEPINTVL socket option, specifies how long, in seconds, to wait for a response to a keepalive before retransmitting. A value of zero uses the system default. If TCP_KEEPINTVL is not supported, this parameter must be zero. This parameter is ignored for connections made via a Unix-domain socket.
tcp_keepalives_count (integer)

On systems that support the TCP_KEEPCNT socket option, specifies how many keepalives can be lost before the connection is considered dead. A value of zero uses the system default. If TCP_KEEPCNT is not supported, this parameter must be zero. This parameter is ignored for connections made via a Unix-domain socket.

18.3.2. Security and Authentication

authentication_timeout (integer)

Maximum time to complete client authentication, in seconds. If a would-be client has not completed the authentication protocol in this much time, the server breaks the connection. This prevents hung clients from occupying a connection indefinitely. The default is one minute (1m). This parameter can only be set in the postgresql.conf file or on the server command line.

ssl (boolean)

Enables SSL connections. Please read Section 17.8 before using this. The default is off. This parameter can only be set at server start.

ssl_renegotiation_limit (integer)

Specifies how much data can flow over an SSL encrypted connection before renegotiation of the session will take place. Renegotiation of the session decreases the chance of doing cryptanalysis when large amounts of data are sent, but it also carries a large performance penalty. The sum of sent and received traffic is used to check the limit. If the parameter is set to 0, renegotiation is disabled. The default is 512MB.

Note: SSL libraries from before November 2009 are insecure when using SSL renegotiation, due to a vulnerability in the SSL protocol. As a stop-gap fix for this vulnerability, some vendors also shipped SSL libraries incapable of doing renegotiation. If any of these libraries are in use on the client or server, SSL renegotiation should be disabled.

ssl_ciphers (string)

Specifies a list of SSL ciphers that are allowed to be used on secure connections. See the openssl manual page for a list of supported ciphers. This parameter is unavailable unless the server is compiled with support for SSL.

password_encryption (boolean)

When a password is specified in CREATE USER or ALTER USER without writing either ENCRYPTED or UNENCRYPTED, this parameter determines whether the password is to be encrypted. The default is on (encrypt the password).

krb_realm (string)

Sets the realm to match Kerberos, GSSAPI and SSPI user names against. See Section 21.2.5, Section 21.2.3 or Section 21.2.4 for details. This parameter can only be set at server start.

krb_server_keyfile (string)

Sets the location of the Kerberos server key file. See Section 21.2.5 or Section 21.2.3 for details. This parameter can only be set at server start.
krb_srvname (string)

Sets the Kerberos service name. See Section 21.2.5 for details. This parameter can only be set at server start.

krb_server_hostname (string)

Sets the host name part of the service principal. This, combined with krb_srvname, is used to generate the complete service principal, that is krb_srvname/krb_server_hostname@REALM. If not set, the default is the server hostname. See Section 21.2.5 for details. This parameter can only be set at server start.

krb_caseins_users (boolean)

Sets whether Kerberos and GSSAPI user names should be treated case-insensitively. The default is off (case sensitive). This parameter can only be set at server start.

db_user_namespace (boolean)

This parameter enables per-database user names. It is off by default. This parameter can only be set in the postgresql.conf file or on the server command line.

If this is on, you should create users as username@dbname. When username is passed by a connecting client, @ and the database name are appended to the user name and that database-specific user name is looked up by the server. Note that when you create users with names containing @ within the SQL environment, you will need to quote the user name.

With this parameter enabled, you can still create ordinary global users. Simply append @ when specifying the user name in the client. The @ will be stripped off before the user name is looked up by the server.

Note: This feature is intended as a temporary measure until a complete solution is found. At that time, this option will be removed.

18.4. Resource Consumption

18.4.1. Memory

shared_buffers (integer)

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 32 megabytes (32MB), but might be less if your kernel settings will not support it (as determined during initdb). This setting must be at least 128 kilobytes and at least 16 kilobytes times max_connections. (Non-default values of BLCKSZ change the minimum.) However, settings significantly higher than the minimum are usually needed for good performance. This parameter can only be set at server start.

If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for shared_buffers is 25% of the memory in your system. There are some workloads where even large settings for shared_buffers are effective, but because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to
shared_buffers will work better than a smaller amount. Larger settings for shared_buffers usually require a corresponding increase in checkpoint_segments, in order to spread out the process of writing large quantities of new or changed data over a longer period of time.

On systems with less than 1GB of RAM, a smaller percentage of RAM is appropriate, so as to leave adequate space for the operating system. Also, on Windows, large values for shared_buffers aren’t as effective. You may find better results keeping the setting relatively low and using the operating system cache more instead. The useful range for shared_buffers on Windows systems is generally from 64MB to 512MB.

Increasing this parameter might cause PostgreSQL to request more System V shared memory than your operating system’s default configuration allows. See Section 17.4.1 for information on how to adjust those parameters, if necessary.

temp_buffers (integer)

Sets the maximum number of temporary buffers used by each database session. These are session-local buffers used only for access to temporary tables. The default is eight megabytes (8MB). The setting can be changed within individual sessions, but only up until the first use of temporary tables within a session; subsequent attempts to change the value will have no effect on that session.

A session will allocate temporary buffers as needed up to the limit given by temp_buffers. The cost of setting a large value in sessions that do not actually need a lot of temporary buffers is only a buffer descriptor, or about 64 bytes, per increment in temp_buffers. However if a buffer is actually used an additional 8192 bytes will be consumed for it (or in general, BLCKSZ bytes).

max_prepared_transactions (integer)

Sets the maximum number of transactions that can be in the “prepared” state simultaneously (see PREPARE TRANSACTION). Setting this parameter to zero disables the prepared-transaction feature. The default is five transactions. This parameter can only be set at server start.

If you are not using prepared transactions, this parameter may as well be set to zero. If you are using them, you will probably want max_prepared_transactions to be at least as large as max_connections, to avoid unwanted failures at the prepare step.

Increasing this parameter might cause PostgreSQL to request more System V shared memory than your operating system’s default configuration allows. See Section 17.4.1 for information on how to adjust those parameters, if necessary.

work_mem (integer)

Specifies the amount of memory to be used by internal sort operations and hash tables before switching to temporary disk files. The value is defaults to one megabyte (1MB). Note that for a complex query, several sort or hash operations might be running in parallel; each one will be allowed to use as much memory as this value specifies before it starts to put data into temporary files. Also, several running sessions could be doing such operations concurrently. So the total memory used could be many times the value of work_mem; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for ORDER BY, DISTINCT, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of IN subqueries.

maintenance_work_mem (integer)

Specifies the maximum amount of memory to be used in maintenance operations, such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. It defaults to 16 megabytes (16MB). Since only one of these operations can be executed at a time by a database session, and an installation normally doesn’t have many of them running concurrently, it’s safe to set this
value significantly larger than `work_mem`. Larger settings might improve performance for vacuuming and for restoring database dumps.

Note that when autovacuum runs, up to `autovacuum_max_workers` times this memory may be allocated, so be careful not to set the default value too high.

```sql
max_stack_depth (integer)
```

Specifies the maximum safe depth of the server’s execution stack. The ideal setting for this parameter is the actual stack size limit enforced by the kernel (as set by `ulimit -s` or local equivalent), less a safety margin of a megabyte or so. The safety margin is needed because the stack depth is not checked in every routine in the server, but only in key potentially-recursive routines such as expression evaluation. The default setting is two megabytes (2MB), which is conservatively small and unlikely to risk crashes. However, it might be too small to allow execution of complex functions. Only superusers can change this setting.

Setting `max_stack_depth` higher than the actual kernel limit will mean that a runaway recursive function can crash an individual backend process. On platforms where PostgreSQL can determine the kernel limit, it will not let you set this variable to an unsafe value. However, not all platforms provide the information, so caution is recommended in selecting a value.

### 18.4.2. Free Space Map

These parameters control the size of the shared free space map (FSM), which tracks the locations of unused space in the database. An undersized free space map can cause the database to consume increasing amounts of disk space over time, because free space that is not in the map cannot be reused; instead PostgreSQL will request more disk space from the operating system when it needs to store new data. The last few lines displayed by a database-wide `VACUUM VERBOSE` command can help in determining if the current settings are adequate. A `NOTICE` message is also printed during such an operation if the current settings are too low.

Increasing these parameters might cause PostgreSQL to request more System V shared memory than your operating system’s default configuration allows. See Section 17.4.1 for information on how to adjust those parameters, if necessary.

```sql
max_fsm_pages (integer)
```

Sets the maximum number of disk pages for which free space will be tracked in the shared free-space map. Six bytes of shared memory are consumed for each page slot. This setting must be at least 16 * `max_fsm_relations`. The default is chosen by initdb depending on the amount of available memory, and can range from 20k to 200k pages. This parameter can only be set at server start.

```sql
max_fsm_relations (integer)
```

Sets the maximum number of relations (tables and indexes) for which free space will be tracked in the shared free-space map. Roughly seventy bytes of shared memory are consumed for each slot. The default is one thousand relations. This parameter can only be set at server start.

**Note:** See the `VACUUM` command for information on setting this parameter.
18.4.3. Kernel Resource Usage

**max_files_per_process (integer)**

Sets the maximum number of simultaneously open files allowed to each server subprocess. The default is one thousand files. If the kernel is enforcing a safe per-process limit, you don’t need to worry about this setting. But on some platforms (notably, most BSD systems), the kernel will allow individual processes to open many more files than the system can really support when a large number of processes all try to open that many files. If you find yourself seeing “Too many open files” failures, try reducing this setting. This parameter can only be set at server start.

**shared_preload_libraries (string)**

This variable specifies one or more shared libraries that are to be preloaded at server start. If more than one library is to be loaded, separate their names with commas. For example, ‘$libdir/mylib’ would cause mylib.so (or on some platforms, mylib.sl) to be preloaded from the installation’s standard library directory. This parameter can only be set at server start.

PostgreSQL procedural language libraries can be preloaded in this way, typically by using the syntax ‘$libdir/plXXX’ where XXX is pgsql, perl, tcl, or python.

By preloading a shared library, the library startup time is avoided when the library is first used. However, the time to start each new server process might increase slightly, even if that process never uses the library. So this parameter is recommended only for libraries that will be used in most sessions.

**Note:** On Windows hosts, preloading a library at server start will not reduce the time required to start each new server process; each server process will re-load all preload libraries. However, shared_preload_libraries is still useful on Windows hosts because some shared libraries may need to perform certain operations that only take place at postmaster start (for example, a shared library may need to reserve lightweight locks or shared memory and you can’t do that after the postmaster has started).

If a specified library is not found, the server will fail to start.

Every PostgreSQL-supported library has a “magic block” that is checked to guarantee compatibility. For this reason, non-PostgreSQL libraries cannot be loaded in this way.

18.4.4. Cost-Based Vacuum Delay

During the execution of `VACUUM` and `ANALYZE` commands, the system maintains an internal counter that keeps track of the estimated cost of the various I/O operations that are performed. When the accumulated cost reaches a limit (specified by `vacuum_cost_limit`), the process performing the operation will sleep for a while (specified by `vacuum_cost_delay`). Then it will reset the counter and continue execution.

The intent of this feature is to allow administrators to reduce the I/O impact of these commands on concurrent database activity. There are many situations in which it is not very important that maintenance commands like `VACUUM` and `ANALYZE` finish quickly; however, it is usually very important that these commands do not significantly interfere with the ability of the system to perform other database operations. Cost-based vacuum delay provides a way for administrators to achieve this.

This feature is disabled by default. To enable it, set the `vacuum_cost_delay` variable to a nonzero value.
vacuum_cost_delay (integer)

The length of time, in milliseconds, that the process will sleep when the cost limit has been exceeded. The default value is zero, which disables the cost-based vacuum delay feature. Positive values enable cost-based vacuuming. Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting vacuum_cost_delay to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10.

vacuum_cost_page_hit (integer)

The estimated cost for vacuuming a buffer found in the shared buffer cache. It represents the cost to lock the buffer pool, lookup the shared hash table and scan the content of the page. The default value is one.

vacuum_cost_page_miss (integer)

The estimated cost for vacuuming a buffer that has to be read from disk. This represents the effort to lock the buffer pool, lookup the shared hash table, read the desired block in from the disk and scan its content. The default value is 10.

vacuum_cost_page_dirty (integer)

The estimated cost charged when vacuum modifies a block that was previously clean. It represents the extra I/O required to flush the dirty block out to disk again. The default value is 20.

vacuum_cost_limit (integer)

The accumulated cost that will cause the vacuuming process to sleep. The default value is 200.

Note: There are certain operations that hold critical locks and should therefore complete as quickly as possible. Cost-based vacuum delays do not occur during such operations. Therefore it is possible that the cost accumulates far higher than the specified limit. To avoid uselessly long delays in such cases, the actual delay is calculated as vacuum_cost_delay * accumulated_balance / vacuum_cost_limit with a maximum of vacuum_cost_delay * 4.

18.4.5. Background Writer

There is a separate server process called the background writer, whose function is to issue writes of “dirty” shared buffers. The intent is that server processes handling user queries should seldom or never have to wait for a write to occur, because the background writer will do it. However there is a net overall increase in I/O load, because a repeatedly-dirtied page might otherwise be written only once per checkpoint interval, but the background writer might write it several times in the same interval. The parameters discussed in this subsection can be used to tune the behavior for local needs.

bgwriter_delay (integer)

Specifies the delay between activity rounds for the background writer. In each round the writer issues writes for some number of dirty buffers (controllable by the following parameters). It then sleeps for bgwriter_delay milliseconds, and repeats. The default value is 200 milliseconds (200ms). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting bgwriter_delay to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10. This parameter can only be set in the postgresql.conf file or on the server command line.
bgwriter_lru_maxpages (integer)

In each round, no more than this many buffers will be written by the background writer. Setting this to zero disables background writing (except for checkpoint activity). The default value is 100 buffers. This parameter can only be set in the `postgresql.conf` file or on the server command line.

bgwriter_lru_multiplier (floating point)

The number of dirty buffers written in each round is based on the number of new buffers that have been needed by server processes during recent rounds. The average recent need is multiplied by `bgwriter_lru_multiplier` to arrive at an estimate of the number of buffers that will be needed during the next round. Dirty buffers are written until there are that many clean, reusable buffers available. (However, no more than `bgwriter_lru_maxpages` buffers will be written per round.) Thus, a setting of 1.0 represents a “just in time” policy of writing exactly the number of buffers predicted to be needed. Larger values provide some cushion against spikes in demand, while smaller values intentionally leave writes to be done by server processes. The default is 2.0. This parameter can only be set in the `postgresql.conf` file or on the server command line.

Smaller values of `bgwriter_lru_maxpages` and `bgwriter_lru_multiplier` reduce the extra I/O load caused by the background writer, but make it more likely that server processes will have to issue writes for themselves, delaying interactive queries.

18.5. Write Ahead Log

See also Section 28.4 for details on WAL and checkpoint tuning.

18.5.1. Settings

fsync (boolean)

If this parameter is on, the PostgreSQL server will try to make sure that updates are physically written to disk, by issuing `fsync()` system calls or various equivalent methods (see `wal_sync_method`). This ensures that the database cluster can recover to a consistent state after an operating system or hardware crash.

However, using `fsync` results in a performance penalty: when a transaction is committed, PostgreSQL must wait for the operating system to flush the write-ahead log to disk. When `fsync` is disabled, the operating system is allowed to do its best in buffering, ordering, and delaying writes. This can result in significantly improved performance. However, if the system crashes, the results of the last few committed transactions might be lost in part or whole. In the worst case, unrecoverable data corruption might occur. (Crashes of the database software itself are not a risk factor here. Only an operating-system-level crash creates a risk of corruption.)

Due to the risks involved, there is no universally correct setting for `fsync`. Some administrators always disable `fsync`, while others only turn it off during initial bulk data loads, where there is a clear restart point if something goes wrong. Others always leave `fsync` enabled. The default is to enable `fsync`, for maximum reliability. If you trust your operating system, your hardware, and your utility company (or your battery backup), you can consider disabling `fsync`.

In many situations, turning off `synchronous_commit` for noncritical transactions can provide much of the potential performance benefit of turning off `fsync`, without the attendant risks of data corruption.
Chapter 18. Server Configuration

This parameter can only be set in the `postgresql.conf` file or on the server command line. If you turn this parameter off, also consider turning off `full_page_writes`.

```plaintext
synchronous_commit (boolean)
```

Specifies whether transaction commit will wait for WAL records to be written to disk before the command returns a “success” indication to the client. The default, and safe, setting is `on`. When `off`, there can be a delay between when success is reported to the client and when the transaction is really guaranteed to be safe against a server crash. (The maximum delay is three times `wal_writer_delay`.) Unlike `fsync`, setting this parameter to `off` does not create any risk of database inconsistency: a crash might result in some recent allegedly-committed transactions being lost, but the database state will be just the same as if those transactions had been aborted cleanly. So, turning `synchronous_commit` off can be a useful alternative when performance is more important than exact certainty about the durability of a transaction. For more discussion see Section 28.3.

This parameter can be changed at any time; the behavior for any one transaction is determined by the setting in effect when it commits. It is therefore possible, and useful, to have some transactions commit synchronously and others asynchronously. For example, to make a single multi-statement transaction commit asynchronously when the default is the opposite, issue `SET LOCAL synchronous_commit TO OFF` within the transaction.

```plaintext
wal_sync_method (string)
```

Method used for forcing WAL updates out to disk. If `fsync` is off then this setting is irrelevant, since updates will not be forced out at all. Possible values are:

- `open_datasync` (write WAL files with `open()` option `O_DSYNC`)
- `fdatasync` (call `fdatasync()` at each commit)
- `fsync` (call `fsync()` at each commit)
- `fsync_writethrough` (call `fsync()` at each commit, forcing write-through of any disk write cache)
- `open_sync` (write WAL files with `open()` option `O_SYNC`)

The `open_*` options also use `O_DIRECT` if available. Not all of these choices are available on all platforms. The default is the first method in the above list that is supported by the platform, except that `fdatasync` is the default on Linux. The default is not necessarily ideal; it might be necessary to change this setting or other aspects of your system configuration in order to create a crash-safe configuration or achieve optimal performance. These aspects are discussed in Section 28.1. This parameter can only be set in the `postgresql.conf` file or on the server command line.

```plaintext
full_page_writes (boolean)
```

When this parameter is on, the PostgreSQL server writes the entire content of each disk page to WAL during the first modification of that page after a checkpoint. This is needed because a page write that is in process during an operating system crash might be only partially completed, leading to an on-disk page that contains a mix of old and new data. The row-level change data normally stored in WAL will not be enough to completely restore such a page during post-crash recovery. Storing the full page image guarantees that the page can be correctly restored, but at a price in increasing the amount of data that must be written to WAL. (Because WAL replay always starts from a checkpoint, it is sufficient to do this during the first change of each page after a checkpoint. Therefore, one way to reduce the cost of full-page writes is to increase the checkpoint interval parameters.)
Turning this parameter off speeds normal operation, but might lead to a corrupt database after an operating system crash or power failure. The risks are similar to turning off fsync, though smaller. It might be safe to turn off this parameter if you have hardware (such as a battery-backed disk controller) or file-system software that reduces the risk of partial page writes to an acceptably low level (e.g., ReiserFS 4).

Turning off this parameter does not affect use of WAL archiving for point-in-time recovery (PITR) (see Section 24.3).

This parameter can only be set in the postgresql.conf file or on the server command line. The default is on.

### wal_buffers

The amount of memory used in shared memory for WAL data. The default is 64 kilobytes (64kB). The setting need only be large enough to hold the amount of WAL data generated by one typical transaction, since the data is written out to disk at every transaction commit. This parameter can only be set at server start.

Increasing this parameter might cause PostgreSQL to request more System V shared memory than your operating system’s default configuration allows. See Section 17.4.1 for information on how to adjust those parameters, if necessary.

### wal_writer_delay

Specifies the delay between activity rounds for the WAL writer. In each round the writer will flush WAL to disk. It then sleeps for wal_writer_delay milliseconds, and repeats. The default value is 200 milliseconds (200ms). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting wal_writer_delay to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10. This parameter can only be set in the postgresql.conf file or on the server command line.

### commit_delay

Time delay between writing a commit record to the WAL buffer and flushing the buffer out to disk, in microseconds. A nonzero delay can allow multiple transactions to be committed with only one fsync() system call, if system load is high enough that additional transactions become ready to commit within the given interval. But the delay is just wasted if no other transactions become ready to commit. Therefore, the delay is only performed if at least commit_siblings other transactions are active at the instant that a server process has written its commit record. The default is zero (no delay).

### commit_siblings

Minimum number of concurrent open transactions to require before performing the commit_delay delay. A larger value makes it more probable that at least one other transaction will become ready to commit during the delay interval. The default is five transactions.

### 18.5.2. Checkpoints

### checkpoint_segments

Maximum number of log file segments between automatic WAL checkpoints (each segment is normally 16 megabytes). The default is three segments. Increasing this parameter can increase the amount of time needed for crash recovery. This parameter can only be set in the postgresql.conf file or on the server command line.
checkpoint_timeout (integer)

Maximum time between automatic WAL checkpoints, in seconds. The default is five minutes (5min). Increasing this parameter can increase the amount of time needed for crash recovery. This parameter can only be set in the `postgresql.conf` file or on the server command line.

checkpoint_completion_target (floating point)

Specifies the target length of checkpoints, as a fraction of the checkpoint interval. The default is 0.5. This parameter can only be set in the `postgresql.conf` file or on the server command line.

checkpoint_warning (integer)

Write a message to the server log if checkpoints caused by the filling of checkpoint segment files happen closer together than this many seconds (which suggests that `checkpoint_segments` ought to be raised). The default is 30 seconds (30s). Zero disables the warning. This parameter can only be set in the `postgresql.conf` file or on the server command line.

18.5.3. Archiving

archive_mode (boolean)

When `archive_mode` is enabled, completed WAL segments can be sent to archive storage by setting `archive_command`. `archive_mode` and `archive_command` are separate variables so that `archive_command` can be changed without leaving archiving mode. This parameter can only be set at server start.

archive_command (string)

The shell command to execute to archive a completed segment of the WAL file series. Any %p in the string is replaced by the path name of the file to archive, and any %f is replaced by the file name only. (The path name is relative to the working directory of the server, i.e., the cluster’s data directory.) Use %% to embed an actual % character in the command. For more information see Section 24.3.1. This parameter can only be set in the `postgresql.conf` file or on the server command line. It is ignored unless `archive_mode` was enabled at server start. If `archive_command` is an empty string (the default) while `archive_mode` is enabled, then WAL archiving is temporarily disabled, but the server continues to accumulate WAL segment files in the expectation that a command will soon be provided.

It is important for the command to return a zero exit status if and only if it succeeds. Examples:

```
archive_command = 'cp "%p" /mnt/server/archivedir/"%f"
archive_command = 'copy "%p" "C:\server\archivedir\%f"'  # Windows
```

archive_timeout (integer)

The `archive_command` is only invoked on completed WAL segments. Hence, if your server generates little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To put a limit on how old unarchived data can be, you can set `archive_timeout` to force the server to switch to a new WAL segment file periodically. When this parameter is greater than zero, the server will switch to a new segment file whenever this many seconds have elapsed since the last segment file switch. Note that archived files that are closed early due to a forced switch are still the same length as completely full files. Therefore, it is unwise to use a very short `archive_timeout` — it will bloat your archive storage. `archive_timeout` settings of a minute or so are usually reasonable. This parameter can only be set in the `postgresql.conf` file or on the server command line.
18.6. Query Planning

18.6.1. Planner Method Configuration

These configuration parameters provide a crude method of influencing the query plans chosen by the query optimizer. If the default plan chosen by the optimizer for a particular query is not optimal, a temporary solution can be found by using one of these configuration parameters to force the optimizer to choose a different plan. Turning one of these settings off permanently is seldom a good idea, however. Better ways to improve the quality of the plans chosen by the optimizer include adjusting the Planner Cost Constants, running ANALYZE more frequently, increasing the value of the default_statistics_target configuration parameter, and increasing the amount of statistics collected for specific columns using ALTER TABLE SET STATISTICS.

`enable_bitmapscan (boolean)`
Enables or disables the query planner’s use of bitmap-scan plan types. The default is on.

`enable_hashagg (boolean)`
Enables or disables the query planner’s use of hashed aggregation plan types. The default is on.

`enable_hashjoin (boolean)`
Enables or disables the query planner’s use of hash-join plan types. The default is on.

`enable_indexscan (boolean)`
Enables or disables the query planner’s use of index-scan plan types. The default is on.

`enable_mergejoin (boolean)`
Enables or disables the query planner’s use of merge-join plan types. The default is on.

`enable_nestloop (boolean)`
Enables or disables the query planner’s use of nested-loop join plans. It’s not possible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is on.

`enable_seqscan (boolean)`
Enables or disables the query planner’s use of sequential scan plan types. It’s not possible to suppress sequential scans entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is on.

`enable_sort (boolean)`
Enables or disables the query planner’s use of explicit sort steps. It’s not possible to suppress explicit sorts entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is on.

`enable_tidscan (boolean)`
Enables or disables the query planner’s use of TID scan plan types. The default is on.

18.6.2. Planner Cost Constants

The cost variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner’s choices. Traditionally, these variables have been referenced to sequential page fetches as the...
unit of cost; that is, `seq_page_cost` is conventionally set to 1.0 and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

**Note:** Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will get. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost` *(floating point)*

Sets the planner’s estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0.

`random_page_cost` *(floating point)*

Sets the planner’s estimate of the cost of a non-sequentially-fetched disk page. The default is 4.0. Reducing this value relative to `seq_page_cost` will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs, which are described by the following parameters.

**Tip:** Although the system will let you set `random_page_cost` to less than `seq_page_cost`, it is not physically sensible to do so. However, setting them equal makes sense if the database is entirely cached in RAM, since in that case there is no penalty for touching pages out of sequence. Also, in a heavily-cached database you should lower both values relative to the CPU parameters, since the cost of fetching a page already in RAM is much smaller than it would normally be.

`cpu_tuple_cost` *(floating point)*

Sets the planner’s estimate of the cost of processing each row during a query. The default is 0.01.

`cpu_index_tuple_cost` *(floating point)*

Sets the planner’s estimate of the cost of processing each index entry during an index scan. The default is 0.005.

`cpu_operator_cost` *(floating point)*

Sets the planner’s estimate of the cost of processing each operator or function executed during a query. The default is 0.0025.

`effective_cache_size` *(integer)*

Sets the planner’s assumption about the effective size of the disk cache that is available to a single query. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both PostgreSQL’s shared buffers and the portion of the kernel’s disk cache that will be used for PostgreSQL data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by PostgreSQL, nor does it reserve kernel disk cache; it is used only for estimation purposes. The default is 128 megabytes (128MB).
18.6.3. Genetic Query Optimizer

geqo (boolean)

Enables or disables genetic query optimization, which is an algorithm that attempts to do query planning without exhaustive searching. This is on by default. The geqo_threshold variable provides a more granular way to disable GEQO for certain classes of queries.

geqo_threshold (integer)

Use genetic query optimization to plan queries with at least this many FROM items involved. (Note that a FULL OUTER JOIN construct counts as only one FROM item.) The default is 12. For simpler queries it is usually best to use the deterministic, exhaustive planner, but for queries with many tables the deterministic planner takes too long.

gcqo_effort (integer)

Controls the trade off between planning time and query plan efficiency in GEQO. This variable must be an integer in the range from 1 to 10. The default value is five. Larger values increase the time spent doing query planning, but also increase the likelihood that an efficient query plan will be chosen.

gcqo_effort doesn’t actually do anything directly; it is only used to compute the default values for the other variables that influence GEQO behavior (described below). If you prefer, you can set the other parameters by hand instead.

gcqo_pool_size (integer)

Controls the pool size used by GEQO. The pool size is the number of individuals in the genetic population. It must be at least two, and useful values are typically 100 to 1000. If it is set to zero (the default setting) then a suitable default is chosen based on gcqo_effort and the number of tables in the query.

gcqo_generations (integer)

Controls the number of generations used by GEQO. Generations specifies the number of iterations of the algorithm. It must be at least one, and useful values are in the same range as the pool size. If it is set to zero (the default setting) then a suitable default is chosen based on gcqo_pool_size.

gcqo_selection_bias (floating point)

Controls the selection bias used by GEQO. The selection bias is the selective pressure within the population. Values can be from 1.50 to 2.00; the latter is the default.

18.6.4. Other Planner Options

default_statistics_target (integer)

Sets the default statistics target for table columns that have not had a column-specific target set via ALTER TABLE SET STATISTICS. Larger values increase the time needed to do ANALYZE, but might improve the quality of the planner’s estimates. The default is 10. For more information on the use of statistics by the PostgreSQL query planner, refer to Section 14.2.

constraint_exclusion (boolean)

Enables or disables the query planner’s use of table constraints to optimize queries. The default is off.
When this parameter is on, the planner compares query conditions with table CHECK constraints, and omits scanning tables for which the conditions contradict the constraints. For example:

CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;

With constraint exclusion enabled, this SELECT will not scan child1000 at all. This can improve performance when inheritance is used to build partitioned tables.

Currently, constraint_exclusion is disabled by default because the constraint checks are relatively expensive, and in many circumstances will yield no savings. It is recommended to turn this on only if you are actually using partitioned tables designed to take advantage of the feature.

Refer to Section 5.9 for more information on using constraint exclusion and partitioning.

from_collapse_limit (integer)

The planner will merge sub-queries into upper queries if the resulting FROM list would have no more than this many items. Smaller values reduce planning time but might yield inferior query plans. The default is eight. It is usually wise to keep this less than geqo_threshold. For more information see Section 14.3.

join_collapse_limit (integer)

The planner will rewrite explicit JOIN constructs (except FULL JOINS) into lists of FROM items whenever a list of no more than this many items would result. Smaller values reduce planning time but might yield inferior query plans.

By default, this variable is set the same as from_collapse_limit, which is appropriate for most uses. Setting it to 1 prevents any reordering of explicit JOINS. Thus, the explicit join order specified in the query will be the actual order in which the relations are joined. The query planner does not always choose the optimal join order; advanced users can elect to temporarily set this variable to 1, and then specify the join order they desire explicitly. For more information see Section 14.3.

18.7. Error Reporting and Logging

18.7.1. Where To Log

log_destination (string)

PostgreSQL supports several methods for logging server messages, including stderr, csvlog and syslog. On Windows, eventlog is also supported. Set this parameter to a list of desired log destinations separated by commas. The default is to log to stderr only. This parameter can only be set in the postgresql.conf file or on the server command line.

If csvlog is included in log_destination, log entries are output in “comma separated value” format, which is convenient for loading them into programs. See Section 18.7.4 for details. logging_collector must be enabled to generate CSV-format log output.

Note: On most Unix systems, you will need to alter the configuration of your system’s syslog daemon in order to make use of the syslog option for log_destination. PostgreSQL can log
to syslog facilities \texttt{LOCAL0} through \texttt{LOCAL7} (see syslog\_facility), but the default syslog configuration on most platforms will discard all such messages. You will need to add something like

\texttt{local0.* /var/log/postgresql}

to the syslog daemon’s configuration file to make it work.

\textbf{logging\_collector} (boolean)

This parameter allows messages sent to stderr, and CSV-format log output, to be captured and redirected into log files. This approach is often more useful than logging to syslog, since some types of messages might not appear in syslog output (a common example is dynamic-linker failure messages). This parameter can only be set at server start.

\textbf{log\_directory} (string)

When \textbf{logging\_collector} is enabled, this parameter determines the directory in which log files will be created. It can be specified as an absolute path, or relative to the cluster data directory. This parameter can only be set in the \texttt{postgresql.conf} file or on the server command line.

\textbf{log\_filename} (string)

When \textbf{logging\_collector} is enabled, this parameter sets the file names of the created log files. The value is treated as a strftime pattern, so %-escapes can be used to specify time-varying file names. (Note that if there are any time-zone-dependent %-escapes, the computation is done in the zone specified by \texttt{log\_timezone}.) If no %-escapes are present, PostgreSQL will append the epoch of the new log file’s creation time. For example, if \texttt{log\_filename} were \texttt{server\_log}, then the chosen file name would be \texttt{server\_log.1093827753} for a log starting at Sun Aug 29 19:02:33 2004 MST. This parameter can only be set in the \texttt{postgresql.conf} file or on the server command line.

If CSV-format output is enabled in \textbf{log\_destination}, .csv will be appended to the times-tamped log file name to create the file name for CSV-format output. (If \texttt{log\_filename} ends in .log, the suffix is replaced instead.) In the case of the example above, the CSV file name will be \texttt{server\_log.1093827753.csv}.

\textbf{log\_rotation\_age} (integer)

When \textbf{logging\_collector} is enabled, this parameter determines the maximum lifetime of an individual log file. After this many minutes have elapsed, a new log file will be created. Set to zero to disable time-based creation of new log files. This parameter can only be set in the \texttt{postgresql.conf} file or on the server command line.

\textbf{log\_rotation\_size} (integer)

When \textbf{logging\_collector} is enabled, this parameter determines the maximum size of an individual log file. After this many kilobytes have been emitted into a log file, a new log file will be created. Set to zero to disable size-based creation of new log files. This parameter can only be set in the \texttt{postgresql.conf} file or on the server command line.

\textbf{log\_truncate\_on\_rotation} (boolean)

When \textbf{logging\_collector} is enabled, this parameter will cause PostgreSQL to truncate (overwrite), rather than append to, any existing log file of the same name. However, truncation will occur only when a new file is being opened due to time-based rotation, not during server startup or size-based rotation. When off, pre-existing files will be appended to in all cases. For example, using this setting in combination with a \texttt{log\_filename} like \texttt{postgresql-\%H.log} would result in generating twenty-four hourly log files and then cyclically overwriting them. This parameter can only be set in the \texttt{postgresql.conf} file or on the server command line.
Example: To keep 7 days of logs, one log file per day named `server_log.Mon, server_log.Tue`, etc, and automatically overwrite last week’s log with this week’s log, set `log_filename` to `server_log.%a`, `log_truncate_on_rotation` to `on`, and `log_rotation_age` to `1440`.

Example: To keep 24 hours of logs, one log file per hour, but also rotate sooner if the log file size exceeds 1GB, set `log_filename` to `server_log.%H%M`, `log_truncate_on_rotation` to `on`, `log_rotation_age` to `60`, and `log_rotation_size` to `1000000`. Including `%M` in `log_filename` allows any size-driven rotations that might occur to select a file name different from the hour’s initial file name.

`syslog_facility` (string)

When logging to syslog is enabled, this parameter determines the syslog “facility” to be used. You can choose from `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7`; the default is `LOCAL0`. See also the documentation of your system’s syslog daemon. This parameter can only be set in the `postgresql.conf` file or on the server command line. This parameter is unavailable unless the server is compiled with support for syslog.

`syslog_ident` (string)

When logging to syslog is enabled, this parameter determines the program name used to identify PostgreSQL messages in syslog logs. The default is `postgres`. This parameter can only be set in the `postgresql.conf` file or on the server command line. This parameter is unavailable unless the server is compiled with support for syslog.

### 18.7.2. When To Log

`client_min_messages` (string)

Controls which message levels are sent to the client. Valid values are `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `LOG`, `NOTICE`, `WARNING`, `ERROR`, `FATAL`, and `PANIC`. Each level includes all the levels that follow it. The later the level, the fewer messages are sent. The default is `NOTICE`. Note that `LOG` has a different rank here than in `log_min_messages`.

`log_min_messages` (string)

Controls which message levels are written to the server log. Valid values are `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL`, and `PANIC`. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log. The default is `NOTICE`. Note that `LOG` has a different rank here than in `client_min_messages`. Only superusers can change this setting.

`log_error_verbosity` (string)

Controls the amount of detail written in the server log for each message that is logged. Valid values are `TERSE`, `DEFAULT`, and `VERBOSE`, each adding more fields to displayed messages. Only superusers can change this setting.

`log_min_error_statement` (string)

Controls whether or not the SQL statement that causes an error condition will be recorded in the server log. The current SQL statement is included in the log entry for any message of the specified severity or higher. Valid values are `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL`, and `PANIC`. The default is `ERROR`, which means statements causing errors, log messages, fatal errors, or panics will be logged. To effectively turn
off logging of failing statements, set this parameter to PANIC. Only superusers can change this setting.

log_min_duration_statement (integer)

Causes the duration of each completed statement to be logged if the statement ran for at least the specified number of milliseconds. Setting this to zero prints all statement durations. Minus-one (the default) disables logging statement durations. For example, if you set it to 250ms then all SQL statements that run 250ms or longer will be logged. Enabling this parameter can be helpful in tracking down unoptimized queries in your applications. Only superusers can change this setting.

For clients using extended query protocol, durations of the Parse, Bind, and Execute steps are logged independently.

**Note:** When using this option together with log_statement, the text of statements that are logged because of log_statement will not be repeated in the duration log message. If you are not using syslog, it is recommended that you log the PID or session ID using log_line_prefix so that you can link the statement message to the later duration message using the process ID or session ID.

silent_mode (boolean)

Runs the server silently. If this parameter is set, the server will automatically run in background and any controlling terminals are disassociated. The server’s standard output and standard error are redirected to /dev/null, so any messages sent to them will be lost. Unless syslog logging is selected or logging_collector is enabled, using this parameter is discouraged because it makes it impossible to see error messages. This parameter can only be set at server start.

Table 18-1 explains the message severity levels used by PostgreSQL. If logging output is sent to syslog or Windows’ eventlog, the severity levels are translated as shown in the table.

**Table 18-1. Message severity levels**

<table>
<thead>
<tr>
<th>Severity</th>
<th>Usage</th>
<th>syslog</th>
<th>eventlog</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG1..DEBUG5</td>
<td>Provides successively-more-detailed information for use by developers.</td>
<td>DEBUG</td>
<td>INFORMATION</td>
</tr>
<tr>
<td>INFO</td>
<td>Provides information implicitly requested by the user, e.g., output from VACUUMVERBOSE.</td>
<td>INFO</td>
<td>INFORMATION</td>
</tr>
<tr>
<td>NOTICE</td>
<td>Provides information that might be helpful to users, e.g., notice of truncation of long identifiers.</td>
<td>NOTICE</td>
<td>INFORMATION</td>
</tr>
<tr>
<td>WARNING</td>
<td>Provides warnings of likely problems, e.g., COMMIT outside a transaction block.</td>
<td>NOTICE</td>
<td>WARNING</td>
</tr>
</tbody>
</table>
### Chapter 18. Server Configuration

<table>
<thead>
<tr>
<th>Severity</th>
<th>Usage</th>
<th>syslog</th>
<th>eventlog</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>Reports an error that caused the current command to abort.</td>
<td>WARNING</td>
<td>ERROR</td>
</tr>
<tr>
<td>LOG</td>
<td>Reports information of interest to administrators, e.g., checkpoint activity.</td>
<td>INFO</td>
<td>INFORMATION</td>
</tr>
<tr>
<td>FATAL</td>
<td>Reports an error that caused the current session to abort.</td>
<td>ERR</td>
<td>ERROR</td>
</tr>
<tr>
<td>PANIC</td>
<td>Reports an error that caused all database sessions to abort.</td>
<td>CRIT</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

### 18.7.3. What To Log

**debug_print_parse**(boolean)

**debug_print_rewritten**(boolean)

**debug_print_plan**(boolean)

**debug_pretty_print**(boolean)

These parameters enable various debugging output to be emitted. For each executed query, they print the resulting parse tree, the query rewriter output, or the execution plan. **debug_pretty_print** indents these displays to produce a more readable but much longer output format. **client_min_messages** or **log_min_messages** must be **DEBUG1** or lower to actually send this output to the client or the server log, respectively. These parameters are off by default.

**log_checkpoints**(boolean)

Causes checkpoints to be logged in the server log. Some statistics about each checkpoint are included in the log messages, including the number of buffers written and the time spent writing them. This parameter can only be set in the **postgresql.conf** file or on the server command line. The default is off.

**log_connections**(boolean)

Causes each attempted connection to the server to be logged, as well as successful completion of client authentication. This parameter can only be set in the **postgresql.conf** file or on the server command line. The default is off.

**Note:** Some client programs, like psql, attempt to connect twice while determining if a password is required, so duplicate “connection received” messages do not necessarily indicate a problem.

**log_disconnections**(boolean)

This outputs a line in the server log similar to **log_connections** but at session termination, and includes the duration of the session. This is off by default. This parameter can only be set in the **postgresql.conf** file or on the server command line.
log_duration (boolean)

Causes the duration of every completed statement to be logged. The default is off. Only superusers can change this setting.

For clients using extended query protocol, durations of the Parse, Bind, and Execute steps are logged independently.

Note: The difference between setting this option and setting log_min_duration_statement to zero is that exceeding log_min_duration_statement forces the text of the query to be logged, but this option doesn’t. Thus, if log_duration is on and log_min_duration_statement has a positive value, all durations are logged but the query text is included only for statements exceeding the threshold. This behavior can be useful for gathering statistics in high-load installations.

log_hostname (boolean)

By default, connection log messages only show the IP address of the connecting host. Turning on this parameter causes logging of the host name as well. Note that depending on your host name resolution setup this might impose a non-negligible performance penalty. This parameter can only be set in the postgresql.conf file or on the server command line.

log_line_prefix (string)

This is a printf-style string that is output at the beginning of each log line. % characters begin “escape sequences” that are replaced with status information as outlined below. Unrecognized escapes are ignored. Other characters are copied straight to the log line. Some escapes are only recognized by session processes, and do not apply to background processes such as the main server process. This parameter can only be set in the postgresql.conf file or on the server command line. The default is an empty string.

<table>
<thead>
<tr>
<th>Escape</th>
<th>Effect</th>
<th>Session only</th>
</tr>
</thead>
<tbody>
<tr>
<td>%u</td>
<td>User name</td>
<td>yes</td>
</tr>
<tr>
<td>%d</td>
<td>Database name</td>
<td>yes</td>
</tr>
<tr>
<td>%r</td>
<td>Remote host name or IP address, and remote port</td>
<td>yes</td>
</tr>
<tr>
<td>%h</td>
<td>Remote host name or IP address</td>
<td>yes</td>
</tr>
<tr>
<td>%p</td>
<td>Process ID</td>
<td>no</td>
</tr>
<tr>
<td>%t</td>
<td>Time stamp without milliseconds</td>
<td>no</td>
</tr>
<tr>
<td>%m</td>
<td>Time stamp with milliseconds</td>
<td>no</td>
</tr>
<tr>
<td>%i</td>
<td>Command tag: type of session’s current command</td>
<td>yes</td>
</tr>
<tr>
<td>%c</td>
<td>Session ID: see below</td>
<td>no</td>
</tr>
<tr>
<td>%l</td>
<td>Number of the log line for each session or process, starting at 1</td>
<td>no</td>
</tr>
<tr>
<td>%s</td>
<td>Process start time stamp</td>
<td>no</td>
</tr>
<tr>
<td>%v</td>
<td>Virtual transaction ID (backendID/localXID)</td>
<td>no</td>
</tr>
</tbody>
</table>
Chapter 18. Server Configuration

### Escape

<table>
<thead>
<tr>
<th>Escape</th>
<th>Effect</th>
<th>Session only</th>
</tr>
</thead>
<tbody>
<tr>
<td>%x</td>
<td>Transaction ID (0 if none is assigned)</td>
<td>no</td>
</tr>
<tr>
<td>%q</td>
<td>Produces no output, but tells non-session processes to stop at this point in the string; ignored by session processes</td>
<td>no</td>
</tr>
<tr>
<td>%%</td>
<td>Literal %</td>
<td>no</td>
</tr>
</tbody>
</table>

The `%c` escape prints a quasi-unique session identifier, consisting of two 4-byte hexadecimal numbers (without leading zeros) separated by a dot. The numbers are the process start time and the process ID, so `%c` can also be used as a space saving way of printing those items.

**Tip:** If you set a nonempty value for `log_line_prefix`, you should usually make its last character be a space, to provide visual separation from the rest of the log line. A punctuation character could be used too.

**Tip:** Syslog produces its own time stamp and process ID information, so you probably do not want to use those escapes if you are logging to syslog.

---

**log_lock_waits (boolean)**

Controls whether a log message is produced when a session waits longer than `deadlock_timeout` to acquire a lock. This is useful in determining if lock waits are causing poor performance. The default is `off`.

**log_statement (string)**

Controls which SQL statements are logged. Valid values are `none`, `ddl`, `mod`, and `all`. `ddl` logs all data definition statements, such as `CREATE`, `ALTER`, and `DROP` statements. `mod` logs all `ddl` statements, plus data-modifying statements such as `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, and `COPY FROM`, `PREPARE`, `EXECUTE`, and `EXPLAIN ANALYZE` statements are also logged if their contained command is of an appropriate type. For clients using extended query protocol, logging occurs when an `Execute` message is received, and values of the Bind parameters are included (with any embedded single-quote marks doubled).

The default is `none`. Only superusers can change this setting.

**Note:** Statements that contain simple syntax errors are not logged even by the `log_statement = all` setting, because the log message is emitted only after basic parsing has been done to determine the statement type. In the case of extended query protocol, this setting likewise does not log statements that fail before the `Execute` phase (i.e., during parse analysis or planning). Set `log_min_error_statement` to `ERROR` (or lower) to log such statements.

**log_temp_files (integer)**

Controls whether temporary files are logged when deleted. Temporary files can be created for sorts, hashes, and temporary query results. A value of zero logs all temporary files, and positive values log only files whose size is equal or greater than the specified number of kilobytes. The default is `-1`, which disables this logging.
log_timezone (string)

Sets the time zone used for timestamps written in the log. Unlike timezone, this value is cluster-wide, so that all sessions will report timestamps consistently. The default is unknown, which means to use whatever the system environment specifies as the time zone. See Section 8.5.3 for more information. This parameter can only be set in the postgresql.conf file or on the server command line.

18.7.4. Using CSV-Format Log Output

Including csvlog in the log_destination list provides a convenient way to import log files into a database table. This option emits log lines in comma-separated-value format, with these columns: timestamp with milliseconds, user name, database name, process ID, host:port number, session ID, per-session or -process line number, command tag, session start time, virtual transaction ID, regular transaction id, error severity, SQL state code, error message, error message detail, hint, internal query that led to the error (if any), character count of the error position thereof, error context, user query that led to the error (if any and enabled by log_min_error_statement), character count of the error position thereof, location of the error in the PostgreSQL source code (if log_error_verbosity is set to verbose). Here is a sample table definition for storing CSV-format log output:

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  PRIMARY KEY (session_id, session_line_num)
);
```

To import a log file into this table, use the COPY FROM command:

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

There are a few things you need to do to simplify importing CSV log files easily and automatically:
1. Set `log_filename` and `log_rotation_age` to provide a consistent, predictable naming scheme for your log files. This lets you predict what the file name will be and know when an individual log file is complete and therefore ready to be imported.

2. Set `log_rotation_size` to 0 to disable size-based log rotation, as it makes the log file name difficult to predict.

3. Set `log_truncate_on_rotation` to `on` so that old log data isn’t mixed with the new in the same file.

4. The table definition above includes a primary key specification. This is useful to protect against accidentally importing the same information twice. The `COPY` command commits all of the data it imports at one time, so any error will cause the entire import to fail. If you import a partial log file and later import the file again when it is complete, the primary key violation will cause the import to fail. Wait until the log is complete and closed before importing. This procedure will also protect against accidentally importing a partial line that hasn’t been completely written, which would also cause `COPY` to fail.

### 18.8. Run-Time Statistics

#### 18.8.1. Query and Index Statistics Collector

These parameters control server-wide statistics collection features. When statistics collection is enabled, the data that is produced can be accessed via the `pg_stat` and `pg_statio` family of system views. Refer to Chapter 26 for more information.

- `track_activities` (boolean)
  Enables the collection of information on the currently executing command of each session, along with the time at which that command began execution. This parameter is on by default. Note that even when enabled, this information is not visible to all users, only to superusers and the user owning the session being reported on; so it should not represent a security risk. Only superusers can change this setting.

- `track_counts` (boolean)
  Enables collection of statistics on database activity. This parameter is on by default, because the autovacuum daemon needs the collected information. Only superusers can change this setting.

- `update_process_title` (boolean)
  Enables updating of the process title every time a new SQL command is received by the server. The process title is typically viewed by the `ps` command, or in Windows by using the Process Explorer. Only superusers can change this setting.
Chapter 18. Server Configuration

18.8.2. Statistics Monitoring

log_statement_stats (boolean)
log_parser_stats (boolean)
log_planner_stats (boolean)
log_executor_stats (boolean)

For each query, write performance statistics of the respective module to the server log. This is a crude profiling instrument. log_statement_stats reports total statement statistics, while the others report per-module statistics. log_statement_stats cannot be enabled together with any of the per-module options. All of these options are disabled by default. Only superusers can change these settings.

18.9. Automatic Vacuuming

These settings control the behavior of the autovacuum feature. Refer to Section 23.1.4 for more information.

autovacuum (boolean)

Controls whether the server should run the autovacuum launcher daemon. This is on by default; however, track_counts must also be turned on for autovacuum to work. This parameter can only be set in the postgresql.conf file or on the server command line.

Note that even when this parameter is disabled, the system will launch autovacuum processes if necessary to prevent transaction ID wraparound. See Section 23.1.3 for more information.

log_autovacuum_min_duration (integer)

Causes each action executed by autovacuum to be logged if it ran for at least the specified number of milliseconds. Setting this to zero logs all autovacuum actions. Minus-one (the default) disables logging autovacuum actions. For example, if you set this to 250ms then all automatic vacuums and analyzes that run 250ms or longer will be logged. Enabling this parameter can be helpful in tracking autovacuum activity. This setting can only be set in the postgresql.conf file or on the server command line.

autovacuum_max_workers (integer)

Specifies the maximum number of autovacuum processes (other than the autovacuum launcher) which may be running at any one time. The default is three. This parameter can only be set in the postgresql.conf file or on the server command line.

autovacuum_naptime (integer)

Specifies the minimum delay between autovacuum runs on any given database. In each round the daemon examines the database and issues VACUUM and ANALYZE commands as needed for tables in that database. The delay is measured in seconds, and the default is one minute (1m). This parameter can only be set in the postgresql.conf file or on the server command line.

autovacuum_vacuum_threshold (integer)

Specifies the minimum number of updated or deleted tuples needed to trigger a VACUUM in any one table. The default is 50 tuples. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.
autovacuum_analyze_threshold (integer)

Specifies the minimum number of inserted, updated or deleted tuples needed to trigger an ANALYZE in any one table. The default is 50 tuples. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

autovacuum_vacuum_scale_factor (floating point)

Specifies a fraction of the table size to add to autovacuum_vacuum_threshold when deciding whether to trigger a VACUUM. The default is 0.2 (20% of table size). This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

autovacuum_analyze_scale_factor (floating point)

Specifies a fraction of the table size to add to autovacuum_analyze_threshold when deciding whether to trigger an ANALYZE. The default is 0.1 (10% of table size). This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

autovacuum_freeze_max_age (integer)

Specifies the maximum age (in transactions) that a table’s pg_class.relro frozenxid field can attain before a VACUUM operation is forced to prevent transaction ID wraparound within the table. Note that the system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled. The default is 200 million transactions. This parameter can only be set at server start, but the setting can be reduced for individual tables by entries in pg_autovacuum. For more information see Section 23.1.3.

autovacuum_vacuum_cost_delay (integer)

Specifies the cost delay value that will be used in automatic VACUUM operations. If -1 is specified, the regular vacuum_cost_delay value will be used. The default value is 20 milliseconds. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

autovacuum_vacuum_cost_limit (integer)

Specifies the cost limit value that will be used in automatic VACUUM operations. If -1 is specified (which is the default), the regular vacuum_cost_limit value will be used. Note that the value is distributed proportionally among the running autovacuum workers, if there is more than one, so that the sum of the limits of each worker never exceeds the limit on this variable. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by entries in pg_autovacuum.

18.10. Client Connection Defaults

18.10.1. Statement Behavior

search_path (string)

This variable specifies the order in which schemas are searched when an object (table, data type, function, etc.) is referenced by a simple name with no schema component. When there are objects of identical names in different schemas, the one found first in the search path is used. An
object that is not in any of the schemas in the search path can only be referenced by specifying
tits containing schema with a qualified (dotted) name.

The value for search_path has to be a comma-separated list of schema names. If one of the list
items is the special value $user, then the schema having the name returned by SESSION_USER
is substituted, if there is such a schema. (If not, $user is ignored.)

The system catalog schema, pg_catalog, is always searched, whether it is mentioned in the
path or not. If it is mentioned in the path then it will be searched in the specified order. If
pg_catalog is not in the path then it will be searched before searching any of the path items.

Likewise, the current session’s temporary-table schema, pg_temp_nnn, is always searched if it
exists. It can be explicitly listed in the path by using the alias pg_temp. If it is not listed in the
path then it is searched first (before even pg_catalog). However, the temporary schema is only
searched for relation (table, view, sequence, etc) and data type names. It will never be searched
for function or operator names.

When objects are created without specifying a particular target schema, they will be placed in
the first schema listed in the search path. An error is reported if the search path is empty.

The default value for this parameter is ‘"$user", public’ (where the second part will be
ignored if there is no schema named public). This supports shared use of a database (where
no users have private schemas, and all share use of public), private per-user schemas, and
combinations of these. Other effects can be obtained by altering the default search path setting,
either globally or per-user.

The current effective value of the search path can be examined via the SQL function
current_schemas(). This is not quite the same as examining the value of search_path,
since current_schemas() shows how the requests appearing in search_path were
resolved.

For more information on schema handling, see Section 5.7.

default_tablespace (string)

This variable specifies the default tablespace in which to create objects (tables and indexes) when
a CREATE command does not explicitly specify a tablespace.

The value is either the name of a tablespace, or an empty string to specify using the default ta-
blespace of the current database. If the value does not match the name of any existing tablespace,
PostgreSQL will automatically use the default tablespace of the current database. If a nondefault
tablespace is specified, the user must have CREATE privilege for it, or creation attempts will fail.

This variable is not used for temporary tables; for them, temp_tablespaces is consulted instead.

For more information on tablespaces, see Section 20.6.

temp_tablespaces (string)

This variable specifies tablespace(s) in which to create temporary objects (temp tables and in-
dexes on temp tables) when a CREATE command does not explicitly specify a tablespace. Temp-
orary files for purposes such as sorting large data sets are also created in these tablespace(s).

The value is a list of names of tablespaces. When there is more than one name in the list, Post-
greSQL chooses a random member of the list each time a temporary object is to be created;
except that within a transaction, successively created temporary objects are placed in successive
tablespaces from the list. If the selected element of the list is an empty string, PostgreSQL will
automatically use the default tablespace of the current database instead.

When temp_tablespaces is set interactively, specifying a nonexistent tablespace is an er-
ror, as is specifying a tablespace for which the user does not have CREATE privilege. However,
when using a previously set value, nonexistent tablespaces are ignored, as are tablespaces for which the user lacks CREATE privilege. In particular, this rule applies when using a value set in postgresql.conf.

The default value is an empty string, which results in all temporary objects being created in the default tablespace of the current database.

See also default_tablespace.

cHECK_FUNCTION_BODIES (boolean)

This parameter is normally on. When set to off, it disables validation of the function body string during CREATE FUNCTION. Disabling validation is occasionally useful to avoid problems such as forward references when restoring function definitions from a dump.

default_transaction_isolation (string)

Each SQL transaction has an isolation level, which can be either “read uncommitted”, “read committed”, “repeatable read”, or “serializable”. This parameter controls the default isolation level of each new transaction. The default is “read committed”.

Consult Chapter 13 and SET TRANSACTION for more information.

default_transaction_read_only (boolean)

A read-only SQL transaction cannot alter non-temporary tables. This parameter controls the default read-only status of each new transaction. The default is off (read/write).

Consult SET TRANSACTION for more information.

SESSION_REPLICATION_ROLE (string)

Controls firing of replication-related triggers and rules for the current session. Setting this variable requires superuser privilege and results in discarding any previously cached query plans. Possible values are origin (the default), replica and local. See ALTER TABLE for more information.

STATEMENT_TIMEOUT (integer)

Abort any statement that takes over the specified number of milliseconds, starting from the time the command arrives at the server from the client. If log_min_error_statement is set to ERROR or lower, the statement that timed out will also be logged. A value of zero (the default) turns off the limitation.

Setting statement_timeout in postgresql.conf is not recommended because it affects all sessions.

VACUUM_FREEZE_MIN_AGE (integer)

Specifies the cutoff age (in transactions) that VACUUM should use to decide whether to replace transaction IDs with FrozenXID while scanning a table. The default is 100 million transactions. Although users can set this value anywhere from zero to one billion, VACUUM will silently limit the effective value to half the value of autovacuum_freeze_max_age, so that there is not an unreasonably short time between forced autovacuums. For more information see Section 23.1.3.

XML_BINARY (string)

Sets how binary values are to be encoded in XML. This applies for example when bytea values are converted to XML by the functions xmlelement or xmlforest. Possible values are base64 and hex, which are both defined in the XML Schema standard. The default is base64. For further information about XML-related functions, see Section 9.14.
The actual choice here is mostly a matter of taste, constrained only by possible restrictions in client applications. Both methods support all possible values, although the hex encoding will be somewhat larger than the base64 encoding.

**xmlOption** (string)

Sets whether `DOCUMENT` or `CONTENT` is implicit when converting between XML and character string values. See Section 8.13 for a description of this. Valid values are `DOCUMENT` and `CONTENT`. The default is `CONTENT`.

According to the SQL standard, the command to set this option is

```sql
SET XML OPTION { DOCUMENT | CONTENT };
```

This syntax is also available in PostgreSQL.

### 18.10.2. Locale and Formatting

**DateStyle** (string)

Sets the display format for date and time values, as well as the rules for interpreting ambiguous date input values. For historical reasons, this variable contains two independent components: the output format specification (`ISO`, `Postgres`, `SQL`, or `German`) and the input/output specification for year/month/day ordering (`DMY`, `MDY`, or `YMD`). These can be set separately or together. The keywords `Euro` and `European` are synonyms for `DMY`; the keywords `US`, `NonEuro`, and `NonEuropean` are synonyms for `MDY`. See Section 8.5 for more information. The built-in default is `ISO`, `MDY`, but `initdb` will initialize the configuration file with a setting that corresponds to the behavior of the chosen `lc_time` locale.

**timezone** (string)

Sets the time zone for displaying and interpreting time stamps. The default is `unknown`, which means to use whatever the system environment specifies as the time zone. See Section 8.5.3 for more information.

**timezone_abbreviations** (string)

Sets the collection of time zone abbreviations that will be accepted by the server for datetime input. The default is ‘Default’, which is a collection that works in most of the world; there are also ‘Australia’ and ‘India’, and other collections can be defined for a particular installation. See Appendix B for more information.

**extra_float_digits** (integer)

This parameter adjusts the number of digits displayed for floating-point values, including `float4`, `float8`, and geometric data types. The parameter value is added to the standard number of digits (`FLT_DIG` or `DBL_DIG` as appropriate). The value can be set as high as 2, to include partially-significant digits; this is especially useful for dumping float data that needs to be restored exactly. Or it can be set negative to suppress unwanted digits.

**client_encoding** (string)

Sets the client-side encoding (character set). The default is to use the database encoding.

**lc_messages** (string)

Sets the language in which messages are displayed. Acceptable values are system-dependent; see Section 22.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.
On some systems, this locale category does not exist. Setting this variable will still work, but there will be no effect. Also, there is a chance that no translated messages for the desired language exist. In that case you will continue to see the English messages.

Only superusers can change this setting, because it affects the messages sent to the server log as well as to the client.

**lc_monetary(string)**

Sets the locale to use for formatting monetary amounts, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 22.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

**lc_numeric(string)**

Sets the locale to use for formatting numbers, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 22.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

**lc_time(string)**

Sets the locale to use for formatting date and time values. (Currently, this setting does nothing, but it might in the future.) Acceptable values are system-dependent; see Section 22.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

**default_text_search_config(string)**

Selects the text search configuration that is used by those variants of the text search functions that do not have an explicit argument specifying the configuration. See Chapter 12 for further information. The built-in default is `pg_catalog.simple`, but `initdb` will initialize the configuration file with a setting that corresponds to the chosen `lc_ctype` locale, if a configuration matching that locale can be identified.

### 18.10.3. Other Defaults

**explain_pretty_print(boolean)**

Determines whether `EXPLAIN VERBOSE` uses the indented or non-indented format for displaying detailed query-tree dumps. The default is `on`.

**dynamic_library_path(string)**

If a dynamically loadable module needs to be opened and the file name specified in the `CREATE FUNCTION` or `LOAD` command does not have a directory component (i.e. the name does not contain a slash), the system will search this path for the required file.

The value for `dynamic_library_path` has to be a list of absolute directory paths separated by colons (or semi-colons on Windows). If a list element starts with the special string `$libdir`, the compiled-in PostgreSQL package library directory is substituted for `$libdir`. This is where the modules provided by the standard PostgreSQL distribution are installed. (Use `pg_config --pkglibdir` to find out the name of this directory.) For example:

- `dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'`
- or, in a Windows environment:
  `dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'`
The default value for this parameter is `$libdir`. If the value is set to an empty string, the automatic path search is turned off.

This parameter can be changed at run time by superusers, but a setting done that way will only persist until the end of the client connection, so this method should be reserved for development purposes. The recommended way to set this parameter is in the `postgresql.conf` configuration file.

`gin_fuzzy_search_limit (integer)`
Soft upper limit of the size of the set returned by GIN index. For more information see Section 52.4.

`local_preload_libraries (string)`
This variable specifies one or more shared libraries that are to be preloaded at connection start. If more than one library is to be loaded, separate their names with commas. This parameter cannot be changed after the start of a particular session.

Because this is not a superuser-only option, the libraries that can be loaded are restricted to those appearing in the `plugins` subdirectory of the installation’s standard library directory. (It is the database administrator’s responsibility to ensure that only “safe” libraries are installed there.) Entries in `local_preload_libraries` can specify this directory explicitly, for example `$libdir/plugins/mylib`, or just specify the library name — `mylib` would have the same effect as `$libdir/plugins/mylib`.

There is no performance advantage to loading a library at session start rather than when it is first used. Rather, the intent of this feature is to allow debugging or performance-measurement libraries to be loaded into specific sessions without an explicit `LOAD` command being given. For example, debugging could be enabled for all sessions under a given user name by setting this parameter with `ALTER USER SET`.

If a specified library is not found, the connection attempt will fail.

Every PostgreSQL-supported library has a “magic block” that is checked to guarantee compatibility. For this reason, non-PostgreSQL libraries cannot be loaded in this way.

### 18.11. Lock Management

`deadlock_timeout (integer)`
This is the amount of time, in milliseconds, to wait on a lock before checking to see if there is a deadlock condition. The check for deadlock is relatively slow, so the server doesn’t run it every time it waits for a lock. We optimistically assume that deadlocks are not common in production applications and just wait on the lock for a while before starting the check for a deadlock. Increasing this value reduces the amount of time wasted in needless deadlock checks, but slows down reporting of real deadlock errors. The default is one second (`1s`), which is probably about the smallest value you would want in practice. On a heavily loaded server you might want to raise it. Ideally the setting should exceed your typical transaction time, so as to improve the odds that a lock will be released before the waiter decides to check for deadlock.

When `log_lock_waits` is set, this parameter also determines the length of time to wait before a log message is issued about the lock wait. If you are trying to investigate locking delays you might want to set a shorter than normal `deadlock_timeout`. 
max_locks_per_transaction (integer)

The shared lock table is created to track locks on \( \text{max_locks_per_transaction} \) objects (e.g., tables); hence, no more than this many distinct objects can be locked at any one time. This parameter controls the average number of object locks allocated for each transaction; individual transactions can lock more objects as long as the locks of all transactions fit in the lock table. This is not the number of rows that can be locked; that value is unlimited. The default, 64, has historically proven sufficient, but you might need to raise this value if you have clients that touch many different tables in a single transaction. This parameter can only be set at server start.

Increasing this parameter might cause PostgreSQL to request more System V shared memory than your operating system’s default configuration allows. See Section 17.4.1 for information on how to adjust those parameters, if necessary.

18.12. Version and Platform Compatibility

18.12.1. Previous PostgreSQL Versions

add_missing_from (boolean)

When on, tables that are referenced by a query will be automatically added to the FROM clause if not already present. This behavior does not comply with the SQL standard and many people dislike it because it can mask mistakes (such as referencing a table where you should have referenced its alias). The default is off. This variable can be enabled for compatibility with releases of PostgreSQL prior to 8.1, where this behavior was allowed by default.

Note that even when this variable is enabled, a warning message will be emitted for each implicit FROM entry referenced by a query. Users are encouraged to update their applications to not rely on this behavior, by adding all tables referenced by a query to the query’s FROM clause (or its USING clause in the case of DELETE).

array_nulls (boolean)

This controls whether the array input parser recognizes unquoted NULL as specifying a null array element. By default, this is on, allowing array values containing null values to be entered. However, PostgreSQL versions before 8.2 did not support null values in arrays, and therefore would treat NULL as specifying a normal array element with the string value “NULL”. For backwards compatibility with applications that require the old behavior, this variable can be turned off.

Note that it is possible to create array values containing null values even when this variable is off.

backslash_quote (string)

This controls whether a quote mark can be represented by \' in a string literal. The preferred, SQL-standard way to represent a quote mark is by doubling it (\") but PostgreSQL has historically also accepted \'. However, use of \' creates security risks because in some client character set encodings, there are multibyte characters in which the last byte is numerically equivalent to ASCII \'. If client-side code does escaping incorrectly then a SQL-injection attack is possible. This risk can be prevented by making the server reject queries in which a quote mark appears to be escaped by a backslash. The allowed values of backslash_quote are on (allow \' always), off (reject always), and safe_encoding (allow only if client encoding does not allow ASCII \ within a multibyte character). safe_encoding is the default setting.
Note that in a standard-conforming string literal, \ just means \ anyway. This parameter affects the handling of non-standard-conforming literals, including escape string syntax (E’...’).

**default_with_oids (boolean)**

This controls whether CREATE TABLE and CREATE TABLE AS include an OID column in newly-created tables, if neither WITH OIDS nor WITHOUT OIDS is specified. It also determines whether OIDs will be included in tables created by SELECT INTO. In PostgreSQL 8.1 default_with_oids is off by default; in prior versions of PostgreSQL, it was on by default.

The use of OIDs in user tables is considered deprecated, so most installations should leave this variable disabled. Applications that require OIDs for a particular table should specify WITH OIDS when creating the table. This variable can be enabled for compatibility with old applications that do not follow this behavior.

**escape_string_warning (boolean)**

When on, a warning is issued if a backslash (\) appears in an ordinary string literal (‘...’ syntax) and standard_conforming_strings is off. The default is on.

Applications that wish to use backslash as escape should be modified to use escape string syntax (E’...’), because the default behavior of ordinary strings will change in a future release for SQL compatibility. This variable can be enabled to help detect applications that will break.

**regex_flavor (string)**

The regular expression “flavor” can be set to advanced, extended, or basic. The default is advanced. The extended setting might be useful for exact backwards compatibility with pre-7.4 releases of PostgreSQL. See Section 9.7.3.1 for details.

**sql_inheritance (boolean)**

This setting controls whether undecorated table references are considered to include inheritance child tables. The default is on, which means child tables are included (thus, a * suffix is assumed by default). If turned off, child tables are not included (thus, an ONLY prefix is assumed). The SQL standard requires child tables to be included, so the off setting is not spec-compliant, but it is provided for compatibility with PostgreSQL releases prior to 7.1. See Section 5.8 for more information.

Turning sql_inheritance off is deprecated, because behavior has been found to be error-prone as well as contrary to SQL standard. Discussions of inheritance behavior elsewhere in this manual generally assume that it is on.

**standard_conforming_strings (boolean)**

This controls whether ordinary string literals (‘...’) treat backslashes literally, as specified in the SQL standard. The default is currently off, causing PostgreSQL to have its historical behavior of treating backslashes as escape characters. The default will change to on in a future release to improve compatibility with the standard. Applications can check this parameter to determine how string literals will be processed. The presence of this parameter can also be taken as an indication that the escape string syntax (E’...’) is supported. Escape string syntax should be used if an application desires backslashes to be treated as escape characters.

**synchronize_seqscans (boolean)**

This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload. When this is enabled, a scan might start in the middle of the table and then “wrap around” the end to cover all rows, so as to synchronize with the activity of scans already in progress. This can result in unpredictable changes in the row ordering returned by queries that have no ORDER BY clause.
Chapter 18. Server Configuration

Setting this parameter to off ensures the pre-8.3 behavior in which a sequential scan always starts from the beginning of the table. The default is on.

18.12.2. Platform and Client Compatibility

transform_null_equals (boolean)

When on, expressions of the form expr = NULL (or NULL = expr) are treated as expr IS NULL, that is, they return true if expr evaluates to the null value, and false otherwise. The correct SQL-spec-compliant behavior of expr = NULL is to always return null (unknown). Therefore this parameter defaults to off.

However, filtered forms in Microsoft Access generate queries that appear to use expr = NULL to test for null values, so if you use that interface to access the database you might want to turn this option on. Since expressions of the form expr = NULL always return the null value (using the correct interpretation) they are not very useful and do not appear often in normal applications, so this option does little harm in practice. But new users are frequently confused about the semantics of expressions involving null values, so this option is not on by default.

Note that this option only affects the exact form = NULL, not other comparison operators or other expressions that are computationally equivalent to some expression involving the equals operator (such as IN). Thus, this option is not a general fix for bad programming.

Refer to Section 9.2 for related information.

18.13. Preset Options

The following “parameters” are read-only, and are determined when PostgreSQL is compiled or when it is installed. As such, they have been excluded from the sample postgresql.conf file. These options report various aspects of PostgreSQL behavior that might be of interest to certain applications, particularly administrative front-ends.

block_size (integer)

Reports the size of a disk block. It is determined by the value of BLCKSZ when building the server. The default value is 8192 bytes. The meaning of some configuration variables (such as shared_buffers) is influenced by block_size. See Section 18.4 for information.

integer_datetimes (boolean)

Reports whether PostgreSQL was built with support for 64-bit-integer dates and times. It is set by configuring with --enable-integer-datetimes when building PostgreSQL. The default value is off.

lc_collate (string)

Reports the locale in which sorting of textual data is done. See Section 22.1 for more information. The value is determined when the database cluster is initialized.

lc_ctype (string)

Reports the locale that determines character classifications. See Section 22.1 for more information. The value is determined when the database cluster is initialized. Ordinarily this will be the same as lc_collate, but for special applications it might be set differently.
max_function_args (integer)

Reports the maximum number of function arguments. It is determined by the value of
FUNC_MAX_ARGS when building the server. The default value is 100 arguments.

max_identifier_length (integer)

Reports the maximum identifier length. It is determined as one less than the value of
NAMEDATALEN when building the server. The default value of NAMEDATALEN is 64; therefore
the default max_identifier_length is 63 bytes.

max_index_keys (integer)

Reports the maximum number of index keys. It is determined by the value of INDEX_MAX_KEYS
when building the server. The default value is 32 keys.

server_encoding (string)

Reports the database encoding (character set). It is determined when the database is created.
Ordinarily, clients need only be concerned with the value of client_encoding.

server_version (string)

Reports the version number of the server. It is determined by the value of PG_VERSION when
building the server.

server_version_num (integer)

Reports the version number of the server as an integer. It is determined by the value of
PG_VERSION_NUM when building the server.

18.14. Customized Options

This feature was designed to allow parameters not normally known to PostgreSQL to be added by
add-on modules (such as procedural languages). This allows add-on modules to be configured in the
standard ways.

custom_variable_classes (string)

This variable specifies one or several class names to be used for custom variables, in the form
of a comma-separated list. A custom variable is a variable not normally known to PostgreSQL
proper but used by some add-on module. Such variables must have names consisting of a class
name, a dot, and a variable name. custom_variable_classes specifies all the class names in
use in a particular installation. This parameter can only be set in the postgresql.conf file or
on the server command line.

The difficulty with setting custom variables in postgresql.conf is that the file must be read before
add-on modules have been loaded, and so custom variables would ordinarily be rejected as unknown.
When custom_variable_classes is set, the server will accept definitions of arbitrary variables
within each specified class. These variables will be treated as placeholders and will have no function
until the module that defines them is loaded. When a module for a specific class is loaded, it will add
the proper variable definitions for its class name, convert any placeholder values according to those
definitions, and issue warnings for any placeholders of its class that remain (which presumably would
be misspelled configuration variables).

Here is an example of what postgresql.conf might contain when using custom variables:

custom_variable_classes = ‘plr,plperl’
plr.path = ‘/usr/lib/R’
plperl.use_strict = true
plruby.use_strict = true  # generates error: unknown class name

18.15. Developer Options

The following parameters are intended for work on the PostgreSQL source, and in some cases to assist with recovery of severely damaged databases. There should be no reason to use them in a production database setup. As such, they have been excluded from the sample `postgresql.conf` file. Note that many of these parameters require special source compilation flags to work at all.

allow_system_table_mods (boolean)

Allows modification of the structure of system tables. This is used by `initdb`. This parameter can only be set at server start.

debg_assertions (boolean)

Turns on various assertion checks. This is a debugging aid. If you are experiencing strange problems or crashes you might want to turn this on, as it might expose programming mistakes. To use this parameter, the macro `USE_ASSERT_CHECKING` must be defined when PostgreSQL is built (accomplished by the configure option `--enable-cassert`). Note that `debg_assertions` defaults to on if PostgreSQL has been built with assertions enabled.

ignore_system_indexes (boolean)

Ignore system indexes when reading system tables (but still update the indexes when modifying the tables). This is useful when recovering from damaged system indexes. This parameter cannot be changed after session start.

post_auth_delay (integer)

If nonzero, a delay of this many seconds occurs when a new server process is started, after it conducts the authentication procedure. This is intended to give an opportunity to attach to the server process with a debugger. This parameter cannot be changed after session start.

pre_auth_delay (integer)

If nonzero, a delay of this many seconds occurs just after a new server process is forked, before it conducts the authentication procedure. This is intended to give an opportunity to attach to the server process with a debugger to trace down misbehavior in authentication. This parameter can only be set in the `postgresql.conf` file or on the server command line.

trace_notify (boolean)

Generates a great amount of debugging output for the LISTEN and NOTIFY commands. `client_min_messages` or `log_min_messages` must be `DEBUG1` or lower to send this output to the client or server log, respectively.

trace_sort (boolean)

If on, emit information about resource usage during sort operations. This parameter is only available if the `TRACE_SORT` macro was defined when PostgreSQL was compiled. (However, `TRACE_SORT` is currently defined by default.)
trace_locks (boolean)
trace_lwlocks (boolean)
trace_userlocks (boolean)
trace_lock_oidmin (boolean)
trace_lock_table (boolean)
debug_deadlocks (boolean)
log_btree_build_stats (boolean)

Various other code tracing and debugging options.

wal_debug (boolean)

If on, emit WAL-related debugging output. This parameter is only available if the WAL_DEBUG macro was defined when PostgreSQL was compiled.

zero_damaged_pages (boolean)

Detection of a damaged page header normally causes PostgreSQL to report an error, aborting the current command. Setting zero_damaged_pages to on causes the system to instead report a warning, zero out the damaged page, and continue processing. This behavior will destroy data, namely all the rows on the damaged page. But it allows you to get past the error and retrieve rows from any undamaged pages that might be present in the table. So it is useful for recovering data if corruption has occurred due to hardware or software error. You should generally not set this on until you have given up hope of recovering data from the damaged page(s) of a table. The default setting is off, and it can only be changed by a superuser.

### 18.16. Short Options

For convenience there are also single letter command-line option switches available for some parameters. They are described in Table 18-2. Some of these options exist for historical reasons, and their presence as a single-letter option does not necessarily indicate an endorsement to use the option heavily.

<table>
<thead>
<tr>
<th>Short option</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>-A x</td>
<td>debug_assertions = x</td>
</tr>
<tr>
<td>-B x</td>
<td>shared_buffers = x</td>
</tr>
<tr>
<td>-d x</td>
<td>log_min_messages = DEBUGx</td>
</tr>
<tr>
<td>-e</td>
<td>datestyle = euro</td>
</tr>
<tr>
<td>-fb, -fh, -fi, -fm, -fn, -fs, -ft</td>
<td>enable_bitmapscan = off, enable_hashjoin = off, enable_indexscan = off, enable_mergejoin = off, enable_nestloop = off, enable_seqscan = off, enable_tidscan = off</td>
</tr>
<tr>
<td>-F</td>
<td>fsync = off</td>
</tr>
<tr>
<td>-h x</td>
<td>listen_addresses = x</td>
</tr>
<tr>
<td>-i</td>
<td>listen_addresses = '*'</td>
</tr>
<tr>
<td>-k x</td>
<td>unix_socket_directory = x</td>
</tr>
<tr>
<td>-l</td>
<td>ssl = on</td>
</tr>
</tbody>
</table>
## Chapter 18. Server Configuration

<table>
<thead>
<tr>
<th>Short option</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>-N x</td>
<td>max_connections = x</td>
</tr>
<tr>
<td>-O</td>
<td>allow_system_table_mods = on</td>
</tr>
<tr>
<td>-p x</td>
<td>port = x</td>
</tr>
<tr>
<td>-P</td>
<td>ignore_system_indexes = on</td>
</tr>
<tr>
<td>-s</td>
<td>log_statement_stats = on</td>
</tr>
<tr>
<td>-S x</td>
<td>work_mem = x</td>
</tr>
<tr>
<td>-tpa, -tpl, -te</td>
<td>log_parser_stats = on, log_planner_stats = on, log_executor_stats = on</td>
</tr>
<tr>
<td>-W x</td>
<td>post_auth_delay = x</td>
</tr>
</tbody>
</table>
Chapter 19. Database Roles and Privileges

PostgreSQL manages database access permissions using the concept of roles. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant membership in a role to another role, thus allowing the member role use of privileges assigned to the role it is a member of.

The concept of roles subsumes the concepts of “users” and “groups”. In PostgreSQL versions before 8.1, users and groups were distinct kinds of entities, but now there are only roles. Any role can act as a user, a group, or both.

This chapter describes how to create and manage roles and introduces the privilege system. More information about the various types of database objects and the effects of privileges can be found in Chapter 5.

19.1. Database Roles

Database roles are conceptually completely separate from operating system users. In practice it might be convenient to maintain a correspondence, but this is not required. Database roles are global across a database cluster installation (and not per individual database). To create a role use the CREATE ROLE SQL command:

```sql
CREATE ROLE name;
```

`name` follows the rules for SQL identifiers: either unadorned without special characters, or double-quoted. (In practice, you will usually want to add additional options, such as LOGIN, to the command. More details appear below.) To remove an existing role, use the analogous DROP ROLE command:

```sql
DROP ROLE name;
```

For convenience, the programs createuser and dropuser are provided as wrappers around these SQL commands that can be called from the shell command line:

```bash
createuser name
dropuser name
```

To determine the set of existing roles, examine the `pg_roles` system catalog, for example

```sql
SELECT rolname FROM pg_roles;
```

The psql program’s `\du` meta-command is also useful for listing the existing roles.

In order to bootstrap the database system, a freshly initialized system always contains one predefined role. This role is always a “superuser”, and by default (unless altered when running `initdb`) it will have the same name as the operating system user that initialized the database cluster. Customarily, this role will be named `postgres`. In order to create more roles you first have to connect as this initial role.

Every connection to the database server is made in the name of some particular role, and this role determines the initial access privileges for commands issued on that connection. The role name to use for a particular database connection is indicated by the client that is initiating the connection request.
in an application-specific fashion. For example, the \texttt{psql} program uses the \texttt{-U} command line option to indicate the role to connect as. Many applications assume the name of the current operating system user by default (including \texttt{createuser} and \texttt{psql}). Therefore it is often convenient to maintain a naming correspondence between roles and operating system users.

The set of database roles a given client connection can connect as is determined by the client authentication setup, as explained in Chapter 21. (Thus, a client is not necessarily limited to connect as the role with the same name as its operating system user, just as a person’s login name need not match her real name.) Since the role identity determines the set of privileges available to a connected client, it is important to carefully configure this when setting up a multiuser environment.

19.2. Role Attributes

A database role can have a number of attributes that define its privileges and interact with the client authentication system.

login privilege

Only roles that have the \texttt{LOGIN} attribute can be used as the initial role name for a database connection. A role with the \texttt{LOGIN} attribute can be considered the same thing as a “database user”. To create a role with login privilege, use either:

\begin{verbatim}
CREATE ROLE name LOGIN;
CREATE USER name;
\end{verbatim}

(\texttt{CREATE USER} is equivalent to \texttt{CREATE ROLE} except that \texttt{CREATE USER} assumes \texttt{LOGIN} by default, while \texttt{CREATE ROLE} does not.)

superuser status

A database superuser bypasses all permission checks. This is a dangerous privilege and should not be used carelessly; it is best to do most of your work as a role that is not a superuser. To create a new database superuser, use \texttt{CREATE ROLE name SUPERUSER}. You must do this as a role that is already a superuser.

database creation

A role must be explicitly given permission to create databases (except for superusers, since those bypass all permission checks). To create such a role, use \texttt{CREATE ROLE name CREATEDB}.

role creation

A role must be explicitly given permission to create more roles (except for superusers, since those bypass all permission checks). To create such a role, use \texttt{CREATE ROLE name CREATEROLE}. A role with \texttt{CREATEROLE} privilege can alter and drop other roles, too, as well as grant or revoke membership in them. However, to create, alter, drop, or change membership of a superuser role, superuser status is required; \texttt{CREATEROLE} is not sufficient for that.

password

A password is only significant if the client authentication method requires the user to supply a password when connecting to the database. The \texttt{password}, \texttt{md5}, and \texttt{crypt} authentication methods make use of passwords. Database passwords are separate from operating system passwords. Specify a password upon role creation with \texttt{CREATE ROLE name PASSWORD ‘string’}.

A role’s attributes can be modified after creation with \texttt{ALTER ROLE}. See the reference pages for the \texttt{CREATE ROLE} and \texttt{ALTER ROLE} commands for details.
Tip: It is good practice to create a role that has the \texttt{CREATEDB} and \texttt{CREATEROLE} privileges, but is not a superuser, and then use this role for all routine management of databases and roles. This approach avoids the dangers of operating as a superuser for tasks that do not really require it.

A role can also have role-specific defaults for many of the run-time configuration settings described in Chapter 18. For example, if for some reason you want to disable index scans (hint: not a good idea) anytime you connect, you can use:

\begin{verbatim}
ALTER ROLE myname SET enable_indexscan TO off;
\end{verbatim}

This will save the setting (but not set it immediately). In subsequent connections by this role it will appear as though \texttt{SET enable_indexscan TO off;} had been executed just before the session started. You can still alter this setting during the session; it will only be the default. To remove a role-specific default setting, use \texttt{ALTER ROLE rolename RESET varname;}. Note that role-specific defaults attached to roles without \texttt{LOGIN} privilege are fairly useless, since they will never be invoked.

### 19.3. Privileges

When an object is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, \textit{privileges} must be granted. There are several different kinds of privilege: \texttt{SELECT}, \texttt{INSERT}, \texttt{UPDATE}, \texttt{DELETE}, \texttt{REFERENCES}, \texttt{TRIGGER}, \texttt{CREATE}, \texttt{CONNECT}, \texttt{TEMPORARY}, \texttt{EXECUTE}, and \texttt{USAGE}. For more information on the different types of privileges supported by PostgreSQL, see the \texttt{GRANT} reference page.

To assign privileges, the \texttt{GRANT} command is used. So, if \texttt{joe} is an existing role, and \texttt{accounts} is an existing table, the privilege to update the table can be granted with:

\begin{verbatim}
GRANT UPDATE ON accounts TO joe;
\end{verbatim}

The special name \texttt{PUBLIC} can be used to grant a privilege to every role on the system. Writing \texttt{ALL} in place of a specific privilege specifies that all privileges that apply to the object will be granted.

To revoke a privilege, use the fittingly named \texttt{REVOKE} command:

\begin{verbatim}
REVOKE ALL ON accounts FROM PUBLIC;
\end{verbatim}

The special privileges of an object’s owner (i.e., the right to modify or destroy the object) are always implicit in being the owner, and cannot be granted or revoked. But the owner can choose to revoke his own ordinary privileges, for example to make a table read-only for himself as well as others.

An object can be assigned to a new owner with an \texttt{ALTER} command of the appropriate kind for the object. Superusers can always do this; ordinary roles can only do it if they are both the current owner of the object (or a member of the owning role) and a member of the new owning role.

### 19.4. Role Membership

It is frequently convenient to group users together to ease management of privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In PostgreSQL this is done by creating
a role that represents the group, and then granting membership in the group role to individual user roles.

To set up a group role, first create the role:

```
CREATE ROLE name;
```

Typically a role being used as a group would not have the LOGIN attribute, though you can set it if you wish.

Once the group role exists, you can add and remove members using the GRANT and REVOKE commands:

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

You can grant membership to other group roles, too (since there isn’t really any distinction between group roles and non-group roles). The database will not let you set up circular membership loops. Also, it is not permitted to grant membership in a role to PUBLIC.

The members of a role can use the privileges of the group role in two ways. First, every member of a group can explicitly do SET ROLE to temporarily “become” the group role. In this state, the database session has access to the privileges of the group role rather than the original login role, and any database objects created are considered owned by the group role not the login role. Second, member roles that have the INHERIT attribute automatically have use of privileges of roles they are members of. As an example, suppose we have done:

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

Immediately after connecting as role joe, a database session will have use of privileges granted directly to joe plus any privileges granted to admin, because joe “inherits” admin’s privileges. However, privileges granted to wheel are not available, because even though joe is indirectly a member of wheel, the membership is via admin which has the NOINHERIT attribute. After:

```
SET ROLE admin;
```

the session would have use of only those privileges granted to admin, and not those granted to joe. After:

```
SET ROLE wheel;
```

the session would have use of only those privileges granted to wheel, and not those granted to either joe or admin. The original privilege state can be restored with any of:

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

Note: The SET ROLE command always allows selecting any role that the original login role is directly or indirectly a member of. Thus, in the above example, it is not necessary to become admin before becoming wheel.
Note: In the SQL standard, there is a clear distinction between users and roles, and users do not automatically inherit privileges while roles do. This behavior can be obtained in PostgreSQL by giving roles being used as SQL roles the INHERIT attribute, while giving roles being used as SQL users the NOINHERIT attribute. However, PostgreSQL defaults to giving all roles the INHERIT attribute, for backwards compatibility with pre-8.1 releases in which users always had use of permissions granted to groups they were members of.

The role attributes LOGIN, SUPERUSER, CREATEDB, and CREATEROLE can be thought of as special privileges, but they are never inherited as ordinary privileges on database objects are. You must actually SET ROLE to a specific role having one of these attributes in order to make use of the attribute. Continuing the above example, we might well choose to grant CREATEDB and CREATEROLE to the admin role. Then a session connecting as role joe would not have these privileges immediately, only after doing SET ROLE admin.

To destroy a group role, use DROP ROLE:

DROP ROLE name;

Any memberships in the group role are automatically revoked (but the member roles are not otherwise affected). Note however that any objects owned by the group role must first be dropped or reassigned to other owners; and any permissions granted to the group role must be revoked.

### 19.5. Functions and Triggers

Functions and triggers allow users to insert code into the backend server that other users might execute unintentionally. Hence, both mechanisms permit users to “Trojan horse” others with relative ease. The only real protection is tight control over who can define functions.

Functions run inside the backend server process with the operating system permissions of the database server daemon. If the programming language used for the function allows unchecked memory accesses, it is possible to change the server’s internal data structures. Hence, among many other things, such functions can circumvent any system access controls. Function languages that allow such access are considered “untrusted”, and PostgreSQL allows only superusers to create functions written in those languages.
Chapter 20. Managing Databases

Every instance of a running PostgreSQL server manages one or more databases. Databases are therefore the topmost hierarchical level for organizing SQL objects ("database objects"). This chapter describes the properties of databases, and how to create, manage, and destroy them.

20.1. Overview

A database is a named collection of SQL objects ("database objects"). Generally, every database object (tables, functions, etc.) belongs to one and only one database. (But there are a few system catalogs, for example `pg_database`, that belong to a whole cluster and are accessible from each database within the cluster.) More accurately, a database is a collection of schemas and the schemas contain the tables, functions, etc. So the full hierarchy is: server, database, schema, table (or some other kind of object, such as a function).

When connecting to the database server, a client must specify in its connection request the name of the database it wants to connect to. It is not possible to access more than one database per connection. (But an application is not restricted in the number of connections it opens to the same or other databases.) Databases are physically separated and access control is managed at the connection level. If one PostgreSQL server instance is to house projects or users that should be separate and for the most part unaware of each other, it is therefore recommendable to put them into separate databases. If the projects or users are interrelated and should be able to use each other's resources they should be put in the same database, but possibly into separate schemas. Schemas are a purely logical structure and who can access what is managed by the privilege system. More information about managing schemas is in Section 5.7.

Databases are created with the `CREATE DATABASE` command (see Section 20.2) and destroyed with the `DROP DATABASE` command (see Section 20.5). To determine the set of existing databases, examine the `pg_database` system catalog, for example

```
SELECT datname FROM pg_database;
```

The psql program's `\l` meta-command and `-l` command-line option are also useful for listing the existing databases.

**Note:** The SQL standard calls databases "catalogs", but there is no difference in practice.

20.2. Creating a Database

In order to create a database, the PostgreSQL server must be up and running (see Section 17.3).

Databases are created with the SQL command `CREATE DATABASE`:

```
CREATE DATABASE name;
```

where `name` follows the usual rules for SQL identifiers. The current role automatically becomes the owner of the new database. It is the privilege of the owner of a database to remove it later on (which also removes all the objects in it, even if they have a different owner).

The creation of databases is a restricted operation. See Section 19.2 for how to grant permission.
Chapter 20. Managing Databases

Since you need to be connected to the database server in order to execute the CREATE DATABASE command, the question remains how the first database at any given site can be created. The first database is always created by the initdb command when the data storage area is initialized. (See Section 17.2.) This database is called postgres. So to create the first “ordinary” database you can connect to postgres.

A second database, template1, is also created by initdb. Whenever a new database is created within the cluster, template1 is essentially cloned. This means that any changes you make in template1 are propagated to all subsequently created databases. Therefore it is unwise to use template1 for real work, but when used judiciously this feature can be convenient. More details appear in Section 20.3.

As a convenience, there is a program that you can execute from the shell to create new databases, createdb.

createdb dbname

createdb does no magic. It connects to the postgres database and issues the CREATE DATABASE command, exactly as described above. The createdb reference page contains the invocation details. Note that createdb without any arguments will create a database with the current user name, which might or might not be what you want.

Note: Chapter 21 contains information about how to restrict who can connect to a given database.

Sometimes you want to create a database for someone else. That role should become the owner of the new database, so he can configure and manage it himself. To achieve that, use one of the following commands:

CREATE DATABASE dbname OWNER rolename;

from the SQL environment, or:

createdb -O rolename dbname

from the shell. You must be a superuser to be allowed to create a database for someone else (that is, for a role you are not a member of).

20.3. Template Databases

CREATE DATABASE actually works by copying an existing database. By default, it copies the standard system database named template1. Thus that database is the “template” from which new databases are made. If you add objects to template1, these objects will be copied into subsequently created user databases. This behavior allows site-local modifications to the standard set of objects in databases. For example, if you install the procedural language PL/pgSQL in template1, it will automatically be available in user databases without any extra action being taken when those databases are made.

There is a second standard system database named template0. This database contains the same data as the initial contents of template1, that is, only the standard objects predefined by your version of PostgreSQL. template0 should never be changed after initdb. By instructing CREATE DATABASE to copy template0 instead of template1, you can create a “virgin” user database that contains none of the site-local additions in template1. This is particularly handy when restoring a pg_dump dump:
the dump script should be restored in a virgin database to ensure that one recreates the correct contents of the dumped database, without any conflicts with additions that can now be present in `template1`.

To create a database by copying `template0`, use:

```
CREATE DATABASE dbname TEMPLATE template0;
```

from the SQL environment, or:

```
createdb -T template0 dbname
```

from the shell.

It is possible to create additional template databases, and indeed one can copy any database in a cluster by specifying its name as the template for `CREATE DATABASE`. It is important to understand, however, that this is not (yet) intended as a general-purpose "COPY DATABASE" facility. The principal limitation is that no other sessions can be connected to the source database while it is being copied. `CREATE DATABASE` will fail if any other connection exists when it starts; otherwise, new connections to the source database are locked out until `CREATE DATABASE` completes.

Two useful flags exist in `pg_database` for each database: the columns `datistemplate` and `datallowconn`. `datistemplate` can be set to indicate that a database is intended as a template for `CREATE DATABASE`. If this flag is set, the database can be cloned by any user with `CREATEDB` privileges; if it is not set, only superusers and the owner of the database can clone it. If `datallowconn` is false, then no new connections to that database will be allowed (but existing sessions are not killed simply by setting the flag false). The `template0` database is normally marked `datallowconn = false` to prevent modification of it. Both `template0` and `template1` should always be marked with `datistemplate = true`.

Note: `template1` and `template0` do not have any special status beyond the fact that the name `template1` is the default source database name for `CREATE DATABASE`. For example, one could drop `template1` and recreate it from `template0` without any ill effects. This course of action might be advisable if one has carelessly added a bunch of junk in `template1`. (To delete `template1`, it must have `pg_database.datistemplate = false`.)

The `postgres` database is also created when a database cluster is initialized. This database is meant as a default database for users and applications to connect to. It is simply a copy of `template1` and can be dropped and recreated if required.

### 20.4. Database Configuration

Recall from Chapter 18 that the PostgreSQL server provides a large number of run-time configuration variables. You can set database-specific default values for many of these settings.

For example, if for some reason you want to disable the GEQO optimizer for a given database, you’d ordinarily have to either disable it for all databases or make sure that every connecting client is careful to issue `SET geqo TO off;`. To make this setting the default within a particular database, you can execute the command:

```
ALTER DATABASE mydb SET geqo TO off;
```

This will save the setting (but not set it immediately). In subsequent connections to this database it will appear as though `SET geqo TO off;` had been executed just before the session started. Note
that users can still alter this setting during their sessions; it will only be the default. To undo any such setting, use \texttt{ALTER DATABASE dbname RESET varname}.

### 20.5. Destroying a Database

Databases are destroyed with the command \texttt{DROP DATABASE}:

\begin{verbatim}
DROP DATABASE name;
\end{verbatim}

Only the owner of the database, or a superuser, can drop a database. Dropping a database removes all objects that were contained within the database. The destruction of a database cannot be undone.

You cannot execute the \texttt{DROP DATABASE} command while connected to the victim database. You can, however, be connected to any other database, including the \texttt{template1} database. \texttt{template1} would be the only option for dropping the last user database of a given cluster.

For convenience, there is also a shell program to drop databases, \texttt{dropdb}:

\begin{verbatim}
dropdb dbname
\end{verbatim}

(Unlike \texttt{createdb}, it is not the default action to drop the database with the current user name.)

### 20.6. Tablespaces

Tablespaces in PostgreSQL allow database administrators to define locations in the file system where the files representing database objects can be stored. Once created, a tablespace can be referred to by name when creating database objects.

By using tablespaces, an administrator can control the disk layout of a PostgreSQL installation. This is useful in at least two ways. First, if the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.

Second, tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

To define a tablespace, use the \texttt{CREATE TABLESPACE} command, for example:

\begin{verbatim}
CREATE TABLESPACE fastspace LOCATION '/mnt/sda1/postgresql/data';
\end{verbatim}

The location must be an existing, empty directory that is owned by the PostgreSQL system user. All objects subsequently created within the tablespace will be stored in files underneath this directory.

\textbf{Note:} There is usually not much point in making more than one tablespace per logical file system, since you cannot control the location of individual files within a logical file system. However, PostgreSQL does not enforce any such limitation, and indeed it is not directly aware of the file system boundaries on your system. It just stores files in the directories you tell it to use.
Creation of the tablespace itself must be done as a database superuser, but after that you can allow ordinary database users to make use of it. To do that, grant them the `CREATE` privilege on it.

Tables, indexes, and entire databases can be assigned to particular tablespaces. To do so, a user with the `CREATE` privilege on a given tablespace must pass the tablespace name as a parameter to the relevant command. For example, the following creates a table in the tablespace `space1`:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

Alternatively, use the `default_tablespace` parameter:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

When `default_tablespace` is set to anything but an empty string, it supplies an implicit `TABLESPACE` clause for `CREATE TABLE` and `CREATE INDEX` commands that do not have an explicit one.

There is also a `temp_tablespaces` parameter, which determines the placement of temporary tables and indexes, as well as temporary files that are used for purposes such as sorting large data sets. This can be a list of tablespace names, rather than only one, so that the load associated with temporary objects can be spread over multiple tablespaces. A random member of the list is picked each time a temporary object is to be created.

The tablespace associated with a database is used to store the system catalogs of that database. Furthermore, it is the default tablespace used for tables, indexes, and temporary files created within the database, if no `TABLESPACE` clause is given and no other selection is specified by `default_tablespace` or `temp_tablespaces` (as appropriate). If a database is created without specifying a tablespace for it, it uses the same tablespace as the template database it is copied from.

Two tablespaces are automatically created by `initdb`. The `pg_global` tablespace is used for shared system catalogs. The `pg_default` tablespace is the default tablespace of the `template1` and `template0` databases (and, therefore, will be the default tablespace for other databases as well, unless overridden by a `TABLESPACE` clause in `CREATE DATABASE`).

Once created, a tablespace can be used from any database, provided the requesting user has sufficient privilege. This means that a tablespace cannot be dropped until all objects in all databases using the tablespace have been removed.

To remove an empty tablespace, use the `DROP TABLESPACE` command.

To determine the set of existing tablespaces, examine the `pg_tablespace` system catalog, for example

```
SELECT spcname FROM pg_tablespace;
```

The psql program’s `\db` meta-command is also useful for listing the existing tablespaces.

PostgreSQL makes use of symbolic links to simplify the implementation of tablespaces. This means that tablespaces can be used only on systems that support symbolic links.

The directory `$PGDATA/pg_tblspc` contains symbolic links that point to each of the non-built-in tablespaces defined in the cluster. Although not recommended, it is possible to adjust the tablespace layout by hand by redefining these links. Two warnings: do not do so while the server is running; and after you restart the server, update the `pg_tablespace` catalog to show the new locations. (If you do not, `pg_dump` will continue to show the old tablespace locations.)
Chapter 21. Client Authentication

When a client application connects to the database server, it specifies which PostgreSQL database user name it wants to connect as, much the same way one logs into a Unix computer as a particular user. Within the SQL environment the active database user name determines access privileges to database objects — see Chapter 19 for more information. Therefore, it is essential to restrict which database users can connect.

Note: As explained in Chapter 19, PostgreSQL actually does privilege management in terms of “roles”. In this chapter, we consistently use database user to mean “role with the LOGIN privilege”.

Authentication is the process by which the database server establishes the identity of the client, and by extension determines whether the client application (or the user who runs the client application) is permitted to connect with the database user name that was requested.

PostgreSQL offers a number of different client authentication methods. The method used to authenticate a particular client connection can be selected on the basis of (client) host address, database, and user.

PostgreSQL database user names are logically separate from user names of the operating system in which the server runs. If all the users of a particular server also have accounts on the server’s machine, it makes sense to assign database user names that match their operating system user names. However, a server that accepts remote connections might have many database users who have no local operating system account, and in such cases there need be no connection between database user names and OS user names.

21.1. The pg_hba.conf file

Client authentication is controlled by a configuration file, which traditionally is named pg_hba.conf and is stored in the database cluster’s data directory. (HBA stands for host-based authentication.) A default pg_hba.conf file is installed when the data directory is initialized by initdb. It is possible to place the authentication configuration file elsewhere, however; see the hba_file configuration parameter.

The general format of the pg_hba.conf file is a set of records, one per line. Blank lines are ignored, as is any text after the # comment character. A record is made up of a number of fields which are separated by spaces and/or tabs. Fields can contain white space if the field value is quoted. Records cannot be continued across lines.

Each record specifies a connection type, a client IP address range (if relevant for the connection type), a database name, a user name, and the authentication method to be used for connections matching these parameters. The first record with a matching connection type, client address, requested database, and user name is used to perform authentication. There is no “fall-through” or “backup”: if one record is chosen and the authentication fails, subsequent records are not considered. If no record matches, access is denied.

A record can have one of the seven formats

- local
database user auth-method [auth-option]
- host
database user CIDR-address auth-method [auth-option]
- hostssl
database user CIDR-address auth-method [auth-option]
- hostnossl
database user CIDR-address auth-method [auth-option]
Chapter 21. Client Authentication

The meaning of the fields is as follows:

**host**

This record matches connection attempts using Unix-domain sockets. Without a record of this type, Unix-domain socket connections are disallowed.

**hostssl**

This record matches connection attempts made using TCP/IP. *host* records match either SSL or non-SSL connection attempts.

Note: Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the `listen_addresses` configuration parameter, since the default behavior is to listen for TCP/IP connections only on the local loopback address `localhost`.

**hostnossl**

This record type has the opposite logic to *hostssl*: it only matches connection attempts made over TCP/IP that do not use SSL.

**database**

Specifies which database names this record matches. The value *all* specifies that it matches all databases. The value *sameuser* specifies that the record matches if the requested database has the same name as the requested user. The value *samerole* specifies that the requested user must be a member of the role with the same name as the requested database. (*samegroup* is an obsolete but still accepted spelling of *samerole*.) Otherwise, this is the name of a specific PostgreSQL database. Multiple database names can be supplied by separating them with commas. A separate file containing database names can be specified by preceding the file name with `@`.

**user**

Specifies which database user names this record matches. The value *all* specifies that it matches all users. Otherwise, this is either the name of a specific database user, or a group name preceded by `+`. (Recall that there is no real distinction between users and groups in PostgreSQL; a `+` mark really means “match any of the roles that are directly or indirectly members of this role”, while a name without a `+` mark matches only that specific role.) Multiple user names can be supplied by separating them with commas. A separate file containing user names can be specified by preceding the file name with `@`.

**CIDR-address**

Specifies the client machine IP address range that this record matches. It contains an IP address in standard dotted decimal notation and a CIDR mask length. (IP addresses can only be specified
Chapter 21. Client Authentication

numerically, not as domain or host names.) The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this must be zero in the given IP address. There must not be any white space between the IP address, the /, and the CIDR mask length.

Typical examples of a CIDR-address are 172.20.143.89/32 for a single host, or 172.20.143.0/24 for a small network, or 10.6.0.0/16 for a larger one. To specify a single host, use a CIDR mask of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes.

An IP address given in IPv4 format will match IPv6 connections that have the corresponding address, for example 127.0.0.1 will match the IPv6 address ::ffff:127.0.0.1. An entry given in IPv6 format will match only IPv6 connections, even if the represented address is in the IPv4-in-IPv6 range. Note that entries in IPv6 format will be rejected if the system’s C library does not have support for IPv6 addresses.

This field only applies to host, hostssl, and hostnossl records.

IP-address

IP-mask

These fields can be used as an alternative to the CIDR-address notation. Instead of specifying the mask length, the actual mask is specified in a separate column. For example, 255.0.0.0 represents an IPv4 CIDR mask length of 8, and 255.255.255.255 represents a CIDR mask length of 32.

These fields only apply to host, hostssl, and hostnossl records.

auth-method

Specifies the authentication method to use when connecting via this record. The possible choices are summarized here; details are in Section 21.2.

trust

Allow the connection unconditionally. This method allows anyone that can connect to the PostgreSQL database server to login as any PostgreSQL user they like, without the need for a password. See Section 21.2.1 for details.

reject

Reject the connection unconditionally. This is useful for “filtering out” certain hosts from a group.

md5

Require the client to supply an MD5-encrypted password for authentication. See Section 21.2.2 for details.

crypt

Note: This option is recommended only for communicating with pre-7.2 clients.

Require the client to supply a crypt()-encrypted password for authentication. md5 is now recommended over crypt. See Section 21.2.2 for details.
password

Require the client to supply an unencrypted password for authentication. Since the password is sent in clear text over the network, this should not be used on untrusted networks. It also does not usually work with threaded client applications. See Section 21.2.2 for details.

gss

Use GSSAPI to authenticate the user. This is only available for TCP/IP connections. See Section 21.2.3 for details.

sspi

Use SSPI to authenticate the user. This is only available on Windows. See Section 21.2.4 for details.

krb5

Use Kerberos V5 to authenticate the user. This is only available for TCP/IP connections. See Section 21.2.5 for details.

ident

Obtain the operating system user name of the client (for TCP/IP connections by contacting the ident server on the client, for local connections by getting it from the operating system) and check if the user is allowed to connect as the requested database user by consulting the map specified after the `ident` key word. See Section 21.2.6 for details.

ldap

Authenticate using LDAP to a central server. See Section 21.2.7 for details.

pam

Authenticate using the Pluggable Authentication Modules (PAM) service provided by the operating system. See Section 21.2.8 for details.

auth-option

The meaning of this optional field depends on the chosen authentication method. Details appear below.

Files included by `@` constructs are read as lists of names, which can be separated by either whitespace or commas. Comments are introduced by `#`, just as in `pg_hba.conf`, and nested `@` constructs are allowed. Unless the file name following `@` is an absolute path, it is taken to be relative to the directory containing the referencing file.

Since the `pg_hba.conf` records are examined sequentially for each connection attempt, the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example, one might wish to use `trust` authentication for local TCP/IP connections but require a password for remote TCP/IP connections. In this case a record specifying `trust` authentication for connections from 127.0.0.1 would appear before a record specifying password authentication for a wider range of allowed client IP addresses.

The `pg_hba.conf` file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the server (using `pg_ctl reload` or `kill -HUP`) to make it re-read the file.
Tip: To connect to a particular database, a user must not only pass the `pg_hba.conf` checks, but must have the `CONNECT` privilege for the database. If you wish to restrict which users can connect to which databases, it's usually easier to control this by granting/revoking `CONNECT` privilege than to put the rules into `pg_hba.conf` entries.

Some examples of `pg_hba.conf` entries are shown in Example 21-1. See the next section for details on the different authentication methods.

**Example 21-1. Example `pg_hba.conf` entries**

```plaintext
# Allow any user on the local system to connect to any database under
# any database user name using Unix-domain sockets (the default for local
# connections).
#
# TYPE    DATABASE    USER    CIDR-ADDRESS    METHOD
local    all         all      all            trust
#
# The same using local loopback TCP/IP connections.
#
# TYPE    DATABASE    USER    CIDR-ADDRESS    METHOD
host     all         all      127.0.0.1/32    trust
#
# The same as the last line but using a separate netmask column
#
# TYPE    DATABASE    USER    IP-ADDRESS    IP-MASK    METHOD
host     all         all      127.0.0.1       255.255.255.255 trust
#
# Allow any user from any host with IP address 192.168.93.x to connect
to database "postgres" as the same user name that ident reports for
# the connection (typically the Unix user name).
#
# TYPE    DATABASE    USER    CIDR-ADDRESS    METHOD
host     postgres    all      192.168.93.0/24 ident sameuser
#
# Allow a user from host 192.168.12.10 to connect to database
# "postgres" if the user's password is correctly supplied.
#
# TYPE    DATABASE    USER    CIDR-ADDRESS    METHOD
host     postgres    all      192.168.12.10/32 md5
#
# In the absence of preceding "host" lines, these two lines will
# reject all connection from 192.168.54.1 (since that entry will be
# matched first), but allow Kerberos 5 connections from anywhere else
# on the Internet. The zero mask means that no bits of the host IP
# address are considered so it matches any host.
#
# TYPE    DATABASE    USER    CIDR-ADDRESS    METHOD
host     all         all      192.168.54.1/32 reject
host     all         all      0.0.0.0/0      krb5
#
# Allow users from 192.168.x.x hosts to connect to any database, if
# they pass the ident check. If, for example, ident says the user is
# "bryanh" and he requests to connect as PostgreSQL user "guest1", the
# connection is allowed if there is an entry in `pg_ident.conf` for map
# "omicron" that says "bryanh" is allowed to connect as "guest1".
```
Chapter 21. Client Authentication

### 21.2. Authentication methods

The following subsections describe the authentication methods in more detail.

#### 21.2.1. Trust authentication

When trust authentication is specified, PostgreSQL assumes that anyone who can connect to the server is authorized to access the database with whatever database user name they specify (including superusers). Of course, restrictions made in the database and user columns still apply. This method should only be used when there is adequate operating-system-level protection on connections to the server.

Trust authentication is appropriate and very convenient for local connections on a single-user workstation. It is usually not appropriate by itself on a multiuser machine. However, you might be able to use trust even on a multiuser machine, if you restrict access to the server’s Unix-domain socket file using file-system permissions. To do this, set the `unix_socket_permissions` (and possibly `unix_socket_group`) configuration parameters as described in Section 18.3. Or you could set the `unix_socket_directory` configuration parameter to place the socket file in a suitably restricted directory.

Setting file-system permissions only helps for Unix-socket connections. Local TCP/IP connections are not restricted by it; therefore, if you want to use file-system permissions for local security, remove the host ... 127.0.0.1 ... line from `pg_hba.conf`, or change it to a non-trust authentication method.

Trust authentication is only suitable for TCP/IP connections if you trust every user on every machine that is allowed to connect to the server by the `pg_hba.conf` lines that specify trust. It is seldom reasonable to use trust for any TCP/IP connections other than those from localhost (127.0.0.1).
21.2.2. Password authentication

The password-based authentication methods are `md5`, `crypt`, and `password`. These methods operate similarly except for the way that the password is sent across the connection: respectively, MD5-hashed, crypt-encrypted, and clear-text. A limitation is that the `crypt` method does not work with passwords that have been encrypted in `pg_authid`.

If you are at all concerned about password “sniffing” attacks then `md5` is preferred, with `crypt` to be used only if you must support pre-7.2 clients. Plain `password` should be avoided especially for connections over the open Internet (unless you use SSL, SSH, or another communications security wrapper around the connection).

PostgreSQL database passwords are separate from operating system user passwords. The password for each database user is stored in the `pg_authid` system catalog. Passwords can be managed with the SQL commands `CREATE USER` and `ALTER USER`, e.g., `CREATE USER foo WITH PASSWORD 'secret';`. By default, that is, if no password has been set up, the stored password is null and password authentication will always fail for that user.

21.2.3. GSSAPI authentication

GSSAPI is an industry-standard protocol for secure authentication defined in RFC 2743. PostgreSQL supports GSSAPI with Kerberos authentication according to RFC 1964. GSSAPI provides automatic authentication (single sign-on) for systems that support it. The authentication itself is secure, but the data sent over the connection will be in clear unless SSL is used.

When GSSAPI uses Kerberos, it uses a standard principal in format `servicename/hostname@realm`. For information about the parts of the principal, and how to set up the required keys, see Section 21.2.5. GSSAPI support has to be enabled when PostgreSQL is built; see Chapter 15 for more information.

21.2.4. SSPI authentication

SSPI is a Windows technology for secure authentication with single sign-on. PostgreSQL will use SSPI in `negotiate` mode, which will use Kerberos when possible and automatically fall back to NTLM in other cases. SSPI authentication only works when both server and client are running Windows.

When using Kerberos authentication, SSPI works the same way GSSAPI does. See Section 21.2.3 for details.

21.2.5. Kerberos authentication

Note: Native Kerberos authentication has been deprecated and should be used only for backward compatibility. New and upgraded installations are encouraged to use the industry-standard GSSAPI authentication (see Section 21.2.3) instead.

Kerberos is an industry-standard secure authentication system suitable for distributed computing over a public network. A description of the Kerberos system is far beyond the scope of this document; in
Chapter 21. Client Authentication

full generality it can be quite complex (yet powerful). The Kerberos FAQ¹ or MIT Kerberos page² can be good starting points for exploration. Several sources for Kerberos distributions exist. Kerberos provides secure authentication but does not encrypt queries or data passed over the network; for that use SSL.

PostgreSQL supports Kerberos version 5. Kerberos support has to be enabled when PostgreSQL is built; see Chapter 15 for more information.

PostgreSQL operates like a normal Kerberos service. The name of the service principal is servicename/hostname@realm.

servicename can be set on the server side using the krb_srvname configuration parameter, and on the client side using the krbsrvname connection parameter. (See also Section 30.1.) The installation default can be changed from the default postgres at build time using ./configure --with-krb-srvname=whatever. In most environments, this parameter never needs to be changed. However, to support multiple PostgreSQL installations on the same host it is necessary. Some Kerberos implementations might also require a different service name, such as Microsoft Active Directory which requires the service name to be in uppercase (POSTGRES).

hostname is the fully qualified host name of the server machine. The service principal’s realm is the preferred realm of the server machine.

Client principals must have their PostgreSQL database user name as their first component, for example pgusertextname@realm. By default, the realm of the client is not checked by PostgreSQL. If you have cross-realm authentication enabled and need to verify the realm, use the krb_realm parameter.

Make sure that your server keytab file is readable (and preferably only readable) by the PostgreSQL server account. (See also Section 17.1.) The location of the key file is specified by the krb_server_keyfile configuration parameter. The default is /usr/local/pgsql/etc/krb5.keytab (or whichever directory was specified as sysconfdir at build time).

The keytab file is generated by the Kerberos software; see the Kerberos documentation for details. The following example is for MIT-compatible Kerberos 5 implementations:

```
  kadmin% ank -randkey postgres/server.my.domain.org
  kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

When connecting to the database make sure you have a ticket for a principal matching the requested database user name. For example, for database user name fred, both principal fred@example.com and fred/users.example.com@example.com could be used to authenticate to the database server.

If you use mod_auth_kerb³ and mod_perl on your Apache web server, you can use AuthType KerberosV5SaveCredentials with a mod_perl script. This gives secure database access over the web, no extra passwords required.

### 21.2.6. Ident-based authentication

The ident authentication method works by obtaining the client’s operating system user name, then determining the allowed database user names using a map file that lists the permitted corresponding

---

pairs of names. The determination of the client’s user name is the security-critical point, and it works differently depending on the connection type.

### 21.2.6.1. Ident Authentication over TCP/IP

The “Identification Protocol” is described in RFC 1413. Virtually every Unix-like operating system ships with an ident server that listens on TCP port 113 by default. The basic functionality of an ident server is to answer questions like “What user initiated the connection that goes out of your port \( x \) and connects to my port \( y \)?". Since PostgreSQL knows both \( x \) and \( y \) when a physical connection is established, it can interrogate the ident server on the host of the connecting client and could theoretically determine the operating system user for any given connection this way.

The drawback of this procedure is that it depends on the integrity of the client: if the client machine is untrusted or compromised an attacker could run just about any program on port 113 and return any user name he chooses. This authentication method is therefore only appropriate for closed networks where each client machine is under tight control and where the database and system administrators operate in close contact. In other words, you must trust the machine running the ident server. Heed the warning:

> The Identification Protocol is not intended as an authorization or access control protocol.  
> —RFC 1413

Some ident servers have a nonstandard option that causes the returned user name to be encrypted, using a key that only the originating machine’s administrator knows. This option must not be used when using the ident server with PostgreSQL, since PostgreSQL does not have any way to decrypt the returned string to determine the actual user name.

### 21.2.6.2. Ident Authentication over Local Sockets

On systems supporting `SO_PEERCRED` requests for Unix-domain sockets (currently Linux, FreeBSD, NetBSD, OpenBSD, and BSD/OS), ident authentication can also be applied to local connections. In this case, no security risk is added by using ident authentication; indeed it is a preferable choice for local connections on such systems.

On systems without `SO_PEERCRED` requests, ident authentication is only available for TCP/IP connections. As a work-around, it is possible to specify the localhost address 127.0.0.1 and make connections to this address. This method is trustworthy to the extent that you trust the local ident server.

### 21.2.6.3. Ident Maps

When using ident-based authentication, after having determined the name of the operating system user that initiated the connection, PostgreSQL checks whether that user is allowed to connect as the database user he is requesting to connect as. This is controlled by the ident map argument that follows the `ident` key word in the `pg_hba.conf` file. There is a predefined ident map `sameuser`, which allows any operating system user to connect as the database user of the same name (if the latter exists). Other maps must be created manually.

Ident maps other than `sameuser` are defined in the ident map file, which by default is named `pg_ident.conf` and is stored in the cluster’s data directory. (It is possible to place the map file elsewhere, however; see the `ident_file` configuration parameter.) The ident map file contains lines of the general form:

```
map-name ident-username database-username
```
Comments and whitespace are handled in the same way as in \texttt{pg\_hba.conf}. The map-name is an arbitrary name that will be used to refer to this mapping in \texttt{pg\_hba.conf}. The other two fields specify which operating system user is allowed to connect as which database user. The same map-name can be used repeatedly to specify more user-mappings within a single map. There is no restriction regarding how many database users a given operating system user can correspond to, nor vice versa.

The \texttt{pg\_ident.conf} file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the server (using \texttt{pg\_ctl reload} or \texttt{kill \ -HUP}) to make it re-read the file.

A \texttt{pg\_ident.conf} file that could be used in conjunction with the \texttt{pg\_hba.conf} file in Example 21-1 is shown in Example 21-2. In this example setup, anyone logged in to a machine on the 192.168 network that does not have the Unix user name \texttt{bryan}, \texttt{ann}, or \texttt{robert} would not be granted access. Unix user \texttt{robert} would only be allowed access when he tries to connect as PostgreSQL user \texttt{bob}, not as \texttt{robert} or anyone else. \texttt{ann} would only be allowed to connect as \texttt{ann}. User \texttt{bryan} would be allowed to connect as either \texttt{bryan} himself or as \texttt{guest1}.

\texttt{Example 21-2. An example pg\_ident.conf file}

\begin{verbatim}
# MAPNAME IDENT-USERNAME PG-USERNAME
omicron bryan bryan
omicron ann ann
# bob has user name robert on these machines
omicron robert bob
# bryan can also connect as guest1
omicron bryan guest1
\end{verbatim}

\subsection{21.2.7. LDAP authentication}

This authentication method operates similarly to \texttt{password} except that it uses LDAP as the authentication method. LDAP is used only to validate the user name/password pairs. Therefore the user must already exist in the database before LDAP can be used for authentication. The server and parameters used are specified after the \texttt{ldap} key word in the file \texttt{pg\_hba.conf}. The format of this parameter is:

\texttt{ldap[s]://servername[:port]/base\:dn[\:prefix[\:suffix]]}

Commas are used to specify multiple items in an \texttt{ldap} component. However, because unquoted commas are treated as item separators in \texttt{pg\_hba.conf}, it is wise to double-quote the \texttt{ldap} URL to preserve any commas present, e.g.:

\texttt{"ldap://ldap.example.net/dc=example,dc=net;EXAMPLE"}

If \texttt{ldaps} is specified instead of \texttt{ldap}, TLS encryption will be enabled for the connection. Note that this will encrypt only the connection between the PostgreSQL server and the LDAP server. The connection between the client and the PostgreSQL server is not affected by this setting. To make use of TLS encryption, you might need to configure the LDAP library prior to configuring PostgreSQL. Note that encrypted LDAP is available only if the platform’s LDAP library supports it.

If no port is specified, the default port as configured in the LDAP library will be used.
Chapter 21. Client Authentication

The server will bind to the distinguished name specified as base dn using the user name supplied by the client. If prefix and suffix is specified, it will be prepended and appended to the user name before the bind. Typically, the prefix parameter is used to specify cn=, or \DOMAIN\ in an Active Directory environment.

21.2.8. PAM authentication

This authentication method operates similarly to password except that it uses PAM (Pluggable Authentication Modules) as the authentication mechanism. The default PAM service name is postgresql. You can optionally supply your own service name after the pam key word in the file pg_hba.conf. PAM is used only to validate user name/password pairs. Therefore the user must already exist in the database before PAM can be used for authentication. For more information about PAM, please read the Linux-PAM Page\(^4\) and the Solaris PAM Page\(^5\).

**Note:** If PAM is set up to read /etc/shadow, authentication will fail because the PostgreSQL server is started by a non-root user. However, this is not an issue with LDAP or other authentication methods.

21.3. Authentication problems

Genuine authentication failures and related problems generally manifest themselves through error messages like the following.

**FATAL:** no pg_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"

This is what you are most likely to get if you succeed in contacting the server, but it does not want to talk to you. As the message suggests, the server refused the connection request because it found no matching entry in its pg_hba.conf configuration file.

**FATAL:** Password authentication failed for user "andym"

Messages like this indicate that you contacted the server, and it is willing to talk to you, but not until you pass the authorization method specified in the pg_hba.conf file. Check the password you are providing, or check your Kerberos or ident software if the complaint mentions one of those authentication types.

**FATAL:** user "andym" does not exist

The indicated user name was not found.

**FATAL:** database "testdb" does not exist

The database you are trying to connect to does not exist. Note that if you do not specify a database name, it defaults to the database user name, which might or might not be the right thing.

\(^4\) http://www.kernel.org/pub/linux/libs/pam/
\(^5\) http://www.sun.com/software/solaris/pam/
Tip: The server log might contain more information about an authentication failure than is reported to the client. If you are confused about the reason for a failure, check the log.
Chapter 22. Localization

This chapter describes the available localization features from the point of view of the administrator. PostgreSQL supports localization with two approaches:

- Using the locale features of the operating system to provide locale-specific collation order, number formatting, translated messages, and other aspects.
- Providing a number of different character sets defined in the PostgreSQL server, including multi-byte character sets, to support storing text in all kinds of languages, and providing character set translation between client and server.

22.1. Locale Support

Locale support refers to an application respecting cultural preferences regarding alphabets, sorting, number formatting, etc. PostgreSQL uses the standard ISO C and POSIX locale facilities provided by the server operating system. For additional information refer to the documentation of your system.

22.1.1. Overview

Locale support is automatically initialized when a database cluster is created using `initdb`. `initdb` will initialize the database cluster with the locale setting of its execution environment by default, so if your system is already set to use the locale that you want in your database cluster then there is nothing else you need to do. If you want to use a different locale (or you are not sure which locale your system is set to), you can instruct `initdb` exactly which locale to use by specifying the `--locale` option. For example:

```
initdb --locale=sv_SE
```

This example sets the locale to Swedish (`sv`) as spoken in Sweden (`SE`). Other possibilities might be `en_US` (U.S. English) and `fr_CA` (French Canadian). If more than one character set can be useful for a locale then the specifications look like this: `cs_CZ.ISO8859-2`. What locales are available under what names on your system depends on what was provided by the operating system vendor and what was installed. (On most systems, the command `locale -a` will provide a list of available locales.)

Occasionally it is useful to mix rules from several locales, e.g., use English collation rules but Spanish messages. To support that, a set of locale subcategories exist that control only a certain aspect of the localization rules:

<table>
<thead>
<tr>
<th>LC_COLLATE</th>
<th>String sort order</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_CTYPE</td>
<td>Character classification (What is a letter? Its upper-case equivalent?)</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td>Language of messages</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>Formatting of currency amounts</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>Formatting of numbers</td>
</tr>
</tbody>
</table>
Formatting of dates and times

The category names translate into names of `initdb` options to override the locale choice for a specific category. For instance, to set the locale to French Canadian, but use U.S. rules for formatting currency, use `initdb --locale=fr_CA --lc-monetary=en_US`.

If you want the system to behave as if it had no locale support, use the special locale `C` or `POSIX`.

The nature of some locale categories is that their value has to be fixed for the lifetime of a database cluster. That is, once `initdb` has run, you cannot change them anymore. `LC_COLLATE` and `LC_CTYPE` are those categories. They affect the sort order of indexes, so they must be kept fixed, or indexes on text columns will become corrupt. PostgreSQL enforces this by recording the values of `LC_COLLATE` and `LC_CTYPE` that are seen by `initdb`. The server automatically adopts those two values when it is started.

The other locale categories can be changed as desired whenever the server is running by setting the run-time configuration variables that have the same name as the locale categories (see Section 18.10.2 for details). The defaults that are chosen by `initdb` are actually only written into the configuration file `postgresql.conf` to serve as defaults when the server is started. If you delete these assignments from `postgresql.conf` then the server will inherit the settings from its execution environment.

Note that the locale behavior of the server is determined by the environment variables seen by the server, not by the environment of any client. Therefore, be careful to configure the correct locale settings before starting the server. A consequence of this is that if client and server are set up in different locales, messages might appear in different languages depending on where they originated.

Note: When we speak of inheriting the locale from the execution environment, this means the following on most operating systems: For a given locale category, say the collation, the following environment variables are consulted in this order until one is found to be set: `LC_ALL`, `LC_COLLATE` (the variable corresponding to the respective category), `LANG`. If none of these environment variables are set then the locale defaults to `C`.

Some message localization libraries also look at the environment variable `LANGUAGE` which overrides all other locale settings for the purpose of setting the language of messages. If in doubt, please refer to the documentation of your operating system, in particular the documentation about `gettext`, for more information.

To enable messages to be translated to the user’s preferred language, NLS must have been enabled at build time. This choice is independent of the other locale support.

### 22.1.2. Behavior

The locale settings influence the following SQL features:

- Sort order in queries using `ORDER BY` on textual data
- The ability to use indexes with `LIKE` clauses
- The `upper`, `lower`, and `initcap` functions
- The `to_char` family of functions
The drawback of using locales other than C or POSIX in PostgreSQL is its performance impact. It slows character handling and prevents ordinary indexes from being used by LIKE. For this reason use locales only if you actually need them.

As a workaround to allow PostgreSQL to use indexes with LIKE clauses under a non-C locale, several custom operator classes exist. These allow the creation of an index that performs a strict character-by-character comparison, ignoring locale comparison rules. Refer to Section 11.9 for more information.

### 22.1.3. Problems

If locale support doesn’t work according to the explanation above, check that the locale support in your operating system is correctly configured. To check what locales are installed on your system, you can use the command `locale -a` if your operating system provides it.

Check that PostgreSQL is actually using the locale that you think it is. **LC_COLLATE** and **LC_CTYPE** settings are determined at initdb time and cannot be changed without repeating initdb. Other locale settings including **LC_MESSAGES** and **LC_MONETARY** are initially determined by the environment the server is started in, but can be changed on-the-fly. You can check the active locale settings using the `SHOW` command.

The directory `src/test/locale` in the source distribution contains a test suite for PostgreSQL’s locale support.

Client applications that handle server-side errors by parsing the text of the error message will obviously have problems when the server’s messages are in a different language. Authors of such applications are advised to make use of the error code scheme instead.

Maintaining catalogs of message translations requires the on-going efforts of many volunteers that want to see PostgreSQL speak their preferred language well. If messages in your language are currently not available or not fully translated, your assistance would be appreciated. If you want to help, refer to Chapter 47 or write to the developers’ mailing list.

### 22.2. Character Set Support

The character set support in PostgreSQL allows you to store text in a variety of character sets (also called encodings), including single-byte character sets such as the ISO 8859 series and multiple-byte character sets such as EUC (Extended Unix Code), UTF-8, and Mule internal code. All supported character sets can be used transparently by clients, but a few are not supported for use within the server (that is, as a server-side encoding). The default character set is selected while initializing your PostgreSQL database cluster using initdb. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

An important restriction, however, is that each database character set must be compatible with the server’s **LC_CTYPE** setting. When **LC_CTYPE** is C or POSIX, any character set is allowed, but for other settings of **LC_CTYPE** there is only one character set that will work correctly. Since the **LC_CTYPE** setting is frozen by initdb, the apparent flexibility to use different encodings in different databases of a cluster is more theoretical than real, except when you select C or POSIX locale (thus disabling any real locale awareness). It is likely that these mechanisms will be revisited in future versions of PostgreSQL.
# 22.2.1. Supported Character Sets

Table 22-1 shows the character sets available for use in PostgreSQL.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Language</th>
<th>Server?</th>
<th>Bytes/Char</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIG5</td>
<td>Big Five</td>
<td>Traditional Chinese</td>
<td>No</td>
<td>1-2</td>
<td>WIN950, Windows950</td>
</tr>
<tr>
<td>EUC_CN</td>
<td>Extended UNIX Code-CN</td>
<td>Simplified Chinese</td>
<td>Yes</td>
<td>1-3</td>
<td></td>
</tr>
<tr>
<td>EUC_TW</td>
<td>Extended UNIX Code-TW</td>
<td>Traditional Chinese, Taiwanese</td>
<td>Yes</td>
<td>1-3</td>
<td></td>
</tr>
<tr>
<td>GB18030</td>
<td>National Standard</td>
<td>Chinese</td>
<td>No</td>
<td>1-2</td>
<td></td>
</tr>
<tr>
<td>GBK</td>
<td>Extended National Standard</td>
<td>Simplified Chinese</td>
<td>No</td>
<td>1-2</td>
<td>WIN936, Windows936</td>
</tr>
<tr>
<td>ISO_8859_5</td>
<td>ISO 8859-5, ECMA 113</td>
<td>Latin/Cyrillic</td>
<td>Yes</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ISO_8859_6</td>
<td>ISO 8859-6, ECMA 114</td>
<td>Latin/Arabic</td>
<td>Yes</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ISO_8859_7</td>
<td>ISO 8859-7, ECMA 118</td>
<td>Latin/Greek</td>
<td>Yes</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ISO_8859_8</td>
<td>ISO 8859-8, ECMA 121</td>
<td>Latin/Hebrew</td>
<td>Yes</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JOHAB</td>
<td>JOHAB</td>
<td>Korean (Hangul)</td>
<td>No</td>
<td>1-3</td>
<td></td>
</tr>
<tr>
<td>KOI8</td>
<td>KOI8-R(U)</td>
<td>Cyrillic</td>
<td>Yes</td>
<td>1</td>
<td>KOI8R</td>
</tr>
<tr>
<td>LATIN1</td>
<td>ISO 8859-1, ECMA 94</td>
<td>Western European</td>
<td>Yes</td>
<td>1</td>
<td>ISO88591</td>
</tr>
<tr>
<td>LATIN2</td>
<td>ISO 8859-2, ECMA 94</td>
<td>Central European</td>
<td>Yes</td>
<td>1</td>
<td>ISO88592</td>
</tr>
<tr>
<td>LATIN3</td>
<td>ISO 8859-3, ECMA 94</td>
<td>South European</td>
<td>Yes</td>
<td>1</td>
<td>ISO88593</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Language</td>
<td>Server?</td>
<td>Bytes/Char</td>
<td>Aliases</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>-------------------</td>
<td>---------</td>
<td>------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>LATIN4</td>
<td>ISO 8859-4, ECMA 94</td>
<td>North European</td>
<td>Yes</td>
<td>1</td>
<td>ISO88594</td>
</tr>
<tr>
<td>LATIN5</td>
<td>ISO 8859-9, ECMA 128</td>
<td>Turkish</td>
<td>Yes</td>
<td>1</td>
<td>ISO88599</td>
</tr>
<tr>
<td>LATIN6</td>
<td>ISO 8859-10, ECMA 144</td>
<td>Nordic</td>
<td>Yes</td>
<td>1</td>
<td>ISO885910</td>
</tr>
<tr>
<td>LATIN7</td>
<td>ISO 8859-13</td>
<td>Baltic</td>
<td>Yes</td>
<td>1</td>
<td>ISO885913</td>
</tr>
<tr>
<td>LATIN8</td>
<td>ISO 8859-14</td>
<td>Celtic</td>
<td>Yes</td>
<td>1</td>
<td>ISO885914</td>
</tr>
<tr>
<td>LATIN9</td>
<td>ISO 8859-15</td>
<td>LATIN1 with Euro and accents</td>
<td>Yes</td>
<td>1</td>
<td>ISO885915</td>
</tr>
<tr>
<td>LATIN10</td>
<td>ISO 8859-16, ASRO SR 14111</td>
<td>Romanian</td>
<td>Yes</td>
<td>1</td>
<td>ISO885916</td>
</tr>
<tr>
<td>MULE_INTERNAL</td>
<td>Mule internal code</td>
<td>Multilingual Emacs</td>
<td>Yes</td>
<td>1-4</td>
<td>Mskanji, ShiftJIS, WIN932, Windows932</td>
</tr>
<tr>
<td>SJIS</td>
<td>Shift JIS</td>
<td>Japanese</td>
<td>No</td>
<td>1-2</td>
<td>WIN949, Windows949</td>
</tr>
<tr>
<td>SHIFT_JIS_2004</td>
<td>Shift JIS, JIS X 0213</td>
<td>Japanese</td>
<td>No</td>
<td>1-2</td>
<td>WIN949, Windows949</td>
</tr>
<tr>
<td>SQL.ASCII</td>
<td>unspecified (see text)</td>
<td>any</td>
<td>Yes</td>
<td>1</td>
<td>WIN949, Windows949</td>
</tr>
<tr>
<td>UHC</td>
<td>Unified Hangul Code</td>
<td>Korean</td>
<td>No</td>
<td>1-2</td>
<td>WIN949, Windows949</td>
</tr>
<tr>
<td>UTF8</td>
<td>Unicode, 8-bit</td>
<td>all</td>
<td>Yes</td>
<td>1-4</td>
<td>Unicode</td>
</tr>
<tr>
<td>WIN866</td>
<td>Windows CP866</td>
<td>Cyrillic</td>
<td>Yes</td>
<td>1</td>
<td>ALT</td>
</tr>
<tr>
<td>WIN874</td>
<td>Windows CP874</td>
<td>Thai</td>
<td>Yes</td>
<td>1</td>
<td>WIN</td>
</tr>
<tr>
<td>WIN1250</td>
<td>Windows CP1250</td>
<td>Central European</td>
<td>Yes</td>
<td>1</td>
<td>WIN</td>
</tr>
<tr>
<td>WIN1251</td>
<td>Windows CP1251</td>
<td>Cyrillic</td>
<td>Yes</td>
<td>1</td>
<td>WIN</td>
</tr>
<tr>
<td>WIN1252</td>
<td>Windows CP1252</td>
<td>Western European</td>
<td>Yes</td>
<td>1</td>
<td>WIN</td>
</tr>
<tr>
<td>WIN1253</td>
<td>Windows CP1253</td>
<td>Greek</td>
<td>Yes</td>
<td>1</td>
<td>WIN</td>
</tr>
<tr>
<td>WIN1254</td>
<td>Windows CP1254</td>
<td>Turkish</td>
<td>Yes</td>
<td>1</td>
<td>WIN</td>
</tr>
<tr>
<td>WIN1255</td>
<td>Windows CP1255</td>
<td>Hebrew</td>
<td>Yes</td>
<td>1</td>
<td>WIN</td>
</tr>
<tr>
<td>WIN1256</td>
<td>Windows CP1256</td>
<td>Arabic</td>
<td>Yes</td>
<td>1</td>
<td>WIN</td>
</tr>
</tbody>
</table>
Chapter 22. Localization

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Language</th>
<th>Server?</th>
<th>Bytes/Char</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIN1257</td>
<td>Windows CP1257</td>
<td>Baltic</td>
<td>Yes</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>WIN1258</td>
<td>Windows CP1258</td>
<td>Vietnamese</td>
<td>Yes</td>
<td>1</td>
<td>ABC, TCVN, TCVN5712, VSCII</td>
</tr>
</tbody>
</table>

Not all APIs support all the listed character sets. For example, the PostgreSQL JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.

The SQL_ASCII setting behaves considerably differently from the other settings. When the server character set is SQL_ASCII, the server interprets byte values 0-127 according to the ASCII standard, while byte values 128-255 are taken as uninterpreted characters. No encoding conversion will be done when the setting is SQL_ASCII. Thus, this setting is not so much a declaration that a specific encoding is in use, as a declaration of ignorance about the encoding. In most cases, if you are working with any non-ASCII data, it is unwise to use the SQL_ASCII setting, because PostgreSQL will be unable to help you by converting or validating non-ASCII characters.

22.2.2. Setting the Character Set

initdb defines the default character set for a PostgreSQL cluster. For example,

```bash
initdb -E EUC_JP
```

sets the default character set (encoding) to EUC_JP (Extended Unix Code for Japanese). You can use --encoding instead of -E if you prefer to type longer option strings. If no -E or --encoding option is given, initdb attempts to determine the appropriate encoding to use based on the specified or default locale.

If you have selected C or POSIX locale, you can create a database with a different character set:

```bash
createdb -E EUC_KR korean
```

This will create a database named korean that uses the character set EUC_KR. Another way to accomplish this is to use this SQL command:

```sql
CREATE DATABASE korean WITH ENCODING 'EUC_KR';
```

The encoding for a database is stored in the system catalog pg_database. You can see it by using the \l option or the \l command of psql.

```bash
$ psql -l
```

List of databases

<table>
<thead>
<tr>
<th>Database</th>
<th>Owner</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>euc_cn</td>
<td>t-ishii</td>
<td>EUC_CN</td>
</tr>
<tr>
<td>euc_jp</td>
<td>t-ishii</td>
<td>EUC_JP</td>
</tr>
<tr>
<td>euc_kr</td>
<td>t-ishii</td>
<td>EUC_KR</td>
</tr>
<tr>
<td>euc_tw</td>
<td>t-ishii</td>
<td>EUC_TW</td>
</tr>
<tr>
<td>mule_internal</td>
<td>t-ishii</td>
<td>MULE_INTERNAL</td>
</tr>
<tr>
<td>postgres</td>
<td>t-ishii</td>
<td>EUC_JP</td>
</tr>
<tr>
<td>regression</td>
<td>t-ishii</td>
<td>SQL_ASCII</td>
</tr>
<tr>
<td>template1</td>
<td>t-ishii</td>
<td>EUC_JP</td>
</tr>
<tr>
<td>test</td>
<td>t-ishii</td>
<td>EUC_JP</td>
</tr>
<tr>
<td>utf8</td>
<td>t-ishii</td>
<td>UTF8</td>
</tr>
</tbody>
</table>

415
Important: On most modern operating systems, PostgreSQL can determine which character set is implied by an `LC_CTYPE` setting, and it will enforce that only the correct database encoding is used. On older systems it is your responsibility to ensure that you use the encoding expected by the locale you have selected. A mistake in this area is likely to lead to strange misbehavior of locale-dependent operations such as sorting.

PostgreSQL will allow superusers to create databases with `SQL_ASCII` encoding even when `LC_CTYPE` is not `C` or `POSIX`. As noted above, `SQL_ASCII` does not enforce that the data stored in the database has any particular encoding, and so this choice poses risks of locale-dependent misbehavior. Using this combination of settings is deprecated and may someday be forbidden altogether.

### 22.2.3. Automatic Character Set Conversion Between Server and Client

PostgreSQL supports automatic character set conversion between server and client for certain character set combinations. The conversion information is stored in the `pg_conversion` system catalog. PostgreSQL comes with some predefined conversions, as shown in Table 22-2. You can create a new conversion using the SQL command `CREATE CONVERSION`.

<table>
<thead>
<tr>
<th>Server Character Set</th>
<th>Available Client Character Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIG5</td>
<td><code>not supported as a server encoding</code></td>
</tr>
<tr>
<td>EUC_CN</td>
<td><code>EUC_CN, MULE_INTERNAL, UTF8</code></td>
</tr>
<tr>
<td>EUC_JP</td>
<td><code>EUC_JP, MULE_INTERNAL, SJIS, UTF8</code></td>
</tr>
<tr>
<td>EUC_KR</td>
<td><code>EUC_KR, MULE_INTERNAL, UTF8</code></td>
</tr>
<tr>
<td>EUC_TW</td>
<td><code>EUC_TW, BIG5, MULE_INTERNAL, UTF8</code></td>
</tr>
<tr>
<td>GB18030</td>
<td><code>not supported as a server encoding</code></td>
</tr>
<tr>
<td>GBK</td>
<td><code>not supported as a server encoding</code></td>
</tr>
<tr>
<td>ISO_8859_5</td>
<td><code>ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866, WIN1251</code></td>
</tr>
<tr>
<td>ISO_8859_6</td>
<td><code>ISO_8859_6, UTF8</code></td>
</tr>
<tr>
<td>ISO_8859_7</td>
<td><code>ISO_8859_7, UTF8</code></td>
</tr>
<tr>
<td>ISO_8859_8</td>
<td><code>ISO_8859_8, UTF8</code></td>
</tr>
<tr>
<td>JOHAB</td>
<td><code>JOHAB, UTF8</code></td>
</tr>
<tr>
<td>KOI8</td>
<td><code>KOI8, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251</code></td>
</tr>
<tr>
<td>LATIN1</td>
<td><code>LATIN1, MULE_INTERNAL, UTF8</code></td>
</tr>
<tr>
<td>LATIN2</td>
<td><code>LATIN2, MULE_INTERNAL, UTF8, WIN1250</code></td>
</tr>
<tr>
<td>LATIN3</td>
<td><code>LATIN3, MULE_INTERNAL, UTF8</code></td>
</tr>
<tr>
<td>LATIN4</td>
<td><code>LATIN4, MULE_INTERNAL, UTF8</code></td>
</tr>
</tbody>
</table>
### Server Character Set

<table>
<thead>
<tr>
<th>Server Character Set</th>
<th>Available Client Character Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATIN5</td>
<td>LATIN5, UTF8</td>
</tr>
<tr>
<td>LATIN6</td>
<td>LATIN6, UTF8</td>
</tr>
<tr>
<td>LATIN7</td>
<td>LATIN7, UTF8</td>
</tr>
<tr>
<td>LATIN8</td>
<td>LATIN8, UTF8</td>
</tr>
<tr>
<td>LATIN9</td>
<td>LATIN9, UTF8</td>
</tr>
<tr>
<td>LATIN10</td>
<td>LATIN10, UTF8</td>
</tr>
<tr>
<td>MULE_INTERNAL</td>
<td>MULE_INTERNAL, EUC_CN, EUC_JP,</td>
</tr>
<tr>
<td></td>
<td>EUC_KR, ISO_8859_5, KOI8, LATIN1</td>
</tr>
<tr>
<td></td>
<td>to LATIN4, SJIS, WIN866, WIN1250, WIN1251</td>
</tr>
<tr>
<td>SJIS</td>
<td>not supported as a server encoding</td>
</tr>
<tr>
<td>SQL_ASCII</td>
<td>any (no conversion will be performed)</td>
</tr>
<tr>
<td>UHC</td>
<td>not supported as a server encoding</td>
</tr>
<tr>
<td>UTF8</td>
<td>all supported encodings</td>
</tr>
<tr>
<td>WIN866</td>
<td>WIN866, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN1251</td>
</tr>
<tr>
<td>WIN874</td>
<td>WIN874, UTF8</td>
</tr>
<tr>
<td>WIN1250</td>
<td>WIN1250, LATIN2, MULE_INTERNAL, UTF8</td>
</tr>
<tr>
<td>WIN1251</td>
<td>WIN1251, ISO_8859_5, KOI8, MULE_INTERNAL, UTF8, WIN866</td>
</tr>
<tr>
<td>WIN1252</td>
<td>WIN1252, UTF8</td>
</tr>
<tr>
<td>WIN1253</td>
<td>WIN1253, UTF8</td>
</tr>
<tr>
<td>WIN1254</td>
<td>WIN1254, UTF8</td>
</tr>
<tr>
<td>WIN1255</td>
<td>WIN1255, UTF8</td>
</tr>
<tr>
<td>WIN1256</td>
<td>WIN1256, UTF8</td>
</tr>
<tr>
<td>WIN1257</td>
<td>WIN1257, UTF8</td>
</tr>
<tr>
<td>WIN1258</td>
<td>WIN1258, UTF8</td>
</tr>
</tbody>
</table>

To enable automatic character set conversion, you have to tell PostgreSQL the character set (encoding) you would like to use in the client. There are several ways to accomplish this:

- **Using the \encoding command in psql.** \encoding allows you to change client encoding on the fly. For example, to change the encoding to SJIS, type:

  \encoding SJIS

- **libpq (Section 30.9) has functions to control the client encoding.**

- **Using SET client_encoding TO.** Setting the client encoding can be done with this SQL command:

  ```sql
  SET CLIENT_ENCODING TO 'value';
  ``

  Also you can use the standard SQL syntax SET NAMES for this purpose:

  ```sql
  SET NAMES 'value';
  ``

  To query the current client encoding:

  ```sql
  SHOW client_encoding;
  ``

  To return to the default encoding:
RESET client_encoding;

- **Using PGCLIENTENCODING.** If the environment variable `PGCLIENTENCODING` is defined in the client’s environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)

- **Using the configuration variable client_encoding.** If the `client_encoding` variable is set, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)

If the conversion of a particular character is not possible — suppose you chose `EUC_JP` for the server and `LATIN1` for the client, then some Japanese characters do not have a representation in `LATIN1` — then an error is reported.

If the client character set is defined as `SQL_ASCII`, encoding conversion is disabled, regardless of the server’s character set. Just as for the server, use of `SQL_ASCII` is unwise unless you are working with all-ASCII data.

### 22.2.4. Further Reading

These are good sources to start learning about various kinds of encoding systems.

[http://www.i18ngurus.com/docs/984813247.html](http://www.i18ngurus.com/docs/984813247.html)

An extensive collection of documents about character sets, encodings, and code pages.


[http://www.unicode.org/](http://www.unicode.org/)

The web site of the Unicode Consortium

RFC 3629

UTF-8 is defined here.
Chapter 23. Routine Database Maintenance Tasks

PostgreSQL, like any database software, requires that certain tasks be performed regularly to achieve optimum performance. The tasks discussed here are required, but they are repetitive in nature and can easily be automated using standard Unix tools such as cron scripts or Windows’ Task Scheduler. But it is the database administrator’s responsibility to set up appropriate scripts, and to check that they execute successfully.

One obvious maintenance task is creation of backup copies of the data on a regular schedule. Without a recent backup, you have no chance of recovery after a catastrophe (disk failure, fire, mistakenly dropping a critical table, etc.). The backup and recovery mechanisms available in PostgreSQL are discussed at length in Chapter 24.

The other main category of maintenance task is periodic “vacuuming” of the database. This activity is discussed in Section 23.1. Closely related to this is updating the statistics that will be used by the query planner, as discussed in Section 23.1.2.

Another task that might need periodic attention is log file management. This is discussed in Section 23.3.

PostgreSQL is low-maintenance compared to some other database management systems. Nonetheless, appropriate attention to these tasks will go far towards ensuring a pleasant and productive experience with the system.

23.1. Routine Vacuuming

PostgreSQL’s VACUUM command has to run on a regular basis for several reasons:

1. To recover or reuse disk space occupied by updated or deleted rows.
2. To update data statistics used by the PostgreSQL query planner.
3. To protect against loss of very old data due to transaction ID wraparound.

The standard form of VACUUM can run in parallel with production database operations. Commands such as SELECT, INSERT, UPDATE, and DELETE will continue to function as normal, though you will not be able to modify the definition of a table with commands such as ALTER TABLE ADD COLUMN while it is being vacuumed. Also, VACUUM requires a substantial amount of I/O traffic, which can cause poor performance for other active sessions. There are configuration parameters that can be adjusted to reduce the performance impact of background vacuuming — see Section 18.4.4.

Fortunately, The Auto-Vacuum Daemon monitors table activity and performs VACUUMs when necessary. Autovacuum works dynamically so it is often better administration-scheduled vacuuming.

23.1.1. Recovering Disk Space

In normal PostgreSQL operation, an UPDATE or DELETE of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multiversion concurrency control (see Chapter 13): the row versions must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any
transaction. The space it occupies must be reclaimed for reuse by new rows, to avoid infinite growth of disk space requirements. This is done by running \texttt{VACUUM}.

There are two variants of the \texttt{VACUUM} command. The first form, known as “lazy vacuum” or just \texttt{VACUUM}, marks dead data in tables and indexes for future reuse; it does \textit{not} attempt to reclaim the space used by this dead data unless the space is at the end of the table and an exclusive table lock can be easily obtained. Unused space at the start or middle of the file does not result in the file being shortened and space returned to the operating system. This variant of \texttt{VACUUM} can be run concurrently with normal database operations.

The second form is the \texttt{VACUUM FULL} command. This uses a more aggressive algorithm for reclaiming the space consumed by dead row versions. Any space that is freed by \texttt{VACUUM FULL} is immediately returned to the operating system, and the table data is physically compacted on the disk. Unfortunately, this variant of the \texttt{VACUUM} command acquires an exclusive lock on each table while \texttt{VACUUM FULL} is processing it. Therefore, frequently using \texttt{VACUUM FULL} can have an extremely negative effect on the performance of concurrent database queries.

Fortunately, \textit{The Auto-Vacuum Daemon} monitors table activity and performs \texttt{VACUUM}s when necessary. This eliminates the need for administrators to worry about disk space recovery in all but the most unusual cases.

For administrators who want to control \texttt{VACUUM} themselves, the standard form of \texttt{VACUUM} is best used to maintain a steady-state usage of disk space. If you need to return disk space to the operating system, you can use \texttt{VACUUM FULL}, but this is unwise if the table will just grow again in the future. Moderately-frequent standard \texttt{VACUUM} runs are a better approach than infrequent \texttt{VACUUM FULL} runs for maintaining heavily-updated tables. However, if some heavily-updated tables have gone too long with infrequent \texttt{VACUUM}, you can use \texttt{VACUUM FULL} or \texttt{CLUSTER} to get performance back (it is much slower to scan a table containing almost only dead rows).

For those not using autovacuum, one approach is to schedule a database-wide \texttt{VACUUM} once a day during low-usage period, supplemented by more frequent vacuuming of heavily-updated tables if necessary. (Some installations with extremely high update rates vacuum their busiest tables as often as once every few minutes.) If you have multiple databases in a cluster, don’t forget to \texttt{VACUUM} each one; the program \texttt{vacuumdb} might be helpful.

\texttt{VACUUM FULL} is recommended for cases where you know you have deleted the majority of rows in a table, so that the steady-state size of the table can be shrunk substantially with \texttt{VACUUM FULL}’s more aggressive approach. Use plain \texttt{VACUUM}, not \texttt{VACUUM FULL}, for routine vacuuming for space recovery.

If you have a table whose entire contents are deleted on a periodic basis, consider doing it with \texttt{TRUNCATE} rather than using \texttt{DELETE} followed by \texttt{VACUUM}. \texttt{TRUNCATE} removes the entire content of the table immediately, without requiring a subsequent \texttt{VACUUM} or \texttt{VACUUM FULL} to reclaim the now-unused disk space.

\subsection*{23.1.2. Updating Planner Statistics}

The PostgreSQL query planner relies on statistical information about the contents of tables in order to generate good plans for queries. These statistics are gathered by the \texttt{ANALYZE} command, which can be invoked by itself or as an optional step in \texttt{VACUUM}. It is important to have reasonably accurate statistics, otherwise poor choices of plans might degrade database performance.

As with vacuuming for space recovery, frequent updates of statistics are more useful for heavily-updated tables than for seldom-updated ones. But even for a heavily-updated table, there might be no need for statistics updates if the statistical distribution of the data is not changing much. A simple rule of thumb is to think about how much the minimum and maximum values of the columns in the
Chapter 23. Routine Database Maintenance Tasks

table change. For example, a timestamp column that contains the time of row update will have a constantly-increasing maximum value as rows are added and updated; such a column will probably need more frequent statistics updates than, say, a column containing URLs for pages accessed on a website. The URL column might receive changes just as often, but the statistical distribution of its values probably changes relatively slowly.

It is possible to run `ANALYZE` on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice, however, it is usually best to just analyze the entire database because it is a fast operation. It uses a statistical random sampling of the rows of a table rather than reading every single row.

**Tip:** Although per-column tweaking of `ANALYZE` frequency might not be very productive, you might well find it worthwhile to do per-column adjustment of the level of detail of the statistics collected by `ANALYZE`. Columns that are heavily used in `WHERE` clauses and have highly irregular data distributions might require a finer-grain data histogram than other columns. See `ALTER TABLE SET STATISTICS`.

Fortunately, *The Auto-Vacuum Daemon* monitors table activity and performs `ANALYZE` when necessary. This eliminates the need for administrators to manually schedule `ANALYZE`.

For those not using autovacuum, one approach is to schedule a database-wide `ANALYZE` once a day at a low-usage time of day; this can usefully be combined with a nightly `VACUUM`. However, sites with relatively slowly changing table statistics might find that this is overkill, and that less-frequent `ANALYZE` runs are sufficient.

### 23.1.3. Preventing Transaction ID Wraparound Failures

PostgreSQL’s MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction’s XID is “in the future” and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits at this writing) a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their outputs become invisible. In short, catastrophic data loss. (Actually the data is still there, but that’s cold comfort if you cannot get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.

The reason that periodic vacuuming solves the problem is that PostgreSQL distinguishes a special XID FrozenXID. This XID is always considered older than every normal XID. Normal XIDs are compared using modulo-$2^{31}$ arithmetic. This means that for every normal XID, there are two billion XIDs that are “older” and two billion that are “newer”; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a row version has been created with a particular normal XID, the row version will appear to be “in the past” for the next two billion transactions, no matter which normal XID we are talking about. If the row version still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent data loss, old row versions must be reassigned the XID FrozenXID sometime before they reach the two-billion-transactions-old mark. Once they are assigned this special XID, they will appear to be “in the past” to all normal transactions regardless of wraparound issues, and so such row versions will be good until deleted, no matter how long that is. This reassignment of old XIDs is handled by `VACUUM`.

`VACUUM`'s behavior is controlled by the configuration parameter `vacuum_freeze_min_age`: any XID older than `vacuum_freeze_min_age` transactions is replaced by FrozenXID. Larger values of
Chapter 23. Routine Database Maintenance Tasks

vacuum_freeze_min_age preserves transactional information longer, while smaller values increase the number of transactions that can elapse before the table must be vacuumed again.

The maximum time that a table can go unvacuumed is two billion transactions minus the vacuum_freeze_min_age that was used when it was last vacuumed. If it were to go unvacuumed for longer than that, data loss could result. To ensure that this does not happen, the Auto-Vacuum Daemon is invoked on any table that might contain XIDs older than the age specified by the configuration parameter autovacuum_freeze_max_age. (This will happen even if autovacuum is otherwise disabled.)

This implies that if a table is not otherwise vacuumed, autovacuum will be invoked on it approximately once every autovacuum_freeze_max_age minus vacuum_freeze_min_age transactions. For tables that are regularly vacuumed for space reclamation purposes, this is of little importance. However, for static tables (including tables that receive inserts, but no updates or deletes), there is no need for vacuuming for space reclamation, and so it can be useful to try to maximize the interval between forced autovacuums on very large static tables. Obviously one can do this either by increasing autovacuum_freeze_max_age or by decreasing vacuum_freeze_min_age.

The sole disadvantage of increasing autovacuum_freeze_max_age is that the pg_clog subdirectory of the database cluster will take more space, because it must store the commit status for all transactions back to the autovacuum_freeze_max_age horizon. The commit status uses two bits per transaction, so if autovacuum_freeze_max_age has its maximum allowed value of two billion, pg_clog can be expected to grow to about half a gigabyte. If this is trivial compared to your total database size, setting autovacuum_freeze_max_age to its maximum allowed value is recommended. Otherwise, set it depending on what you are willing to allow for pg_clog storage. (The default, 200 million transactions, translates to about 50MB of pg_clog storage.)

One disadvantage of decreasing vacuum_freeze_min_age is that it might cause VACUUM to do useless work: changing a table row’s XID to FrozenXID is a waste of time if the row is modified soon thereafter (causing it to acquire a new XID). So the setting should be large enough that rows are not frozen until they are unlikely to change any more. Another disadvantage of decreasing this setting is that details about exactly which transaction inserted or modified a row will be lost sooner. This information sometimes comes in handy, particularly when trying to analyze what went wrong after a database failure. For these two reasons, decreasing this setting is not recommended except for completely static tables.

To track the age of the oldest XIDs in a database, VACUUM stores XID statistics in the system tables pg_class and pg_database. In particular, the relfrozenxid column of a table’s pg_class row contains the freeze cutoff XID that was used by the last VACUUM for that table. All normal XIDs older than this cutoff XID are guaranteed to have been replaced by FrozenXID within the table. Similarly, the datfrozenxid column of a database’s pg_database row is a lower bound on the normal XIDs appearing in that database — it is just the minimum of the per-table relfrozenxid values within the database. A convenient way to examine this information is to execute queries such as:

```sql
SELECT relname, age(relfrozenxid) FROM pg_class WHERE relkind = 'r';
SELECT datname, age(datfrozenxid) FROM pg_database;
```

The age column measures the number of transactions from the cutoff XID to the current transaction’s XID. Immediately after a VACUUM, age(relfrozenxid) should be a little more than the vacuum_freeze_min_age setting that was used (more by the number of transactions started since the VACUUM started). If age(relfrozenxid) exceeds autovacuum_freeze_max_age, an autovacuum will soon be forced for the table.

If for some reason autovacuum fails to clear old XIDs from a table, the system will begin to emit warning messages like this when the database’s oldest XIDs reach ten million transactions from the wraparound point:
Chapter 23. Routine Database Maintenance Tasks

WARNING: database "mydb" must be vacuumed within 177009986 transactions
HINT: To avoid a database shutdown, execute a full-database VACUUM in "mydb".

If these warnings are ignored, the system will shut down and refuse to execute any new transactions once there are fewer than 1 million transactions left until wraparound:

ERROR: database is not accepting commands to avoid wraparound data loss in database "mydb"
HINT: Stop the postmaster and use a standalone backend to VACUUM in "mydb".

The 1-million-transaction safety margin exists to let the administrator recover without data loss, by manually executing the required VACUUM commands. However, since the system will not execute commands once it has gone into the safety shutdown mode, the only way to do this is to stop the server and use a single-user backend to execute VACUUM. The shutdown mode is not enforced by a single-user backend. See the postgres reference page for details about using a single-user backend.

23.1.4. The Auto-Vacuum Daemon

Beginning in PostgreSQL 8.1, there is an optional feature called autovacuum, whose purpose is to automate the execution of VACUUM and ANALYZE commands. When enabled, autovacuum checks for tables that have had a large number of inserted, updated or deleted tuples. These checks use the statistics collection facility; therefore, autovacuum cannot be used unless track_counts is set to true.

In the default configuration, autovacuuming is enabled and the related configuration parameters are appropriately set.

Beginning in PostgreSQL 8.3, autovacuum has a multiprocess architecture: There is a daemon process, called the autovacuum launcher, which is in charge of starting autovacuum worker processes for all databases. The launcher will distribute the work across time, but attempt to start one worker on each database every autovacuum_naptime seconds. One worker will be launched for each database, with a maximum of autovacuum_max_workers processes running at the same time. If there are more than autovacuum_max_workers databases to be processed, the next database will be processed as soon as the first worker finishes. The worker processes will check each table within its database and execute VACUUM and/or ANALYZE as needed.

The autovacuum_max_workers setting limits how many workers may be running at any time. If several large tables all become eligible for vacuuming in a short amount of time, all autovacuum workers may end up vacuuming those tables for a very long time. This would result in other tables and databases not being vacuumed until a worker became available. There is not a limit on how many workers might be in a single database, but workers do try to avoid repeating work that has already been done by other workers. Note that the number of running workers does not count towards the max_connections nor the superuser_reserved_connections limits.

Tables whose relfrozenxid value is more than autovacuum_freeze_max_age transactions old are always vacuumed. Otherwise, two conditions are used to determine which operation(s) to apply.

If the number of obsolete tuples since the last VACUUM exceeds the “vacuum threshold”, the table is vacuumed. The vacuum threshold is defined as:

\[
\text{vacuum threshold} = \text{vacuum base threshold} + \text{vacuum scale factor} \times \text{number of tuples}
\]

where the vacuum base threshold is autovacuum_vacuum_threshold, the vacuum scale factor is autovacuum_vacuum_scale_factor, and the number of tuples is pg_class.reltuples. The number of obsolete tuples is obtained from the statistics collector; it is a semi-accurate count updated by each UPDATE and DELETE operation. (It is only semi-accurate because some information might be lost under heavy load.) For analyze, a similar condition is used: the threshold, defined as:
analyze threshold = analyze base threshold + analyze scale factor * number of tuples

is compared to the total number of tuples inserted or updated since the last ANALYZE.

The default thresholds and scale factors are taken from postgresql.conf, but it is possible to override them on a table-by-table basis by making entries in the system catalog pg_autovacuum. If a pg_autovacuum row exists for a particular table, the settings it specifies are applied; otherwise the global settings are used. See Section 18.9 for more details on the global settings.

Besides the base threshold values and scale factors, there are five more parameters that can be set for each table in pg_autovacuum. The first, pg_autovacuum.enabled, can be set to false to instruct the autovacuum daemon to skip that particular table entirely. In this case autovacuum will only touch the table if it must do so to prevent transaction ID wraparound. The next two parameters, the vacuum cost delay (pg_autovacuum.vac_cost_delay) and the vacuum cost limit (pg_autovacuum.vac_cost_limit), are used to set table-specific values for the Cost-Based Vacuum Delay feature. The last two parameters, (pg_autovacuum.freeze_min_age) and (pg_autovacuum.freeze_max_age), are used to set table-specific values for vacuum_freeze_min_age and autovacuum_freeze_max_age respectively.

If any of the values in pg_autovacuum are set to a negative number, or if a row is not present at all in pg_autovacuum for any particular table, the corresponding values from postgresql.conf are used.

There is not currently any support for making pg_autovacuum entries, except by doing manual INSERTs into the catalog. This feature will be improved in future releases, and it is likely that the catalog definition will change.

Caution

The contents of the pg_autovacuum system catalog are currently not saved in database dumps created by the tools pg_dump and pg_dumpall. If you want to preserve them across a dump/reload cycle, make sure you dump the catalog manually.

When multiple workers are running, the cost limit is “balanced” among all the running workers, so that the total impact on the system is the same, regardless of the number of workers actually running.

23.2. Routine Reindexing

In some situations it is worthwhile to rebuild indexes periodically with the REINDEX command.

In PostgreSQL releases before 7.4, periodic reindexing was frequently necessary to avoid “index bloat”, due to lack of internal space reclamation in B-tree indexes. Any situation in which the range of index keys changed over time — for example, an index on timestamps in a table where old entries are eventually deleted — would result in bloat, because index pages for no-longer-needed portions of the key range were not reclaimed for re-use. Over time, the index size could become indefinitely much larger than the amount of useful data in it.

In PostgreSQL 7.4 and later, index pages that have become completely empty are reclaimed for re-use. There is still a possibility for inefficient use of space: if all but a few index keys on a page have been deleted, the page remains allocated. So a usage pattern in which all but a few keys in each range are eventually deleted will see poor use of space. For such usage patterns, periodic reindexing is recommended.
The potential for bloat in non-B-tree indexes has not been well characterized. It is a good idea to keep an eye on the index’s physical size when using any non-B-tree index type.

Also, for B-tree indexes a freshly-constructed index is somewhat faster to access than one that has been updated many times, because logically adjacent pages are usually also physically adjacent in a newly built index. (This consideration does not currently apply to non-B-tree indexes.) It might be worthwhile to reindex periodically just to improve access speed.

### 23.3. Log File Maintenance

It is a good idea to save the database server’s log output somewhere, rather than just routing it to /dev/null. The log output is invaluable when it comes time to diagnose problems. However, the log output tends to be voluminous (especially at higher debug levels) and you won’t want to save it indefinitely. You need to “rotate” the log files so that new log files are started and old ones removed after a reasonable period of time.

If you simply direct the stderr of postgres into a file, you will have log output, but the only way to truncate the log file is to stop and restart the server. This might be OK if you are using PostgreSQL in a development environment, but few production servers would find this behavior acceptable.

A better approach is to send the server’s stderr output to some type of log rotation program. There is a built-in log rotation program, which you can use by setting the configuration parameter `logging_collector` to `true` in `postgresql.conf`. The control parameters for this program are described in Section 18.7.1. You can also use this approach to capture the log data in machine readable CSV format.

Alternatively, you might prefer to use an external log rotation program, if you have one that you are already using with other server software. For example, the rotatelogs tool included in the Apache distribution can be used with PostgreSQL. To do this, just pipe the server’s stderr output to the desired program. If you start the server with `pg_ctl`, then stderr is already redirected to stdout, so you just need a pipe command, for example:

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

Another production-grade approach to managing log output is to send it all to syslog and let syslog deal with file rotation. To do this, set the configuration parameter `log_destination` to `syslog` (to log to syslog only) in `postgresql.conf`. Then you can send a SIGHUP signal to the syslog daemon whenever you want to force it to start writing a new log file. If you want to automate log rotation, the logrotate program can be configured to work with log files from syslog.

On many systems, however, syslog is not very reliable, particularly with large log messages; it might truncate or drop messages just when you need them the most. Also, on Linux, syslog will sync each message to disk, yielding poor performance. (You can use a − at the start of the file name in the syslog configuration file to disable this behavior.)

Note that all the solutions described above take care of starting new log files at configurable intervals, but they do not handle deletion of old, no-longer-interesting log files. You will probably want to set up a batch job to periodically delete old log files. Another possibility is to configure the rotation program so that old log files are overwritten cyclically.

---

425
Chapter 24. Backup and Restore

As with everything that contains valuable data, PostgreSQL databases should be backed up regularly. While the procedure is essentially simple, it is important to have a clear understanding of the underlying techniques and assumptions.

There are three fundamentally different approaches to backing up PostgreSQL data:

- SQL dump
- File system level backup
- Continuous archiving

Each has its own strengths and weaknesses. Each is discussed in turn below.

24.1. SQL Dump

The idea behind this dump method is to generate a text file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump. PostgreSQL provides the utility program `pg_dump` for this purpose. The basic usage of this command is:

```
pg_dump dbname > outfile
```

As you see, `pg_dump` writes its results to the standard output. We will see below how this can be useful.

`pg_dump` is a regular PostgreSQL client application (albeit a particularly clever one). This means that you can do this backup procedure from any remote host that has access to the database. But remember that `pg_dump` does not operate with special permissions. In particular, it must have read access to all tables that you want to back up, so in practice you almost always have to run it as a database superuser.

To specify which database server `pg_dump` should contact, use the command line options `-h host` and `-p port`. The default host is the local host or whatever your `PGHOST` environment variable specifies. Similarly, the default port is indicated by the `PGPORT` environment variable or, failing that, by the compiled-in default. (Conveniently, the server will normally have the same compiled-in default.)

Like any other PostgreSQL client application, `pg_dump` will by default connect with the database user name that is equal to the current operating system user name. To override this, either specify the `-U` option or set the environment variable `PGUSER`. Remember that `pg_dump` connections are subject to the normal client authentication mechanisms (which are described in Chapter 21).

Dumps created by `pg_dump` are internally consistent, that is, the dump represents a snapshot of the database as of the time `pg_dump` begins running. `pg_dump` does not block other operations on the database while it is working. (Exceptions are those operations that need to operate with an exclusive lock, such as most forms of `ALTER TABLE`.)

**Important:** If your database schema relies on OIDs (for instance as foreign keys) you must instruct `pg_dump` to dump the OIDs as well. To do this, use the `-o` command line option.
24.1.1. Restoring the dump

The text files created by pg_dump are intended to be read in by the psql program. The general command form to restore a dump is

```plaintext
psql dbname < infile
```

where `infile` is what you used as `outfile` for the pg_dump command. The database `dbname` will not be created by this command, so you must create it yourself from `template0` before executing psql (e.g., with `createdb -T template0 dbname`). psql supports options similar to pg_dump’s for specifying the database server to connect to and the user name to use. See the psql reference page for more information.

Before restoring a SQL dump, all the users who own objects or were granted permissions on objects in the dumped database must already exist. If they do not, then the restore will fail to recreate the objects with the original ownership and/or permissions. (Sometimes this is what you want, but usually it is not.)

By default, the psql script will continue to execute after an SQL error is encountered. You might wish to use the following command at the top of the script to alter that behaviour and have psql exit with an exit status of 3 if an SQL error occurs:

```plaintext
\set ON_ERROR_STOP
```

Either way, you will have an only partially restored database. Alternatively, you can specify that the whole dump should be restored as a single transaction, so the restore is either fully completed or fully rolled back. This mode can be specified by passing the `-1` or `--single-transaction` command-line options to psql. When using this mode, be aware that even the smallest of errors can rollback a restore that has already run for many hours. However, that might still be preferable to manually cleaning up a complex database after a partially restored dump.

The ability of pg_dump and psql to write to or read from pipes makes it possible to dump a database directly from one server to another, for example:

```plaintext
pg_dump -h host1 dbname | psql -h host2 dbname
```

**Important:** The dumps produced by pg_dump are relative to `template0`. This means that any languages, procedures, etc. added via `template1` will also be dumped by pg_dump. As a result, when restoring, if you are using a customized `template1`, you must create the empty database from `template0`, as in the example above.

After restoring a backup, it is wise to run `ANALYZE` on each database so the query optimizer has useful statistics. An easy way to do this is to run `vacuumdb -a -z`; this is equivalent to running `VACUUM ANALYZE` on each database manually. For more advice on how to load large amounts of data into PostgreSQL efficiently, refer to Section 14.4.

24.1.2. Using pg_dumpall

pg_dump dumps only a single database at a time, and it does not dump information about roles or tablespaces (because those are cluster-wide rather than per-database). To support convenient dumping of the entire contents of a database cluster, the pg_dumpall program is provided. pg_dumpall backs
up each database in a given cluster, and also preserves cluster-wide data such as role and tablespace
definitions. The basic usage of this command is:

```
pg_dumpall > outfile
```

The resulting dump can be restored with psql:

```
psql -f infile postgres
```

(Actually, you can specify any existing database name to start from, but if you are reloading into an
empty cluster then `postgres` should usually be used.) It is always necessary to have database supe-
ruser access when restoring a pg_dumpall dump, as that is required to restore the role and tablespace
information. If you use tablespaces, be careful that the tablespace paths in the dump are appropriate
for the new installation.

pg_dumpall works by emitting commands to re-create roles, tablespaces, and empty databases, then
invoking pg_dump for each database. This means that while each database will be internally consist-
tent, the snapshots of different databases might not be exactly in-sync.

### 24.1.3. Handling large databases

Since PostgreSQL allows tables larger than the maximum file size on your system, it can be problem-
atric to dump such a table to a file, since the resulting file will likely be larger than the maximum size
allowed by your system. Since pg_dump can write to the standard output, you can use standard Unix
tools to work around this possible problem. There are several ways to do it:

**Use compressed dumps.** You can use your favorite compression program, for example gzip:

```
pg_dump dbname | gzip > filename.gz
```

Reload with:

```
gunzip -c filename.gz | psql dbname
```

or:

```
cat filename.gz | gunzip | psql dbname
```

**Use split.** The `split` command allows you to split the output into pieces that are acceptable in size
to the underlying file system. For example, to make chunks of 1 megabyte:

```
pg_dump dbname | split -b 1M - filename
```

Reload with:

```
cat filename* | psql dbname
```

**Use pg_dump’s custom dump format.** If PostgreSQL was built on a system with the zlib compres-
sion library installed, the custom dump format will compress data as it writes it to the output file. This
will produce dump file sizes similar to using gzip, but it has the added advantage that tables can be
restored selectively. The following command dumps a database using the custom dump format:

```
pg_dump -Fc dbname > filename
```

A custom-format dump is not a script for psql, but instead must be restored with `pg_restore`, for
example:
Chapter 24. Backup and Restore

pg_restore -d dbname filename

See the pg_dump and pg_restore reference pages for details.

For very large databases, you might need to combine split with one of the other two approaches.

24.2. File System Level Backup

An alternative backup strategy is to directly copy the files that PostgreSQL uses to store the data in the database. In Section 17.2 it is explained where these files are located, but you have probably found them already if you are interested in this method. You can use whatever method you prefer for doing usual file system backups, for example:

tar -cf backup.tar /usr/local/pgsql/data

There are two restrictions, however, which make this method impractical, or at least inferior to the pg_dump method:

1. The database server must be shut down in order to get a usable backup. Half-way measures such as disallowing all connections will not work (in part because tar and similar tools do not take an atomic snapshot of the state of the file system, but also because of internal buffering within the server). Information about stopping the server can be found in Section 17.5. Needless to say that you also need to shut down the server before restoring the data.

2. If you have dug into the details of the file system layout of the database, you might be tempted to try to back up or restore only certain individual tables or databases from their respective files or directories. This will not work because the information contained in these files contains only half the truth. The other half is in the commit log files pg_clog/*, which contain the commit status of all transactions. A table file is only usable with this information. Of course it is also impossible to restore only a table and the associated pg_clog data because that would render all other tables in the database cluster useless. So file system backups only work for complete backup and restoration of an entire database cluster.

An alternative file-system backup approach is to make a “consistent snapshot” of the data directory, if the file system supports that functionality (and you are willing to trust that it is implemented correctly). The typical procedure is to make a “frozen snapshot” of the volume containing the database, then copy the whole data directory (not just parts, see above) from the snapshot to a backup device, then release the frozen snapshot. This will work even while the database server is running. However, a backup created in this way saves the database files in a state where the database server was not properly shut down; therefore, when you start the database server on the backed-up data, it will think the previous server instance had crashed and replay the WAL log. This is not a problem, just be aware of it (and be sure to include the WAL files in your backup).

If your database is spread across multiple file systems, there might not be any way to obtain exactly-simultaneous frozen snapshots of all the volumes. For example, if your data files and WAL log are on different disks, or if tablespaces are on different file systems, it might not be possible to use snapshot backup because the snapshots must be simultaneous. Read your file system documentation very carefully before trusting to the consistent-snapshot technique in such situations. The safest approach is to shut down the database server for long enough to establish all the frozen snapshots.
Another option is to use rsync to perform a file system backup. This is done by first running rsync while the database server is running, then shutting down the database server just long enough to do a second rsync. The second rsync will be much quicker than the first, because it has relatively little data to transfer, and the end result will be consistent because the server was down. This method allows a file system backup to be performed with minimal downtime.

Note that a file system backup will not necessarily be smaller than an SQL dump. On the contrary, it will most likely be larger. (pg_dump does not need to dump the contents of indexes for example, just the commands to recreate them.) However, taking a file system backup might be faster.

### 24.3. Continuous Archiving and Point-In-Time Recovery (PITR)

At all times, PostgreSQL maintains a *write ahead log* (WAL) in the `pg_xlog/` subdirectory of the cluster’s data directory. The log describes every change made to the database’s data files. This log exists primarily for crash-safety purposes: if the system crashes, the database can be restored to consistency by “replaying” the log entries made since the last checkpoint. However, the existence of the log makes it possible to use a third strategy for backing up databases: we can combine a file-system-level backup with backup of the WAL files. If recovery is needed, we restore the backup and then replay from the backed-up WAL files to bring the backup up to current time. This approach is more complex to administer than either of the previous approaches, but it has some significant benefits:

- We do not need a perfectly consistent backup as the starting point. Any internal inconsistency in the backup will be corrected by log replay (this is not significantly different from what happens during crash recovery). So we don’t need file system snapshot capability, just tar or a similar archiving tool.

- Since we can string together an indefinitely long sequence of WAL files for replay, continuous backup can be achieved simply by continuing to archive the WAL files. This is particularly valuable for large databases, where it might not be convenient to take a full backup frequently.

- There is nothing that says we have to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique supports *point-in-time recovery*: it is possible to restore the database to its state at any time since your base backup was taken.

- If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a *warm standby system*: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

As with the plain file-system-backup technique, this method can only support restoration of an entire database cluster, not a subset. Also, it requires a lot of archival storage: the base backup might be bulky, and a busy system will generate many megabytes of WAL traffic that have to be archived. Still, it is the preferred backup technique in many situations where high reliability is needed.

To recover successfully using continuous archiving (also called “online backup” by many database vendors), you need a continuous sequence of archived WAL files that extends back at least as far as the start time of your backup. So to get started, you should set up and test your procedure for archiving WAL files before you take your first base backup. Accordingly, we first discuss the mechanics of archiving WAL files.
Chapter 24. Backup and Restore

24.3.1. Setting up WAL archiving

In an abstract sense, a running PostgreSQL system produces an indefinitely long sequence of WAL records. The system physically divides this sequence into WAL segment files, which are normally 16MB apiece (although the segment size can be altered when building PostgreSQL). The segment files are given numeric names that reflect their position in the abstract WAL sequence. When not using WAL archiving, the system normally creates just a few segment files and then “recycles” them by renaming no-longer-needed segment files to higher segment numbers. It’s assumed that a segment file whose contents precede the checkpoint-before-last is no longer of interest and can be recycled.

When archiving WAL data, we need to capture the contents of each segment file once it is filled, and save that data somewhere before the segment file is recycled for reuse. Depending on the application and the available hardware, there could be many different ways of “saving the data somewhere”: we could copy the segment files to an NFS-mounted directory on another machine, write them onto a tape drive (ensuring that you have a way of identifying the original name of each file), or batch them together and burn them onto CDs, or something else entirely. To provide the database administrator with as much flexibility as possible, PostgreSQL tries not to make any assumptions about how the archiving will be done. Instead, PostgreSQL lets the administrator specify a shell command to be executed to copy a completed segment file to wherever it needs to go. The command could be as simple as a `cp`, or it could invoke a complex shell script — it’s all up to you.

To enable WAL archiving, set the `archive_mode` configuration parameter to `on`, and specify the shell command to use in the `archive_command` configuration parameter. In practice these settings will always be placed in the `postgresql.conf` file. In `archive_command`, any `%p` is replaced by the path name of the file to archive, while any `%f` is replaced by the file name only. (The path name is relative to the current working directory, i.e., the cluster’s data directory.) Write `%%` if you need to embed an actual `%` character in the command. The simplest useful command is something like:

```bash
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'

archive_command = 'copy "%p" "C:\server\archivedir\%f"'
```

which will copy archivable WAL segments to the directory `/mnt/server/archivedir`. (This is an example, not a recommendation, and might not work on all platforms.) After the `%p` and `%f` parameters have been replaced, the actual command executed might look like this:

```bash
test ! -f /mnt/server/archivedir/00000010000000A900000065 && cp pg_xlog/00000010000000A900000065
```

A similar command will be generated for each new file to be archived.

The archive command will be executed under the ownership of the same user that the PostgreSQL server is running as. Since the series of WAL files being archived contains effectively everything in your database, you will want to be sure that the archived data is protected from prying eyes; for example, archive into a directory that does not have group or world read access.

It is important that the archive command return zero exit status if and only if it succeeded. Upon getting a zero result, PostgreSQL will assume that the WAL segment file has been successfully archived, and will remove or recycle it. However, a nonzero status tells PostgreSQL that the file was not archived; it will try again periodically until it succeeds.

The archive command should generally be designed to refuse to overwrite any pre-existing archive file. This is an important safety feature to preserve the integrity of your archive in case of administrator error (such as sending the output of two different servers to the same archive directory).

It is advisable to test your proposed archive command to ensure that it indeed does not overwrite an existing file, and that it returns nonzero status in this case. The example command above for Unix ensures this by including a separate `test` step. On some Unix platforms, `cp` has switches such as `-i` that can be used to do the same thing less verbosely, but you should not rely on these without verifying
that the right exit status is returned. (In particular, GNU \texttt{cp} will return status zero when \texttt{-i} is used and the target file already exists, which is not the desired behavior.)

While designing your archiving setup, consider what will happen if the archive command fails repeatedly because some aspect requires operator intervention or the archive runs out of space. For example, this could occur if you write to tape without an autochanger; when the tape fills, nothing further can be archived until the tape is swapped. You should ensure that any error condition or request to a human operator is reported appropriately so that the situation can be resolved reasonably quickly. The \texttt{pg\_xlog/} directory will continue to fill with WAL segment files until the situation is resolved. (If the filesystem containing \texttt{pg\_xlog/} fills up, PostgreSQL will do a PANIC shutdown. No prior transactions will be lost, but the database will be unavailable until you free some space.)

The speed of the archiving command is not important, so long as it can keep up with the average rate at which your server generates WAL data. Normal operation continues even if the archiving process falls a little behind. If archiving falls significantly behind, this will increase the amount of data that would be lost in the event of a disaster. It will also mean that the \texttt{pg\_xlog/} directory will contain large numbers of not-yet-archived segment files, which could eventually exceed available disk space. You are advised to monitor the archiving process to ensure that it is working as you intend.

In writing your archive command, you should assume that the file names to be archived can be up to 64 characters long and can contain any combination of ASCII letters, digits, and dots. It is not necessary to remember the original relative path (\texttt{%p}) but it is necessary to remember the file name (\texttt{%f}).

Note that although WAL archiving will allow you to restore any modifications made to the data in your PostgreSQL database, it will not restore changes made to configuration files (that is, \texttt{postgresql.conf}, \texttt{pg\_hba.conf} and \texttt{pg\_ident.conf}), since those are edited manually rather than through SQL operations. You might wish to keep the configuration files in a location that will be backed up by your regular file system backup procedures. See Section 18.2 for how to relocate the configuration files.

The archive command is only invoked on completed WAL segments. Hence, if your server generates only little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To put a limit on how old unarchived data can be, you can set \texttt{archive\_timeout} to force the server to switch to a new WAL segment file at least that often. Note that archived files that are ended early due to a forced switch are still the same length as completely full files. It is therefore unwise to set a very short \texttt{archive\_timeout} — it will bloat your archive storage. \texttt{archive\_timeout} settings of a minute or so are usually reasonable.

Also, you can force a segment switch manually with \texttt{pg\_switch\_xlog}, if you want to ensure that a just-finished transaction is archived as soon as possible. Other utility functions related to WAL management are listed in Table 9-53.

When \texttt{archive\_mode} is \texttt{off} some SQL commands are optimized to avoid WAL logging, as described in Section 14.4.7. If archiving were turned on during execution of one of these statements, WAL would not contain enough information for archive recovery. (Crash recovery is unaffected.) For this reason, \texttt{archive\_mode} can only be changed at server start. However, \texttt{archive\_command} can be changed with a configuration file reload. If you wish to temporarily stop archiving, one way to do it is to set \texttt{archive\_command} to the empty string (\texttt{""}). This will cause WAL files to accumulate in \texttt{pg\_xlog/} until a working \texttt{archive\_command} is re-established.
24.3.2. Making a Base Backup

The procedure for making a base backup is relatively simple:

1. Ensure that WAL archiving is enabled and working.

2. Connect to the database as a superuser, and issue the command:

   ```
   SELECT pg_start_backup('label');
   ```

   where `label` is any string you want to use to uniquely identify this backup operation. (One good practice is to use the full path where you intend to put the backup dump file.) `pg_start_backup` creates a backup label file, called `backup_label`, in the cluster directory with information about your backup.

   It does not matter which database within the cluster you connect to to issue this command. You can ignore the result returned by the function; but if it reports an error, deal with that before proceeding.

   `pg_start_backup` can take a long time to finish. This is because it performs a checkpoint, and the I/O required for a checkpoint will be spread out over a significant period of time, by default half your inter-checkpoint interval (see the configuration parameter `checkpoint_completion_target`). Usually this is what you want because it minimizes the impact on query processing. If you just want to start the backup as soon as possible, execute a `CHECKPOINT` command (which performs a checkpoint as quickly as possible) and then immediately execute `pg_start_backup`. Then there will be very little for `pg_start_backup`'s checkpoint to do, and it won’t take long.

3. Perform the backup, using any convenient file-system-backup tool such as tar or cpio. It is neither necessary nor desirable to stop normal operation of the database while you do this.

4. Again connect to the database as a superuser, and issue the command:

   ```
   SELECT pg_stop_backup();
   ```

   This terminates the backup mode and performs an automatic switch to the next WAL segment. The reason for the switch is to arrange that the last WAL segment file written during the backup interval is immediately ready to archive.

5. Once the WAL segment files used during the backup are archived, you are done. The file identified by `pg_stop_backup`'s result is the last segment that needs to be archived to complete the backup. Archival of these files will happen automatically, since you have already configured `archive_command`. In many cases, this happens fairly quickly, but you are advised to monitor your archival system to ensure this has taken place so that you can be certain you have a complete backup.

Some backup tools that you might wish to use emit warnings or errors if the files they are trying to copy change while the copy proceeds. This situation is normal, and not an error, when taking a base backup of an active database; so you need to ensure that you can distinguish complaints of this sort from real errors. For example, some versions of rsync return a separate exit code for “vanished source files”, and you can write a driver script to accept this exit code as a non-error case. Also, some versions of GNU tar return an error code indistinguishable from a fatal error if a file was truncated while tar was copying it. Fortunately, GNU tar versions 1.16 and later exit with 1 if a file was changed during the backup, and 2 for other errors.

It is not necessary to be very concerned about the amount of time elapsed between `pg_start_backup` and the start of the actual backup, nor between the end of the backup and `pg_stop_backup`; a few minutes’ delay won’t hurt anything. (However, if you normally run the
server with `full_page_writes` disabled, you might notice a drop in performance between `pg_start_backup` and `pg_stop_backup`, since `full_page_writes` is effectively forced on during backup mode.) You must ensure that these steps are carried out in sequence without any possible overlap, or you will invalidate the backup.

Be certain that your backup dump includes all of the files underneath the database cluster directory (e.g., `/usr/local/pgsql/data`). If you are using tablespaces that do not reside underneath this directory, be careful to include them as well (and be sure that your backup dump archives symbolic links as links, otherwise the restore will mess up your tablespaces).

You can, however, omit from the backup dump the files within the `pg_xlog/` subdirectory of the cluster directory. This slight complication is worthwhile because it reduces the risk of mistakes when restoring. This is easy to arrange if `pg_xlog/` is a symbolic link pointing to someplace outside the cluster directory, which is a common setup anyway for performance reasons.

To make use of the backup, you will need to keep around all the WAL segment files generated during and after the file system backup. To aid you in doing this, the `pg_stop_backup` function creates a backup history file that is immediately stored into the WAL archive area. This file is named after the first WAL segment file that you need to have to make use of the backup. For example, if the starting WAL file is `000000100001234000055CD` the backup history file will be named something like `000000100001234000055CD.007C9330.backup`. (The second part of the file name stands for an exact position within the WAL file, and can ordinarily be ignored.) Once you have safely archived the file system backup and the WAL segment files used during the backup (as specified in the backup history file), all archived WAL segments with names numerically less are no longer needed to recover the file system backup and can be deleted. However, you should consider keeping several backup sets to be absolutely certain that you can recover your data.

The backup history file is just a small text file. It contains the label string you gave to `pg_start_backup`, as well as the starting and ending times and WAL segments of the backup. If you used the label to identify where the associated dump file is kept, then the archived history file is enough to tell you which dump file to restore, should you need to do so.

Since you have to keep around all the archived WAL files back to your last base backup, the interval between base backups should usually be chosen based on how much storage you want to expend on archived WAL files. You should also consider how long you are prepared to spend recovering, if recovery should be necessary — the system will have to replay all those WAL segments, and that could take awhile if it has been a long time since the last base backup.

It’s also worth noting that the `pg_start_backup` function makes a file named `backup_label` in the database cluster directory, which is then removed again by `pg_stop_backup`. This file will of course be archived as a part of your backup dump file. The backup label file includes the label string you gave to `pg_start_backup`, as well as the time at which `pg_start_backup` was run, and the name of the starting WAL file. In case of confusion it will therefore be possible to look inside a backup dump file and determine exactly which backup session the dump file came from.

It is also possible to make a backup dump while the server is stopped. In this case, you obviously cannot use `pg_start_backup` or `pg_stop_backup`, and you will therefore be left to your own devices to keep track of which backup dump is which and how far back the associated WAL files go. It is generally better to follow the continuous archiving procedure above.
24.3.3. Recovering using a Continuous Archive Backup

Okay, the worst has happened and you need to recover from your backup. Here is the procedure:

1. Stop the server, if it’s running.

2. If you have the space to do so, copy the whole cluster data directory and any tablespaces to a temporary location in case you need them later. Note that this precaution will require that you have enough free space on your system to hold two copies of your existing database. If you do not have enough space, you need at least to copy the contents of the pg_xlog subdirectory of the cluster data directory, as it might contain logs which were not archived before the system went down.

3. Clean out all existing files and subdirectories under the cluster data directory and under the root directories of any tablespaces you are using.

4. Restore the database files from your backup dump. Be careful that they are restored with the right ownership (the database system user, not root!) and with the right permissions. If you are using tablespaces, you should verify that the symbolic links in pg_tblspc/ were correctly restored.

5. Remove any files present in pg_xlog/; these came from the backup dump and are therefore probably obsolete rather than current. If you didn’t archive pg_xlog/ at all, then recreate it, being careful to ensure that you re-establish it as a symbolic link if you had it set up that way before. Be sure to recreate the subdirectory pg_xlog/archive_status/ as well.

6. If you had unarchived WAL segment files that you saved in step 2, copy them into pg_xlog/.
   (It is best to copy them, not move them, so that you still have the unmodified files if a problem occurs and you have to start over.)

7. Create a recovery command file recovery.conf in the cluster data directory (see Recovery Settings). You might also want to temporarily modify pg_hba.conf to prevent ordinary users from connecting until you are sure the recovery has worked.

8. Start the server. The server will go into recovery mode and proceed to read through the archived WAL files it needs. Should the recovery be terminated because of an external error, the server can simply be restarted and it will continue recovery. Upon completion of the recovery process, the server will rename recovery.conf to recovery.done (to prevent accidentally re-entering recovery mode in case of a crash later) and then commence normal database operations.

9. Inspect the contents of the database to ensure you have recovered to where you want to be. If not, return to step 1. If all is well, let in your users by restoring pg_hba.conf to normal.

The key part of all this is to set up a recovery command file that describes how you want to recover and how far the recovery should run. You can use recovery.conf.sample (normally installed in the installation share/ directory) as a prototype. The one thing that you absolutely must specify in recovery.conf is the restore_command, which tells PostgreSQL how to get back archived WAL file segments. Like the archive_command, this is a shell command string. It can contain %f, which is replaced by the name of the desired log file, and %p, which is replaced by the path name to copy the log file to. (The path name is relative to the current working directory, i.e., the cluster’s data directory.) Write % if you need to embed an actual % character in the command. The simplest useful command is something like:

```bash
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

which will copy previously archived WAL segments from the directory /mnt/server/archivedir. You could of course use something much more complicated, perhaps even a shell script that requests
Chapter 24. Backup and Restore

the operator to mount an appropriate tape.

It is important that the command return nonzero exit status on failure. The command will be asked for log files that are not present in the archive; it must return nonzero when so asked. This is not an error condition. Be aware also that the base name of the %p path will be different from %f; do not expect them to be interchangeable.

WAL segments that cannot be found in the archive will be sought in pg_xlog/; this allows use of recent un-archived segments. However segments that are available from the archive will be used in preference to files in pg_xlog/. The system will not overwrite the existing contents of pg_xlog/ when retrieving archived files.

Normally, recovery will proceed through all available WAL segments, thereby restoring the database to the current point in time (or as close as we can get given the available WAL segments). So a normal recovery will end with a “file not found” message, the exact text of the error message depending upon your choice of restore_command. You may also see an error message at the start of recovery for a file named something like 00000001.history. This is also normal and does not indicate a problem in simple recovery situations. See Section 24.3.4 for discussion.

If you want to recover to some previous point in time (say, right before the junior DBA dropped your main transaction table), just specify the required stopping point in recovery.conf. You can specify the stop point, known as the “recovery target”, either by date/time or by completion of a specific transaction ID. As of this writing only the date/time option is very usable, since there are no tools to help you identify with any accuracy which transaction ID to use.

Note: The stop point must be after the ending time of the base backup, i.e., the time of pg_stop_backup. You cannot use a base backup to recover to a time when that backup was still going on. (To recover to such a time, you must go back to your previous base backup and roll forward from there.)

If recovery finds a corruption in the WAL data then recovery will complete at that point and the server will not start. In such a case the recovery process could be re-run from the beginning, specifying a “recovery target” before the point of corruption so that recovery can complete normally. If recovery fails for an external reason, such as a system crash or if the WAL archive has become inaccessible, then the recovery can simply be restarted and it will restart almost from where it failed. Recovery restart works much like checkpointing in normal operation: the server periodically forces all its state to disk, and then updates the pg_control file to indicate that the already-processed WAL data need not be scanned again.

24.3.3.1. Recovery Settings

These settings can only be made in the recovery.conf file, and apply only for the duration of the recovery. They must be reset for any subsequent recovery you wish to perform. They cannot be changed once recovery has begun.

restore_command(string)

The shell command to execute to retrieve an archived segment of the WAL file series. This parameter is required. Any %f in the string is replaced by the name of the file to retrieve from the archive, and any %p is replaced by the path name to copy it to on the server. (The path name is relative to the current working directory, i.e., the cluster’s data directory.) Any %r is replaced by the name of the file containing the last valid restart point. That is the earliest file that must be kept to allow a restore to be restartable, so this information can be used to truncate the archive to just the minimum required to support restart of the current restore. %r would only be used in
Chapter 24. Backup and Restore

a warm-standby configuration (see Section 24.4). Write `%` to embed an actual `%` character in the command.

It is important for the command to return a zero exit status if and only if it succeeds. The command will be asked for file names that are not present in the archive; it must return nonzero when so asked. Examples:

```bash
restore_command = 'cp /mnt/server/archivedir/%f "%p"
restore_command = 'copy "C:\server\archivedir\%f" "%p"' # Windows
```

`recovery_target_time` (timestamp)

This parameter specifies the time stamp up to which recovery will proceed. At most one of `recovery_target_time` and `recovery_target_xid` can be specified. The default is to recover to the end of the WAL log. The precise stopping point is also influenced by `recovery_target_inclusive`.

`recovery_target_xid` (string)

This parameter specifies the transaction ID up to which recovery will proceed. Keep in mind that while transaction IDs are assigned sequentially at transaction start, transactions can complete in a different numeric order. The transactions that will be recovered are those that committed before (and optionally including) the specified one. At most one of `recovery_target_xid` and `recovery_target_time` can be specified. The default is to recover to the end of the WAL log. The precise stopping point is also influenced by `recovery_target_inclusive`.

`recovery_target_inclusive` (boolean)

Specifies whether we stop just after the specified recovery target (true), or just before the recovery target (false). Applies to both `recovery_target_time` and `recovery_target_xid`, whichever one is specified for this recovery. This indicates whether transactions having exactly the target commit time or ID, respectively, will be included in the recovery. Default is true.

`recovery_target_timeline` (string)

Specifies recovering into a particular timeline. The default is to recover along the same timeline that was current when the base backup was taken. You would only need to set this parameter in complex re-recovery situations, where you need to return to a state that itself was reached after a point-in-time recovery. See Section 24.3.4 for discussion.

`log_restartpoints` (boolean)

Specifies whether to log each restart point as it occurs. This can be helpful to track the progress of a long recovery. Default is false.

24.3.4. Timelines

The ability to restore the database to a previous point in time creates some complexities that are akin to science-fiction stories about time travel and parallel universes. In the original history of the database, perhaps you dropped a critical table at 5:15PM on Tuesday evening, but didn’t realize your mistake until Wednesday noon. Unfazed, you get out your backup, restore to the point-in-time 5:14PM Tuesday evening, and are up and running. In this history of the database universe, you never dropped the table at all. But suppose you later realize this wasn’t such a great idea after all, and would like to return to sometime Wednesday morning in the original history. You won’t be able to if, while your database was up-and-running, it overwrote some of the sequence of WAL segment files that led up to the time you now wish you could get back to. So you really want to distinguish the series of WAL
records generated after you’ve done a point-in-time recovery from those that were generated in the original database history.

To deal with these problems, PostgreSQL has a notion of timelines. Whenever an archive recovery is completed, a new timeline is created to identify the series of WAL records generated after that recovery. The timeline ID number is part of WAL segment file names, and so a new timeline does not overwrite the WAL data generated by previous timelines. It is in fact possible to archive many different timelines. While that might seem like a useless feature, it’s often a lifesaver. Consider the situation where you aren’t quite sure what point-in-time to recover to, and so have to do several point-in-time recoveries by trial and error until you find the best place to branch off from the old history. Without timelines this process would soon generate an unmanageable mess. With timelines, you can recover to any prior state, including states in timeline branches that you later abandoned.

Each time a new timeline is created, PostgreSQL creates a “timeline history” file that shows which timeline it branched off from and when. These history files are necessary to allow the system to pick the right WAL segment files when recovering from an archive that contains multiple timelines. Therefore, they are archived into the WAL archive area just like WAL segment files. The history files are just small text files, so it’s cheap and appropriate to keep them around indefinitely (unlike the segment files which are large). You can, if you like, add comments to a history file to make your own notes about how and why this particular timeline came to be. Such comments will be especially valuable when you have a thicket of different timelines as a result of experimentation.

The default behavior of recovery is to recover along the same timeline that was current when the base backup was taken. If you want to recover into some child timeline (that is, you want to return to some state that was itself generated after a recovery attempt), you need to specify the target timeline ID in recovery.conf. You cannot recover into timelines that branched off earlier than the base backup.

### 24.3.5. Tips and Examples

Some tips for configuring continuous archiving are given here.

#### 24.3.5.1. Standalone hot backups

It is possible to use PostgreSQL’s backup facilities to produce standalone hot backups. These are backups that cannot be used for point-in-time recovery, yet are typically much faster to backup and restore than pg_dump dumps. (They are also much larger than pg_dump dumps, so in some cases the speed advantage could be negated.)

To prepare for standalone hot backups, set `archive_mode` to on, and set up an `archive_command` that performs archiving only when a “switch file” exists. For example:

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f /var/lib/pgsql/archive/%f && cp %p /var/lib/pgsql/archive/%f)
```

This command will perform archiving when `/var/lib/pgsql/backup_in_progress` exists, and otherwise silently return zero exit status (allowing PostgreSQL to recycle the unwanted WAL file).

With this preparation, a backup can be taken using a script like the following:

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
sleep 20
rm /var/lib/pgsql/backup_in_progress
```

```
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```
Chapter 24. Backup and Restore

The switch file /var/lib/pgsql/backup_in_progress is created first, enabling archiving of completed WAL files to occur. After the backup the switch file is removed. Archived WAL files are then added to the backup so that both base backup and all required WAL files are part of the same tar file.

24.3.5.2. archive_command scripts

Many people choose to use scripts to define their archive_command, so that their postgresql.conf entry looks very simple:

```
archive_command = 'local_backup_script.sh "%p" "%f"
```

Using a separate script file is advisable any time you want to use more than a single command in the archiving process. This allows all complexity to be managed within the script, which can be written in a popular scripting language such as bash or perl.

Examples of requirements that might be solved within a script include:

- Copying data to secure off-site data storage
- Batching WAL files so that they are transferred every three hours, rather than one at a time
- Interfacing with other backup and recovery software
- Interfacing with monitoring software to report errors

**Tip:** When using an archive_command script, it’s desirable to enable logging_collector. Any messages written to stderr from the script will then appear in the database server log, allowing complex configurations to be diagnosed easily if they fail.

24.3.6. Caveats

At this writing, there are several limitations of the continuous archiving technique. These will probably be fixed in future releases:

- Operations on hash indexes are not presently WAL-logged, so replay will not update these indexes. The recommended workaround is to manually `REINDEX` each such index after completing a recovery operation.
- If a `CREATE DATABASE` command is executed while a base backup is being taken, and then the template database that the `CREATE DATABASE` copied is modified while the base backup is still in progress, it is possible that recovery will cause those modifications to be propagated into the created database as well. This is of course undesirable. To avoid this risk, it is best not to modify any template databases while taking a base backup.
- `CREATE TABLESPACE` commands are WAL-logged with the literal absolute path, and will therefore be replayed as tablespace creations with the same absolute path. This might be undesirable if the log is being replayed on a different machine. It can be dangerous even if the log is being replayed on the same machine, but into a new data directory: the replay will still overwrite the
Chapter 24. Backup and Restore

contents of the original tablespace. To avoid potential gotchas of this sort, the best practice is to take a new base backup after creating or dropping tablespaces.

It should also be noted that the default WAL format is fairly bulky since it includes many disk page snapshots. These page snapshots are designed to support crash recovery, since we might need to fix partially-written disk pages. Depending on your system hardware and software, the risk of partial writes might be small enough to ignore, in which case you can significantly reduce the total volume of archived logs by turning off page snapshots using the full_page_writes parameter. (Read the notes and warnings in Chapter 28 before you do so.) Turning off page snapshots does not prevent use of the logs for PITR operations. An area for future development is to compress archived WAL data by removing unnecessary page copies even when full_page_writes is on. In the meantime, administrators might wish to reduce the number of page snapshots included in WAL by increasing the checkpoint interval parameters as much as feasible.

24.4. Warm Standby Servers for High Availability

Continuous archiving can be used to create a high availability (HA) cluster configuration with one or more standby servers ready to take over operations if the primary server fails. This capability is widely referred to as warm standby or log shipping.

The primary and standby server work together to provide this capability, though the servers are only loosely coupled. The primary server operates in continuous archiving mode, while each standby server operates in continuous recovery mode, reading the WAL files from the primary. No changes to the database tables are required to enable this capability, so it offers low administration overhead in comparison with some other replication approaches. This configuration also has relatively low performance impact on the primary server.

Directly moving WAL records from one database server to another is typically described as log shipping. PostgreSQL implements file-based log shipping, which means that WAL records are transferred one file (WAL segment) at a time. WAL files (16MB) can be shipped easily and cheaply over any distance, whether it be to an adjacent system, another system on the same site or another system on the far side of the globe. The bandwidth required for this technique varies according to the transaction rate of the primary server. Record-based log shipping is also possible with custom-developed procedures, as discussed in Section 24.4.4.

It should be noted that the log shipping is asynchronous, i.e. the WAL records are shipped after transaction commit. As a result there is a window for data loss should the primary server suffer a catastrophic failure: transactions not yet shipped will be lost. The length of the window of data loss can be limited by use of the archive_timeout parameter, which can be set as low as a few seconds if required. However such low settings will substantially increase the bandwidth requirements for file shipping. If you need a window of less than a minute or so, it’s probably better to look into record-based log shipping.

The standby server is not available for access, since it is continually performing recovery processing. Recovery performance is sufficiently good that the standby will typically be only moments away from full availability once it has been activated. As a result, we refer to this capability as a warm standby configuration that offers high availability. Restoring a server from an archived base backup and rollforward will take considerably longer, so that technique only offers a solution for disaster recovery, not high availability.
24.4.1. Planning

It is usually wise to create the primary and standby servers so that they are as similar as possible, at least from the perspective of the database server. In particular, the path names associated with tablespaces will be passed across as-is, so both primary and standby servers must have the same mount paths for tablespaces if that feature is used. Keep in mind that if `CREATE TABLESPACE` is executed on the primary, any new mount point needed for it must be created on both the primary and all standby servers before the command is executed. Hardware need not be exactly the same, but experience shows that maintaining two identical systems is easier than maintaining two dissimilar ones over the lifetime of the application and system. In any case the hardware architecture must be the same — shipping from, say, a 32-bit to a 64-bit system will not work.

In general, log shipping between servers running different major PostgreSQL release levels will not be possible. It is the policy of the PostgreSQL Global Development Group not to make changes to disk formats during minor release upgrades, so it is likely that running different minor release levels on primary and standby servers will work successfully. However, no formal support for that is offered and you are advised to keep primary and standby servers at the same release level as much as possible. When updating to a new minor release, the safest policy is to update the standby servers first — a new minor release is more likely to be able to read WAL files from a previous minor release than vice versa.

There is no special mode required to enable a standby server. The operations that occur on both primary and standby servers are entirely normal continuous archiving and recovery tasks. The only point of contact between the two database servers is the archive of WAL files that both share: primary writing to the archive, standby reading from the archive. Care must be taken to ensure that WAL archives for separate primary servers do not become mixed together or confused. The archive need not be large, if it is only required for the standby operation.

The magic that makes the two loosely coupled servers work together is simply a `restore_command` used on the standby that waits for the next WAL file to become available from the primary. The `restore_command` is specified in the `recovery.conf` file on the standby server. Normal recovery processing would request a file from the WAL archive, reporting failure if the file was unavailable. For standby processing it is normal for the next file to be unavailable, so we must be patient and wait for it to appear. A waiting `restore_command` can be written as a custom script that loops after polling for the existence of the next WAL file. There must also be some way to trigger failover, which should interrupt the `restore_command`, break the loop and return a file-not-found error to the standby server. This ends recovery and the standby will then come up as a normal server.

Pseudocode for a suitable `restore_command` is:

```c
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L); // wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
    CopyWALFileForRecovery();
```

A working example of a waiting `restore_command` is provided as a `contrib` module named `pg_standby`. This example can be extended as needed to support specific configurations or environments.
Chapter 24. Backup and Restore

PostgreSQL does not provide the system software required to identify a failure on the primary and notify the standby system and then the standby database server. Many such tools exist and are well integrated with other aspects required for successful failover, such as IP address migration.

The means for triggering failover is an important part of planning and design. The `restore_command` is executed in full once for each WAL file. The process running the `restore_command` is therefore created and dies for each file, so there is no daemon or server process and so we cannot use signals and a signal handler. A more permanent notification is required to trigger the failover. It is possible to use a simple timeout facility, especially if used in conjunction with a known `archive_timeout` setting on the primary. This is somewhat error prone since a network problem or busy primary server might be sufficient to initiate failover. A notification mechanism such as the explicit creation of a trigger file is less error prone, if this can be arranged.

The size of the WAL archive can be minimized by using the `%r` option of the `restore_command`. This option specifies the last archive file name that needs to be kept to allow the recovery to restart correctly. This can be used to truncate the archive once files are no longer required, if the archive is writable from the standby server.

### 24.4.2. Implementation

The short procedure for configuring a standby server is as follows. For full details of each step, refer to previous sections as noted.

1. Set up primary and standby systems as near identically as possible, including two identical copies of PostgreSQL at the same release level.

2. Set up continuous archiving from the primary to a WAL archive located in a directory on the standby server. Ensure that `archive_mode`, `archive_command` and `archive_timeout` are set appropriately on the primary (see Section 24.3.1).

3. Make a base backup of the primary server (see Section 24.3.2), and load this data onto the standby.

4. Begin recovery on the standby server from the local WAL archive, using a `recovery.conf` that specifies a `restore_command` that waits as described previously (see Section 24.3.3).

Recovery treats the WAL archive as read-only, so once a WAL file has been copied to the standby system it can be copied to tape at the same time as it is being read by the standby database server. Thus, running a standby server for high availability can be performed at the same time as files are stored for longer term disaster recovery purposes.

For testing purposes, it is possible to run both primary and standby servers on the same system. This does not provide any worthwhile improvement in server robustness, nor would it be described as HA.

### 24.4.3. Failover

If the primary server fails then the standby server should begin failover procedures.

If the standby server fails then no failover need take place. If the standby server can be restarted, even some time later, then the recovery process can also be immediately restarted, taking advantage of restartable recovery. If the standby server cannot be restarted, then a full new standby server instance should be created.
If the primary server fails and then immediately restarts, you must have a mechanism for informing it that it is no longer the primary. This is sometimes known as STONITH (Shoot the Other Node In The Head), which is necessary to avoid situations where both systems think they are the primary, which will lead to confusion and ultimately data loss.

Many failover systems use just two systems, the primary and the standby, connected by some kind of heartbeat mechanism to continually verify the connectivity between the two and the viability of the primary. It is also possible to use a third system (called a witness server) to prevent some cases of inappropriate failover, but the additional complexity might not be worthwhile unless it is set up with sufficient care and rigorous testing.

Once failover to the standby occurs, we have only a single server in operation. This is known as a degenerate state. The former standby is now the primary, but the former primary is down and might stay down. To return to normal operation we must fully recreate a standby server, either on the former primary system when it comes up, or on a third, possibly new, system. Once complete the primary and standby can be considered to have switched roles. Some people choose to use a third server to provide backup to the new primary until the new standby server is recreated, though clearly this complicates the system configuration and operational processes.

So, switching from primary to standby server can be fast but requires some time to re-prepare the failover cluster. Regular switching from primary to standby is useful, since it allows regular downtime on each system for maintenance. This also serves as a test of the failover mechanism to ensure that it will really work when you need it. Written administration procedures are advised.

### 24.4.4. Record-based Log Shipping

PostgreSQL directly supports file-based log shipping as described above. It is also possible to implement record-based log shipping, though this requires custom development.

An external program can call the `pg_xlogfile_name_offset()` function (see Section 9.23) to find out the file name and the exact byte offset within it of the current end of WAL. It can then access the WAL file directly and copy the data from the last known end of WAL through the current end over to the standby server(s). With this approach, the window for data loss is the polling cycle time of the copying program, which can be very small, but there is no wasted bandwidth from forcing partially-used segment files to be archived. Note that the standby servers’ `restore_command` scripts still deal in whole WAL files, so the incrementally copied data is not ordinarily made available to the standby servers. It is of use only when the primary dies — then the last partial WAL file is fed to the standby before allowing it to come up. So correct implementation of this process requires cooperation of the `restore_command` script with the data copying program.

### 24.5. Migration Between Releases

This section discusses how to migrate your database data from one PostgreSQL release to a newer one. The software installation procedure *per se* is not the subject of this section; those details are in Chapter 15.

As a general rule, the internal data storage format is subject to change between major releases of PostgreSQL (where the number after the first dot changes). This does not apply to different minor releases under the same major release (where the number after the second dot changes); these always have compatible storage formats. For example, releases 8.1.1, 8.2.3, and 8.3 are not compatible, whereas 8.2.3 and 8.2.4 are. When you update between compatible versions, you can simply replace
Chapter 24. Backup and Restore

the executables and reuse the data directory on disk. Otherwise you need to back up your data and restore it on the new server. This has to be done using pg_dump; file system level backup methods obviously won’t work. There are checks in place that prevent you from using a data directory with an incompatible version of PostgreSQL, so no great harm can be done by trying to start the wrong server version on a data directory.

It is recommended that you use the pg_dump and pg_dumpall programs from the newer version of PostgreSQL, to take advantage of any enhancements that might have been made in these programs. Current releases of the dump programs can read data from any server version back to 7.0.

The least downtime can be achieved by installing the new server in a different directory and running both the old and the new servers in parallel, on different ports. Then you can use something like:

```bash
pg_dumpall -p 5432 | psql -d postgres -p 6543
```

to transfer your data. Or use an intermediate file if you want. Then you can shut down the old server and start the new server at the port the old one was running at. You should make sure that the old database is not updated after you begin to run pg_dumpall, otherwise you will lose that data. See Chapter 21 for information on how to prohibit access.

It is also possible to use replication methods, such as Slony, to create a slave server with the updated version of PostgreSQL. The slave can be on the same computer or a different computer. Once it has synced up with the master server (running the older version of PostgreSQL), you can switch masters and make the slave the master and shut down the older database instance. Such a switch-over results in only several seconds of downtime for an upgrade.

In practice you probably want to test your client applications on the new setup before switching over completely. This is another reason for setting up concurrent installations of old and new versions.

If you cannot or do not want to run two servers in parallel, you can do the backup step before installing the new version, bring down the server, move the old version out of the way, install the new version, start the new server, and restore the data. For example:

```bash
pg_dumpall > backup
pg_ctl stop
mv /usr/local/pgsql /usr/local/pgsql.old
cd ~/postgresql-8.3.23
gmake install
initdb -D /usr/local/pgsql/data
postgres -D /usr/local/pgsql/data
psql -f backup postgres
```

See Chapter 17 about ways to start and stop the server and other details. The installation instructions will advise you of strategic places to perform these steps.

**Note:** When you “move the old installation out of the way” it might no longer be perfectly usable. Some of the executable programs contain absolute paths to various installed programs and data files. This is usually not a big problem, but if you plan on using two installations in parallel for a while you should assign them different installation directories at build time. (This problem is rectified in PostgreSQL 8.0 and later, so long as you move all subdirectories containing installed files together; for example if `/usr/local/postgres/bin/` goes to `/usr/local/postgres.old/bin/`, then `/usr/local/postgres/share/` must go to `/usr/local/postgres.old/share/`. In pre-8.0 releases moving an installation like this will not work.)
Chapter 25. High Availability, Load Balancing, and Replication

Database servers can work together to allow a second server to take over quickly if the primary server fails (high availability), or to allow several computers to serve the same data (load balancing). Ideally, database servers could work together seamlessly. Web servers serving static web pages can be combined quite easily by merely load-balancing web requests to multiple machines. In fact, read-only database servers can be combined relatively easily too. Unfortunately, most database servers have a read/write mix of requests, and read/write servers are much harder to combine. This is because though read-only data needs to be placed on each server only once, a write to any server has to be propagated to all servers so that future read requests to those servers return consistent results.

This synchronization problem is the fundamental difficulty for servers working together. Because there is no single solution that eliminates the impact of the sync problem for all use cases, there are multiple solutions. Each solution addresses this problem in a different way, and minimizes its impact for a specific workload.

Some solutions deal with synchronization by allowing only one server to modify the data. Servers that can modify data are called read/write or "master" servers. Servers that can reply to read-only queries are called "slave" servers. Servers that cannot be accessed until they are changed to master servers are called "standby" servers.

Some solutions are synchronous, meaning that a data-modifying transaction is not considered committed until all servers have committed the transaction. This guarantees that a failover will not lose any data and that all load-balanced servers will return consistent results no matter which server is queried. In contrast, asynchronous solutions allow some delay between the time of a commit and its propagation to the other servers, opening the possibility that some transactions might be lost in the switch to a backup server, and that load balanced servers might return slightly stale results. Asynchronous communication is used when synchronous would be too slow.

Solutions can also be categorized by their granularity. Some solutions can deal only with an entire database server, while others allow control at the per-table or per-database level.

Performance must be considered in any choice. There is usually a trade-off between functionality and performance. For example, a full synchronous solution over a slow network might cut performance by more than half, while an asynchronous one might have a minimal performance impact.

The remainder of this section outlines various failover, replication, and load balancing solutions. A glossary¹ is also available.

Shared Disk Failover

Shared disk failover avoids synchronization overhead by having only one copy of the database. It uses a single disk array that is shared by multiple servers. If the main database server fails, the standby server is able to mount and start the database as though it was recovering from a database crash. This allows rapid failover with no data loss.

Shared hardware functionality is common in network storage devices. Using a network file system is also possible, though care must be taken that the file system has full POSIX behavior (see Section 17.2.1). One significant limitation of this method is that if the shared disk array fails or becomes corrupt, the primary and standby servers are both nonfunctional. Another issue is that the standby server should never access the shared storage while the primary server is running.

¹. http://www.postgres-r.org/documentation/terms
Chapter 25. High Availability, Load Balancing, and Replication

File System (Block-Device) Replication

A modified version of shared hardware functionality is file system replication, where all changes to a file system are mirrored to a file system residing on another computer. The only restriction is that the mirroring must be done in a way that ensures the standby server has a consistent copy of the file system — specifically, writes to the standby must be done in the same order as those on the master. DRBD is a popular file system replication solution for Linux.

Warm Standby Using Point-In-Time Recovery (PITR)

A warm standby server (see Section 24.4) can be kept current by reading a stream of write-ahead log (WAL) records. If the main server fails, the warm standby contains almost all of the data of the main server, and can be quickly made the new master database server. This is asynchronous and can only be done for the entire database server.

Master-Slave Replication

A master-slave replication setup sends all data modification queries to the master server. The master server asynchronously sends data changes to the slave server. The slave can answer read-only queries while the master server is running. The slave server is ideal for data warehouse queries.

Slony-I is an example of this type of replication, with per-table granularity, and support for multiple slaves. Because it updates the slave server asynchronously (in batches), there is possible data loss during fail over.

Statement-Based Replication Middleware

With statement-based replication middleware, a program intercepts every SQL query and sends it to one or all servers. Each server operates independently. Read-write queries are sent to all servers, while read-only queries can be sent to just one server, allowing the read workload to be distributed.

If queries are simply broadcast unmodified, functions like `random()`, `CURRENT_TIMESTAMP`, and sequences would have different values on different servers. This is because each server operates independently, and because SQL queries are broadcast (and not actual modified rows). If this is unacceptable, either the middleware or the application must query such values from a single server and then use those values in write queries. Also, care must be taken that all transactions either commit or abort on all servers, perhaps using two-phase commit (`PREPARE TRANSACTION` and `COMMIT PREPARED`). Pgpool-II and Sequoia are examples of this type of replication.

Asynchronous Multimaster Replication

For servers that are not regularly connected, like laptops or remote servers, keeping data consistent among servers is a challenge. Using asynchronous multimaster replication, each server works independently, and periodically communicates with the other servers to identify conflicting transactions. The conflicts can be resolved by users or conflict resolution rules.

Synchronous Multimaster Replication

In synchronous multimaster replication, each server can accept write requests, and modified data is transmitted from the original server to every other server before each transaction commits. Heavy write activity can cause excessive locking, leading to poor performance. In fact, write performance is often worse than that of a single server. Read requests can be sent to any server. Some implementations use shared disk to reduce the communication overhead. Synchronous multimaster replication is best for mostly read workloads, though its big advantage is that any server can accept write requests — there is no need to partition workloads between master and slave servers, and because the data changes are sent from one server to another, there is no problem with non-deterministic functions like `random()`.
Chapter 25. High Availability, Load Balancing, and Replication

PostgreSQL does not offer this type of replication, though PostgreSQL two-phase commit (*PREPARE TRANSACTION* and *COMMIT PREPARED*) can be used to implement this in application code or middleware.

**Commercial Solutions**

Because PostgreSQL is open source and easily extended, a number of companies have taken PostgreSQL and created commercial closed-source solutions with unique failover, replication, and load balancing capabilities.

Table 25-1 summarizes the capabilities of the various solutions listed above.

**Table 25-1. High Availability, Load Balancing, and Replication Feature Matrix**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Shared Disk Failover</th>
<th>File System Replication</th>
<th>Warm Standby Using PITR</th>
<th>Master-Slave Replication</th>
<th>Statement-Based Replication Middleware</th>
<th>Asynchronous Multi-master Replication</th>
<th>Synchronous Multi-master Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>No special hardware required</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Allows multiple master servers</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No master server overhead</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No waiting for multiple servers</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master failure will never lose data</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slaves accept read-only queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Per-table granularity</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No conflict resolution necessary</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>Shared Disk Failover</td>
<td>File System Replication</td>
<td>Warm Standby Using PITR</td>
<td>Master-Slave Replication</td>
<td>Statement-Based Replication Middleware</td>
<td>Asynchronous Multi-master Replication</td>
<td>Synchronous Multi-master Replication</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------</td>
<td>-------------------------</td>
<td>-------------------------</td>
<td>--------------------------</td>
<td>----------------------------------------</td>
<td>--------------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>Communication Method</td>
<td>shared disk</td>
<td>disk blocks</td>
<td>WAL</td>
<td>table rows</td>
<td>SQL</td>
<td>table rows</td>
<td>table rows and row locks</td>
</tr>
</tbody>
</table>

There are a few solutions that do not fit into the above categories:

Data Partitioning

Data partitioning splits tables into data sets. Each set can be modified by only one server. For example, data can be partitioned by offices, e.g. London and Paris, with a server in each office. If queries combining London and Paris data are necessary, an application can query both servers, or master/slave replication can be used to keep a read-only copy of the other office’s data on each server.

Multiple-Server Parallel Query Execution

Many of the above solutions allow multiple servers to handle multiple queries, but none allow a single query to use multiple servers to complete faster. This solution allows multiple servers to work concurrently on a single query. It is usually accomplished by splitting the data among servers and having each server execute its part of the query and return results to a central server where they are combined and returned to the user. Pgpool-II has this capability. Also, this can be implemented using the PL/Proxy toolset.
Chapter 26. Monitoring Database Activity

A database administrator frequently wonders, “What is the system doing right now?” This chapter discusses how to find that out.

Several tools are available for monitoring database activity and analyzing performance. Most of this chapter is devoted to describing PostgreSQL’s statistics collector, but one should not neglect regular Unix monitoring programs such as `ps`, `top`, `iostat`, and `vmstat`. Also, once one has identified a poorly-performing query, further investigation might be needed using PostgreSQL’s `EXPLAIN` command. Section 14.1 discusses `EXPLAIN` and other methods for understanding the behavior of an individual query.

26.1. Standard Unix Tools

On most platforms, PostgreSQL modifies its command title as reported by `ps`, so that individual server processes can readily be identified. A sample display is

```
$ ps auxww | grep ^postgres
postgres 960 0.0 1.1 6104 1480 pts/1 SN 13:17 0:00 postgres -i
postgres 963 0.0 1.1 7084 1472 pts/1 SN 13:17 0:00 postgres: writer process
postgres 965 0.0 1.1 6152 1512 pts/1 SN 13:17 0:00 postgres: stats collector
postgres 998 0.0 2.3 6532 2992 pts/1 SN 13:18 0:00 postgres: tgl runbug 127.0.0.1 idle
postgres 1003 0.0 2.4 6532 3128 pts/1 SN 13:19 0:00 postgres: tgl regression [local] SELECT waiting
postgres 1016 0.1 2.4 6532 3080 pts/1 SN 13:19 0:00 postgres: tgl regression
```

(The appropriate invocation of `ps` varies across different platforms, as do the details of what is shown. This example is from a recent Linux system.) The first process listed here is the master server process. The command arguments shown for it are the same ones given when it was launched. The next two processes are background worker processes automatically launched by the master process. (The “stats collector” process will not be present if you have set the system not to start the statistics collector.) Each of the remaining processes is a server process handling one client connection. Each such process sets its command line display in the form

```
postgres: user database host activity
```

The user, database, and connection source host items remain the same for the life of the client connection, but the activity indicator changes. The activity can be `idle` (i.e., waiting for a client command), `idle in transaction` (waiting for client inside a `BEGIN` block), or a command type name such as `SELECT`. Also, waiting is attached if the server process is presently waiting on a lock held by another server process. In the above example we can infer that process 1003 is waiting for process 1016 to complete its transaction and thereby release some lock or other.

If you have turned off `update_process_title` then the activity indicator is not updated; the process title is set only once when a new process is launched. On some platforms this saves a useful amount of per-command overhead, on others it’s insignificant.

Tip: Solaris requires special handling. You must use `/usr/ucb/ps`, rather than `/bin/ps`. You also must use two `w` flags, not just one. In addition, your original invocation of the `postgres` command must have a shorter `ps` status display than that provided by each server process. If you fail to do all three things, the `ps` output for each server process will be the original `postgres` command line.
26.2. The Statistics Collector

PostgreSQL’s **statistics collector** is a subsystem that supports collection and reporting of information about server activity. Presently, the collector can count accesses to tables and indexes in both disk-block and individual-row terms. It also tracks total numbers of rows in each table, and the last vacuum and analyze times for each table.

PostgreSQL also supports determining the exact command currently being executed by other server processes. This is an independent facility that does not depend on the collector process.

### 26.2.1. Statistics Collection Configuration

Since collection of statistics adds some overhead to query execution, the system can be configured to collect or not collect information. This is controlled by configuration parameters that are normally set in `postgresql.conf`. (See Chapter 18 for details about setting configuration parameters.)

The parameter `track_counts` controls whether information is actually sent to the collector process and thus determines whether any run-time overhead occurs for event counting.

The parameter `track_activities` enables monitoring of the current command being executed by any server process.

Normally these parameters are set in `postgresql.conf` so that they apply to all server processes, but it is possible to turn them on or off in individual sessions using the `SET` command. (To prevent ordinary users from hiding their activity from the administrator, only superusers are allowed to change these parameters with `SET`.)

### 26.2.2. Viewing Collected Statistics

Several predefined views, listed in Table 26-1, are available to show the results of statistics collection. Alternatively, one can build custom views using the underlying statistics functions.

When using the statistics to monitor current activity, it is important to realize that the information does not update instantaneously. Each individual server process transmits new statistical counts to the collector just before going idle; so a query or transaction still in progress does not affect the displayed totals. Also, the collector itself emits a new report at most once per `PGSTAT_STAT_INTERVAL` milliseconds (500 unless altered while building the server). So the displayed information lags behind actual activity. However, current-query information collected by `track_activities` is always up-to-date.

Another important point is that when a server process is asked to display any of these statistics, it first fetches the most recent report emitted by the collector process and then continues to use this snapshot for all statistical views and functions until the end of its current transaction. So the statistics will appear not to change as long as you continue the current transaction. Similarly, information about the current queries of all processes is collected when any such information is first requested within a transaction, and the same information will be displayed throughout the transaction. This is a feature, not a bug, because it allows you to perform several queries on the statistics and correlate the results without worrying that the numbers are changing underneath you. But if you want to see new results with each query, be sure to do the queries outside any transaction block. Alternatively, you can invoke `pg_stat_clear_snapshot()`, which will discard the current transaction’s statistics snapshot (if any). The next use of statistical information will cause a new snapshot to be fetched.

**Table 26-1. Standard Statistics Views**
<table>
<thead>
<tr>
<th>View Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pg_stat_activity</code></td>
<td>One row per server process, showing database OID, database name, process ID, user OID, user name, current query, query’s waiting status, time at which the current transaction and current query began execution, time at which the process was started, and client’s address and port number. The columns that report data on the current query are available unless the parameter <code>track_activities</code> has been turned off. Furthermore, these columns are only visible if the user examining the view is a superuser or the same as the user owning the process being reported on.</td>
</tr>
<tr>
<td><code>pg_stat_bgwriter</code></td>
<td>One row only, showing cluster-wide statistics from the background writer: number of scheduled checkpoints, requested checkpoints, buffers written by checkpoints and cleaning scans, and the number of times the background writer stopped a cleaning scan because it had written too many buffers. Also includes statistics about the shared buffer pool, including buffers written by backends (that is, not by the background writer) and total buffers allocated.</td>
</tr>
<tr>
<td><code>pg_stat_database</code></td>
<td>One row per database, showing database OID, database name, number of active server processes connected to that database, number of transactions committed and rolled back in that database, total disk blocks read, total buffer hits (i.e., block read requests avoided by finding the block already in buffer cache), number of rows returned, fetched, inserted, updated and deleted.</td>
</tr>
<tr>
<td><code>pg_stat_all_tables</code></td>
<td>For each table in the current database (including TOAST tables), the table OID, schema and table name, number of sequential scans initiated, number of live rows fetched by sequential scans, number of index scans initiated (over all indexes belonging to the table), number of live rows fetched by index scans, numbers of row insertions, updates, and deletions, number of row updates that were HOT (i.e., no separate index update), numbers of live and dead rows, the last time the table was vacuumed manually, the last time it was vacuumed by the autovacuum daemon, the last time it was analyzed manually, and the last time it was analyzed by the autovacuum daemon.</td>
</tr>
<tr>
<td><code>pg_stat_sys_tables</code></td>
<td>Same as <code>pg_stat_all_tables</code>, except that only system tables are shown.</td>
</tr>
</tbody>
</table>
### View Name | Description
--- | ---
pg_stat_user_tables | Same as pg_stat_all_tables, except that only user tables are shown.
pg_stat_all_indexes | For each index in the current database, the table and index OID, schema, table and index name, number of index scans initiated on that index, number of index entries returned by index scans, and number of live table rows fetched by simple index scans using that index.
pg_stat_sys_indexes | Same as pg_stat_all_indexes, except that only indexes on system tables are shown.
pg_stat_user_indexes | Same as pg_stat_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_tables | For each table in the current database (including TOAST tables), the table OID, schema and table name, number of disk blocks read from that table, number of buffer hits, numbers of disk blocks read and buffer hits in all indexes of that table, numbers of disk blocks read and buffer hits from that table’s auxiliary TOAST table (if any), and numbers of disk blocks read and buffer hits for the TOAST table’s index.
pg_statio_sys_tables | Same as pg_statio_all_tables, except that only system tables are shown.
pg_statio_user_tables | Same as pg_statio_all_tables, except that only user tables are shown.
pg_statio_all_indexes | For each index in the current database, the table and index OID, schema, table and index name, numbers of disk blocks read and buffer hits in that index.
pg_statio_sys_indexes | Same as pg_statio_all_indexes, except that only indexes on system tables are shown.
pg_statio_user_indexes | Same as pg_statio_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_sequences | For each sequence object in the current database, the sequence OID, schema and sequence name, numbers of disk blocks read and buffer hits in that sequence.
pg_statio_sys_sequences | Same as pg_statio_all_sequences, except that only indexes on system tables are shown.
pg_statio_user_sequences | Same as pg_statio_all_sequences, except that only indexes on user tables are shown.

The per-index statistics are particularly useful to determine which indexes are being used and how effective they are.

Beginning in PostgreSQL 8.1, indexes can be used either directly or via “bitmap scans”. In a bitmap scan the output of several indexes can be combined via AND or OR rules; so it is difficult to associate
individual heap row fetches with specific indexes when a bitmap scan is used. Therefore, a bitmap scan increments the \texttt{pg\_stat\_all\_indexes.idx\_tup\_read} count(s) for the index(es) it uses, and it increments the \texttt{pg\_stat\_all\_tables.idx\_tup\_fetch} count for the table, but it does not affect \texttt{pg\_stat\_all\_indexes.idx\_tup\_fetch}.

Note: Before PostgreSQL 8.1, the \texttt{idx\_tup\_read} and \texttt{idx\_tup\_fetch} counts were essentially always equal. Now they can be different even without considering bitmap scans, because \texttt{idx\_tup\_read} counts index entries retrieved from the index while \texttt{idx\_tup\_fetch} counts live rows fetched from the table; the latter will be less if any dead or not-yet-committed rows are fetched using the index.

The \texttt{pg\_statio\_views} are primarily useful to determine the effectiveness of the buffer cache. When the number of actual disk reads is much smaller than the number of buffer hits, then the cache is satisfying most read requests without invoking a kernel call. However, these statistics do not give the entire story: due to the way in which PostgreSQL handles disk I/O, data that is not in the PostgreSQL buffer cache might still reside in the kernel’s I/O cache, and might therefore still be fetched without requiring a physical read. Users interested in obtaining more detailed information on PostgreSQL I/O behavior are advised to use the PostgreSQL statistics collector in combination with operating system utilities that allow insight into the kernel’s handling of I/O.

Other ways of looking at the statistics can be set up by writing queries that use the same underlying statistics access functions as these standard views do. These functions are listed in Table 26-2. The per-database access functions take a database OID as argument to identify which database to report on. The per-table and per-index functions take a table or index OID. (Note that only tables and indexes in the current database can be seen with these functions.) The per-server-process access functions take a server process number, which ranges from one to the number of currently active server processes.

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{pg_stat_get_db_numbackends}</td>
<td>\texttt{integer}</td>
<td>Number of active server processes for database</td>
</tr>
<tr>
<td>\texttt{pg_stat_get_db_xact_commit}</td>
<td>\texttt{bigint}</td>
<td>Transactions committed in database</td>
</tr>
<tr>
<td>\texttt{pg_stat_get_db_xact_rollback}</td>
<td>\texttt{bigint}</td>
<td>Transactions rolled back in database</td>
</tr>
<tr>
<td>\texttt{pg_stat_get_db_blocks_fetched}</td>
<td>\texttt{bigint}</td>
<td>Number of disk block fetch requests for database</td>
</tr>
<tr>
<td>\texttt{pg_stat_get_db_blocks_hit}</td>
<td>\texttt{bigint}</td>
<td>Number of disk block fetch requests found in cache for database</td>
</tr>
<tr>
<td>\texttt{pg_stat_get_db_tuples_returned}</td>
<td>\texttt{bigint}</td>
<td>Number of tuples returned for database</td>
</tr>
<tr>
<td>\texttt{pg_stat_get_db_tuples_fetched}</td>
<td>\texttt{bigint}</td>
<td>Number of tuples fetched for database</td>
</tr>
<tr>
<td>\texttt{pg_stat_get_db_tuples_inserted}</td>
<td>\texttt{bigint}</td>
<td>Number of tuples inserted in database</td>
</tr>
<tr>
<td>\texttt{pg_stat_get_db_tuples_updated}</td>
<td>\texttt{bigint}</td>
<td>Number of tuples updated in database</td>
</tr>
</tbody>
</table>
## Chapter 26. Monitoring Database Activity

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pg_stat_get_db_tuples_deleted</code></td>
<td>bigint</td>
<td>Number of tuples deleted in database</td>
</tr>
<tr>
<td><code>pg_stat_get_numscans(oid)</code></td>
<td>bigint</td>
<td>Number of sequential scans done when argument is a table, or number of index scans done when argument is an index</td>
</tr>
<tr>
<td><code>pg_stat_get_tuples_returned</code></td>
<td>bigint</td>
<td>Number of rows read by sequential scans when argument is a table, or number of index entries returned when argument is an index</td>
</tr>
<tr>
<td><code>pg_stat_get_tuples_fetched</code></td>
<td>bigint</td>
<td>Number of table rows fetched by bitmap scans when argument is a table, or table rows fetched by simple index scans using the index when argument is an index</td>
</tr>
<tr>
<td><code>pg_stat_get_tuples_inserted</code></td>
<td>bigint</td>
<td>Number of rows inserted into table</td>
</tr>
<tr>
<td><code>pg_stat_get_tuples_updated</code></td>
<td>bigint</td>
<td>Number of rows updated in table (includes HOT updates)</td>
</tr>
<tr>
<td><code>pg_stat_get_tuples_deleted</code></td>
<td>bigint</td>
<td>Number of rows deleted from table</td>
</tr>
<tr>
<td><code>pg_stat_get_tuples_hot_updated</code></td>
<td>bigint</td>
<td>Number of rows HOT-updated in table</td>
</tr>
<tr>
<td><code>pg_stat_get_live_tuples(oid)</code></td>
<td>bigint</td>
<td>Number of live rows in table</td>
</tr>
<tr>
<td><code>pg_stat_get_dead_tuples(oid)</code></td>
<td>bigint</td>
<td>Number of dead rows in table</td>
</tr>
<tr>
<td><code>pg_stat_get_blocks_fetched</code></td>
<td>bigint</td>
<td>Number of disk block fetch requests for table or index</td>
</tr>
<tr>
<td><code>pg_stat_get_blocks_hit(oid)</code></td>
<td>bigint</td>
<td>Number of disk block requests found in cache for table or index</td>
</tr>
<tr>
<td><code>pg_stat_get_last_vacuum_time(oid)</code></td>
<td>timestamptz</td>
<td>Time of the last vacuum initiated by the user on this table</td>
</tr>
<tr>
<td><code>pg_stat_get_last_autovacuum_time(oid)</code></td>
<td>timestamptz</td>
<td>Time of the last vacuum initiated by the autovacuum daemon on this table</td>
</tr>
<tr>
<td><code>pg_stat_get_last_analyze_time(oid)</code></td>
<td>timestamptz</td>
<td>Time of the last analyze initiated by the user on this table</td>
</tr>
<tr>
<td><code>pg_stat_get_last_autoanalyze_time(oid)</code></td>
<td>timestamptz</td>
<td>Time of the last analyze initiated by the autovacuum daemon on this table</td>
</tr>
</tbody>
</table>
### Chapter 26. Monitoring Database Activity

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pg_stat_get_backend_idset()</code></td>
<td><code>setof integer</code></td>
<td>Set of currently active server process numbers (from 1 to the number of active server processes). See usage example in the text.</td>
</tr>
<tr>
<td><code>pg_backend_pid()</code></td>
<td><code>integer</code></td>
<td>Process ID of the server process attached to the current session.</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_pid(integer)</code></td>
<td><code>integer</code></td>
<td>Process ID of the given server process.</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_dbid(integer)</code></td>
<td><code>oid</code></td>
<td>Database ID of the given server process.</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_userid(integer)</code></td>
<td><code>oid</code></td>
<td>User ID of the given server process.</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_activity(integer)</code></td>
<td><code>text</code></td>
<td>Active command of the given server process, but only if the current user is a superuser or the same user as that of the session being queried (and <code>track_activities</code> is on).</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_waiting(integer)</code></td>
<td><code>boolean</code></td>
<td>True if the given server process is waiting for a lock, but only if the current user is a superuser or the same user as that of the session being queried (and <code>track_activities</code> is on).</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_activity_start(integer)</code></td>
<td><code>timestamp with time zone</code></td>
<td>The time at which the given server process’ currently executing query was started, but only if the current user is a superuser or the same user as that of the session being queried (and <code>track_activities</code> is on).</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_xact_start(integer)</code></td>
<td><code>timestamp with time zone</code></td>
<td>The time at which the given server process’ currently executing transaction was started, but only if the current user is a superuser or the same user as that of the session being queried (and <code>track_activities</code> is on).</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_start(integer)</code></td>
<td><code>timestamp with time zone</code></td>
<td>The time at which the given server process was started, or null if the current user is not a superuser nor the same user as that of the session being queried.</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_client_addr</code></td>
<td><code>inet</code></td>
<td>The IP address of the client connected to the given server process. Null if the connection is over a Unix domain socket. Also null if the current user is not a superuser nor the same user as that of the session being queried</td>
</tr>
<tr>
<td><code>pg_stat_get_backend_client_port</code></td>
<td><code>integer</code></td>
<td>The TCP port number of the client connected to the given server process. -1 if the connection is over a Unix domain socket. Null if the current user is not a superuser nor the same user as that of the session being queried</td>
</tr>
<tr>
<td><code>pg_stat_get_bgwriter_timed_checkpoints</code></td>
<td><code>bigint</code></td>
<td>The number of times the background writer has started timed checkpoints (because the <code>checkpoint_timeout</code> time has expired)</td>
</tr>
<tr>
<td><code>pg_stat_get_bgwriter_requested_checkpoints</code></td>
<td><code>bigint</code></td>
<td>The number of times the background writer has started checkpoints based on requests from backends because the <code>checkpoint_segments</code> has been exceeded or because the <code>CHECKPOINT</code> command has been issued</td>
</tr>
<tr>
<td><code>pg_stat_get_bgwriter_buf_written_checkpoints</code></td>
<td><code>bigint</code></td>
<td>The number of buffers written by the background writer during checkpoints</td>
</tr>
<tr>
<td><code>pg_stat_get_bgwriter_buf_written_clean</code></td>
<td><code>bigint</code></td>
<td>The number of buffers written by the background writer for routine cleaning of dirty pages</td>
</tr>
<tr>
<td><code>pg_stat_get_bgwriter_maxwritten_clean</code></td>
<td><code>bigint</code></td>
<td>The number of times the background writer has stopped its cleaning scan because it has written more buffers than specified in the <code>bgwriter_lru_maxpages</code> parameter</td>
</tr>
<tr>
<td><code>pg_stat_get_buf_written_backend</code></td>
<td><code>bigint</code></td>
<td>The number of buffers written by backends because they needed to allocate a new buffer</td>
</tr>
<tr>
<td><code>pg_stat_get_buf_alloc</code></td>
<td><code>bigint</code></td>
<td>The total number of buffer allocations</td>
</tr>
</tbody>
</table>
Chapter 26. Monitoring Database Activity

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pg_stat_clear_snapshot()</code></td>
<td>void</td>
<td>Discard the current statistics snapshot</td>
</tr>
<tr>
<td><code>pg_stat_reset()</code></td>
<td>void</td>
<td>Reset all statistics counters for the current database to zero (requires superuser privileges)</td>
</tr>
</tbody>
</table>

Note: `blocks_fetched` minus `blocks_hit` gives the number of kernel read() calls issued for the table, index, or database; but the actual number of physical reads is usually lower due to kernel-level buffering.

The function `pg_stat_get_backend_idset` provides a convenient way to generate one row for each active server process. For example, to show the PIDs and current queries of all server processes:

```sql
SELECT pg_stat_get_backend_pid(s.backendid) AS procpid,
       pg_stat_get_backend_activity(s.backendid) AS current_query
FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

26.3. Viewing Locks

Another useful tool for monitoring database activity is the `pg_locks` system table. It allows the database administrator to view information about the outstanding locks in the lock manager. For example, this capability can be used to:

- View all the locks currently outstanding, all the locks on relations in a particular database, all the locks on a particular relation, or all the locks held by a particular PostgreSQL session.
- Determine the relation in the current database with the most ungranted locks (which might be a source of contention among database clients).
- Determine the effect of lock contention on overall database performance, as well as the extent to which contention varies with overall database traffic.

Details of the `pg_locks` view appear in Section 44.46. For more information on locking and managing concurrency with PostgreSQL, refer to Chapter 13.

26.4. Dynamic Tracing

PostgreSQL provides facilities to support dynamic tracing of the database server. This allows an external utility to be called at specific points in the code and thereby trace execution. Currently, this facility is primarily intended for use by database developers, as it requires substantial familiarity with the code.

A number of trace points, often called probes, are already inserted into the source code. By default these probes are disabled, and the user needs to explicitly tell the configure script to make the probes available in PostgreSQL.
Currently, only the DTrace utility is supported, which is only available on Solaris Express and Solaris 10+. It is expected that DTrace will be available in the future on FreeBSD and Mac OS X. Supporting other dynamic tracing utilities is theoretically possible by changing the definitions for the `PG_TRACE` macros in `src/include/pg_trace.h`.

### 26.4.1. Compiling for Dynamic Tracing

By default, trace points are disabled, so you will need to explicitly tell the configure script to make the probes available in PostgreSQL. To include DTrace support specify `--enable-dtrace` to configure. See Section 15.5 for further information.

### 26.4.2. Built-in Trace Points

A few standard trace points are provided in the source code (of course, more can be added as needed for a particular problem). These are shown in Table 26-3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameters</th>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction__start</td>
<td>(int transactionId)</td>
<td>The start of a new transaction.</td>
</tr>
<tr>
<td>transaction__commit</td>
<td>(int transactionId)</td>
<td>The successful completion of a transaction.</td>
</tr>
<tr>
<td>transaction__abort</td>
<td>(int transactionId)</td>
<td>The unsuccessful completion of a transaction.</td>
</tr>
<tr>
<td>lwlock__acquire</td>
<td>(int lockid, int mode)</td>
<td>An LWLock has been acquired.</td>
</tr>
<tr>
<td>lwlock__release</td>
<td>(int lockid, int mode)</td>
<td>An LWLock has been released.</td>
</tr>
<tr>
<td>lwlock__startwait</td>
<td>(int lockid, int mode)</td>
<td>An LWLock was not immediately available and a backend has begun to wait for the lock to become available.</td>
</tr>
<tr>
<td>lwlock__endwait</td>
<td>(int lockid, int mode)</td>
<td>A backend has been released from its wait for an LWLock.</td>
</tr>
<tr>
<td>lwlock__condacquire</td>
<td>(int lockid, int mode)</td>
<td>An LWLock was successfully acquired when the caller specified no waiting.</td>
</tr>
<tr>
<td>lwlock__condacquire__fail</td>
<td>(int lockid, int mode)</td>
<td>An LWLock was not successfully acquired when the caller specified no waiting.</td>
</tr>
<tr>
<td>lock__startwait</td>
<td>(int locktag_field2, int lockmode)</td>
<td>A request for a heavyweight lock (lmgr lock) has begun to wait because the lock is not available.</td>
</tr>
<tr>
<td>lock__endwait</td>
<td>(int locktag_field2, int lockmode)</td>
<td>A request for a heavyweight lock (lmgr lock) has finished waiting (i.e., has acquired the lock).</td>
</tr>
</tbody>
</table>
26.4.3. Using Trace Points

The example below shows a DTrace script for analyzing transaction counts on the system, as an alternative to snapshotting `pg_stat_database` before and after a performance test:

```d
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
    @start["Start"] = count();
    self->ts = timestamp;
}

postgresql$1:::transaction-abort
{
    @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
    @commit["Commit"] = count();
    @time["Total time (ns)"] = sum(timestamp - self->ts);
    self->ts=0;
}
```

Note how the double underline in trace point names needs to be replaced by a hyphen when using D script. When executed, the example D script gives output such as:

```
# ./txn_count.d 'pgrep -n postgres'
^C

Start 71
Commit 70
Total time (ns) 2312105013
```

You should remember that trace programs need to be carefully written and debugged prior to their use, otherwise the trace information collected might be meaningless. In most cases where problems are found it is the instrumentation that is at fault, not the underlying system. When discussing information found using dynamic tracing, be sure to enclose the script used to allow that too to be checked and discussed.

26.4.4. Defining Trace Points

New trace points can be defined within the code wherever the developer desires, though this will require a recompilation.

A trace point can be inserted by using one of the trace macros. These are chosen according to how many variables will be made available for inspection at that trace point. Tracing the occurrence of an event can be achieved with a single line, using just the trace point name, e.g.:

```d
PG_TRACE (my__new__trace__point);
```
More complex trace points can be provided with one or more variables for inspection by the dynamic tracing utility by using the \texttt{PG\_TRACE\_n} macro that corresponds to the number of parameters after the trace point name:

\begin{verbatim}
PG\_TRACE3 (my\_complex\_event, varX, varY, varZ);
\end{verbatim}

The definition of the transaction\_start trace point is shown below:

\begin{verbatim}
static void StartTransaction(void)
{
    ... 
    /*
    * generate a new transaction id
    */
    s->transactionId = GetNewTransactionId(false);
    XactLockTableInsert(s->transactionId);
    PG\_TRACE1(transaction\_start, s->transactionId);
    ... 
}
\end{verbatim}

Note how the transaction ID is made available to the dynamic tracing utility.

The dynamic tracing utility might require you to further define these trace points. For example, DTrace requires you to add new probes to the file \texttt{src/backend/utils/probes.d} as shown here:

\begin{verbatim}
provider postgresql {
    ... 
    probe transaction\_start(int);
    ... 
};
\end{verbatim}

You should take care that the data types specified for the probe arguments match the data types of the variables used in the \texttt{PG\_TRACE} macro. This is not checked at compile time. You can check that your newly added trace point is available by recompiling, then running the new binary, and as root, executing a DTrace command such as:

\begin{verbatim}
dtrace -l -n transaction-start
\end{verbatim}
Chapter 27. Monitoring Disk Usage

This chapter discusses how to monitor the disk usage of a PostgreSQL database system.

27.1. Determining Disk Usage

Each table has a primary heap disk file where most of the data is stored. If the table has any columns with potentially-wide values, there is also a TOAST file associated with the table, which is used to store values too wide to fit comfortably in the main table (see Section 53.2). There will be one index on the TOAST table, if present. There might also be indexes associated with the base table. Each table and index is stored in a separate disk file — possibly more than one file, if the file would exceed one gigabyte. Naming conventions for these files are described in Section 53.1.

You can monitor disk space from three ways: using SQL functions listed in Table 9-54, using VACUUM information, and from the command line using the tools in contrib/oid2name. The SQL functions are the easiest to use and report information about tables, tables with indexes and long value storage (TOAST), databases, and tablespaces.

Using psql on a recently vacuumed or analyzed database, you can issue queries to see the disk usage of any table:

```
SELECT relfilenode, relpages FROM pg_class WHERE relname = 'customer';
```

```
relfilenode | relpages
-------------+----------
16806 | 60
(1 row)
```

Each page is typically 8 kilobytes. (Remember, relpages is only updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX.) The relfilenode value is of interest if you want to examine the table’s disk file directly.

To show the space used by TOAST tables, use a query like the following:

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid FROM pg_class
      WHERE relname = 'customer') ss
WHERE oid = ss.reltoastrelid
  OR oid = (SELECT reltoastidxid FROM pg_class
            WHERE oid = ss.reltoastrelid)
ORDER BY relname;
```

```
relname        | relpages
---------------+----------
pg_toast_16806 | 0
pg_toast_16806_index | 1
```

You can easily display index sizes, too:

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer'
  AND c.oid = i.indrelid
```

461
Chapter 27. Monitoring Disk Usage

AND c2.oid = i.indexrelid
ORDER BY c2.relname;

<table>
<thead>
<tr>
<th>relname</th>
<th>relpages</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer_id_indexdex</td>
<td>26</td>
</tr>
</tbody>
</table>

It is easy to find your largest tables and indexes using this information:

```
SELECT relname, relpages FROM pg_class ORDER BY relpages DESC;
```

<table>
<thead>
<tr>
<th>relname</th>
<th>relpages</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigtable</td>
<td>3290</td>
</tr>
<tr>
<td>customer</td>
<td>3144</td>
</tr>
</tbody>
</table>

You can also use `contrib/oid2name` to show disk usage. See `README.oid2name` in that directory for examples. It includes a script that shows disk usage for each database.

27.2. Disk Full Failure

The most important disk monitoring task of a database administrator is to make sure the disk doesn’t grow full. A filled data disk will not result in data corruption, but it might prevent useful activity from occurring. If the disk holding the WAL files grows full, database server panic and consequent shutdown might occur.

If you cannot free up additional space on the disk by deleting other things, you can move some of the database files to other file systems by making use of tablespaces. See Section 20.6 for more information about that.

**Tip:** Some file systems perform badly when they are almost full, so do not wait until the disk is completely full to take action.

If your system supports per-user disk quotas, then the database will naturally be subject to whatever quota is placed on the user the server runs as. Exceeding the quota will have the same bad effects as running out of space entirely.
Chapter 28. Reliability and the Write-Ahead Log

This chapter explains how the Write-Ahead Log is used to obtain efficient, reliable operation.

28.1. Reliability

Reliability is an important property of any serious database system, and PostgreSQL does everything possible to guarantee reliable operation. One aspect of reliable operation is that all data recorded by a committed transaction should be stored in a nonvolatile area that is safe from power loss, operating system failure, and hardware failure (except failure of the nonvolatile area itself, of course). Successfully writing the data to the computer’s permanent storage (disk drive or equivalent) ordinarily meets this requirement. In fact, even if a computer is fatally damaged, if the disk drives survive they can be moved to another computer with similar hardware and all committed transactions will remain intact.

While forcing data periodically to the disk platters might seem like a simple operation, it is not. Because disk drives are dramatically slower than main memory and CPUs, several layers of caching exist between the computer’s main memory and the disk platters. First, there is the operating system’s buffer cache, which caches frequently requested disk blocks and combines disk writes. Fortunately, all operating systems give applications a way to force writes from the buffer cache to disk, and PostgreSQL uses those features. (See the wal_sync_method parameter to adjust how this is done.)

Next, there might be a cache in the disk drive controller; this is particularly common on RAID controller cards. Some of these caches are write-through, meaning writes are passed along to the drive as soon as they arrive. Others are write-back, meaning data is passed on to the drive at some later time. Such caches can be a reliability hazard because the memory in the disk controller cache is volatile, and will lose its contents in a power failure. Better controller cards have battery-backed caches, meaning the card has a battery that maintains power to the cache in case of system power loss. After power is restored the data will be written to the disk drives.

And finally, most disk drives have caches. Some are write-through while some are write-back, and the same concerns about data loss exist for write-back drive caches as exist for disk controller caches. Consumer-grade IDE and SATA drives are particularly likely to have write-back caches that will not survive a power failure. To check write caching on Linux use hdparm -I; it is enabled if there is a * next to Write cache. hdparm -W to turn off write caching. On FreeBSD use atacmd. (For SCSI disks use sdparm to turn off WCE.) On Solaris the disk write cache is controlled by format -e. (The Solaris ZFS file system is safe with disk write-cache enabled because it issues its own disk cache flush commands.) On Windows if wal_sync_method is open_dataasync (the default), write caching is disabled by unchecking My Computer\Open\{select disk drive}\Properties\Hardware\Properties\Policies\Enable write caching on the disk. Also on Windows, fsync and fsync_writethrough never do write caching. The fsync_writethrough option can also be used to disable write caching on MacOS X.

When the operating system sends a write request to the disk hardware, there is little it can do to make sure the data has arrived at a truly non-volatile storage area. Rather, it is the administrator’s responsibility to be sure that all storage components ensure data integrity. Avoid disk controllers that have non-battery-backed write caches. At the drive level, disable write-back caching if the drive cannot guarantee the data will be written before shutdown.

1. http://sg.torque.net/sdparm.html
2. http://www.sun.com/bigadmin/content/submitted/format_utility.jsp
Another risk of data loss is posed by the disk platter write operations themselves. Disk platters are divided into sectors, commonly 512 bytes each. Every physical read or write operation processes a whole sector. When a write request arrives at the drive, it might be for 512 bytes, 1024 bytes, or 8192 bytes, and the process of writing could fail due to power loss at any time, meaning some of the 512-byte sectors were written, and others were not. To guard against such failures, PostgreSQL periodically writes full page images to permanent storage before modifying the actual page on disk. By doing this, during crash recovery PostgreSQL can restore partially-written pages. If you have a battery-backed disk controller or file-system software that prevents partial page writes (e.g., ReiserFS 4), you can turn off this page imaging by using the full_page_writes parameter.

28.2. Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) is a standard method for ensuring data integrity. A detailed description can be found in most (if not all) books about transaction processing. Briefly, WAL’s central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, after log records describing the changes have been flushed to permanent storage. If we follow this procedure, we do not need to flush data pages to disk on every transaction commit, because we know that in the event of a crash we will be able to recover the database using the log: any changes that have not been applied to the data pages can be redone from the log records. (This is roll-forward recovery, also known as REDO.)

Using WAL results in a significantly reduced number of disk writes, because only the log file needs to be flushed to disk to guarantee that a transaction is committed, rather than every data file changed by the transaction. The log file is written sequentially, and so the cost of syncing the log is much less than the cost of flushing the data pages. This is especially true for servers handling many small transactions touching different parts of the data store. Furthermore, when the server is processing many small concurrent transactions, one fsync of the log file may suffice to commit many transactions.

WAL also makes it possible to support on-line backup and point-in-time recovery, as described in Section 24.3. By archiving the WAL data we can support reverting to any time instant covered by the available WAL data: we simply install a prior physical backup of the database, and replay the WAL log just as far as the desired time. What’s more, the physical backup doesn’t have to be an instantaneous snapshot of the database state — if it is made over some period of time, then replaying the WAL log for that period will fix any internal inconsistencies.

28.3. Asynchronous Commit

Asynchronous commit is an option that allows transactions to complete more quickly, at the cost that the most recent transactions may be lost if the database should crash. In many applications this is an acceptable trade-off.

As described in the previous section, transaction commit is normally synchronous: the server waits for the transaction’s WAL records to be flushed to permanent storage before returning a success indication to the client. The client is therefore guaranteed that a transaction reported to be committed will be preserved, even in the event of a server crash immediately after. However, for short transactions this delay is a major component of the total transaction time. Selecting asynchronous commit mode means that the server returns success as soon as the transaction is logically completed, before the WAL records it generated have actually made their way to disk. This can provide a significant boost in throughput for small transactions.
Chapter 28. Reliability and the Write-Ahead Log

Asynchronous commit introduces the risk of data loss. There is a short time window between the report of transaction completion to the client and the time that the transaction is truly committed (that is, it is guaranteed not to be lost if the server crashes). Thus asynchronous commit should not be used if the client will take external actions relying on the assumption that the transaction will be remembered. As an example, a bank would certainly not use asynchronous commit for a transaction recording an ATM’s dispensing of cash. But in many scenarios, such as event logging, there is no need for a strong guarantee of this kind.

The risk that is taken by using asynchronous commit is of data loss, not data corruption. If the database should crash, it will recover by replaying WAL up to the last record that was flushed. The database will therefore be restored to a self-consistent state, but any transactions that were not yet flushed to disk will not be reflected in that state. The net effect is therefore loss of the last few transactions. Because the transactions are replayed in commit order, no inconsistency can be introduced — for example, if transaction B made changes relying on the effects of a previous transaction A, it is not possible for A’s effects to be lost while B’s effects are preserved.

The user can select the commit mode of each transaction, so that it is possible to have both synchronous and asynchronous commit transactions running concurrently. This allows flexible trade-offs between performance and certainty of transaction durability. The commit mode is controlled by the user-settable parameter synchronous_commit, which can be changed in any of the ways that a configuration parameter can be set. The mode used for any one transaction depends on the value of synchronous_commit when transaction commit begins.

Certain utility commands, for instance DROP TABLE, are forced to commit synchronously regardless of the setting of synchronous_commit. This is to ensure consistency between the server’s file system and the logical state of the database. The commands supporting two-phase commit, such as PREPARE TRANSACTION, are also always synchronous.

If the database crashes during the risk window between an asynchronous commit and the writing of the transaction’s WAL records, then changes made during that transaction will be lost. The duration of the risk window is limited because a background process (the “WAL writer”) flushes unwritten WAL records to disk every wal_writer_delay milliseconds. The actual maximum duration of the risk window is three times wal_writer_delay because the WAL writer is designed to favor writing whole pages at a time during busy periods.

Caution

An immediate-mode shutdown is equivalent to a server crash, and will therefore cause loss of any unflushed asynchronous commits.

Asynchronous commit provides behavior different from setting fsync = off. fsync is a server-wide setting that will alter the behavior of all transactions. It disables all logic within PostgreSQL that attempts to synchronize writes to different portions of the database, and therefore a system crash (that is, a hardware or operating system crash, not a failure of PostgreSQL itself) could result in arbitrarily bad corruption of the database state. In many scenarios, asynchronous commit provides most of the performance improvement that could be obtained by turning off fsync, but without the risk of data corruption.

commit_delay also sounds very similar to asynchronous commit, but it is actually a synchronous commit method (in fact, commit_delay is ignored during an asynchronous commit). commit_delay causes a delay just before a synchronous commit attempts to flush WAL to disk, in the hope that a single flush executed by one such transaction can also serve other transactions committing at about the same time. Setting commit_delay can only help when there are many concurrently committing transactions, and it is difficult to tune it to a value that actually helps rather than hurting throughput.
28.4. WAL Configuration

There are several WAL-related configuration parameters that affect database performance. This section explains their use. Consult Chapter 18 for general information about setting server configuration parameters.

**Checkpoints** are points in the sequence of transactions at which it is guaranteed that the data files have been updated with all information written before the checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the log file. In the event of a crash, the crash recovery procedure looks at the latest checkpoint record to determine the point in the log (known as the redo record) from which it should start the REDO operation. Any changes made to data files before that point are known to be already on disk. Hence, after a checkpoint has been made, any log segments preceding the one containing the redo record are no longer needed and can be recycled or removed. (When WAL archiving is being done, the log segments must be archived before being recycled or removed.)

The server's background writer process will automatically perform a checkpoint every so often. A checkpoint is created every checkpoint_segments log segments, or every checkpoint_timeout seconds, whichever comes first. The default settings are 3 segments and 300 seconds respectively. It is also possible to force a checkpoint by using the SQL command `CHECKPOINT`.

Reducing `checkpoint_segments` and/or `checkpoint_timeout` causes checkpoints to be done more often. This allows faster after-crash recovery (since less work will need to be redone). However, one must balance this against the increased cost of flushing dirty data pages more often. If `full_page_writes` is set (as is the default), there is another factor to consider. To ensure data page consistency, the first modification of a data page after each checkpoint results in logging the entire page content. In that case, a smaller checkpoint interval increases the volume of output to the WAL log, partially negating the goal of using a smaller interval, and in any case causing more disk I/O.

Checkpoints are fairly expensive, first because they require writing out all currently dirty buffers, and second because they result in extra subsequent WAL traffic as discussed above. It is therefore wise to set the checkpointing parameters high enough that checkpoints don’t happen too often. As a simple sanity check on your checkpointing parameters, you can set the `checkpoint_warning` parameter. If checkpoints happen closer together than `checkpoint_warning` seconds, a message will be output to the server log recommending increasing `checkpoint_segments`. Occasional appearance of such a message is not cause for alarm, but if it appears often then the checkpoint control parameters should be increased. Bulk operations such as large `COPY` transfers might cause a number of such warnings to appear if you have not set `checkpoint_segments` high enough.

To avoid flooding the I/O system with a burst of page writes, writing dirty buffers during a checkpoint is spread over a period of time. That period is controlled by `checkpoint_completion_target`, which is given as a fraction of the checkpoint interval. The I/O rate is adjusted so that the checkpoint finishes when the given fraction of `checkpoint_segments` WAL segments have been consumed since checkpoint start, or the given fraction of `checkpoint_timeout` seconds have elapsed, whichever is sooner. With the default value of 0.5, PostgreSQL can be expected to complete each checkpoint in about half the time before the next checkpoint starts. On a system that’s very close to maximum I/O throughput during normal operation, you might want to increase `checkpoint_completion_target` to reduce the I/O load from checkpoints. The disadvantage of this is that prolonging checkpoints affects recovery time, because more WAL segments will need to be kept around for possible use in recovery. Although `checkpoint_completion_target` can be set as high as 1.0, it is best to keep it less than that (perhaps 0.9 at most) since checkpoints include some other activities besides writing dirty buffers. A setting of 1.0 is quite likely to result in checkpoints not being completed on time, which would result in performance loss due to unexpected variation in the number of WAL segments needed.
There will always be at least one WAL segment file, and will normally not be more than \((2 + \text{checkpoint_completion_target}) \times \text{checkpoint_segments} + 1\) files. Each segment file is normally 16 MB (though this size can be altered when building the server). You can use this to estimate space requirements for WAL. Ordinarily, when old log segment files are no longer needed, they are recycled (renamed to become the next segments in the numbered sequence). If, due to a short-term peak of log output rate, there are more than \(3 \times \text{checkpoint_segments} + 1\) segment files, the unneeded segment files will be deleted instead of recycled until the system gets back under this limit.

There are two commonly used internal WAL functions: LogInsert and LogFlush. LogInsert is used to place a new record into the WAL buffers in shared memory. If there is no space for the new record, LogInsert will have to write (move to kernel cache) a few filled WAL buffers. This is undesirable because LogInsert is used on every database low level modification (for example, row insertion) at a time when an exclusive lock is held on affected data pages, so the operation needs to be as fast as possible. What is worse, writing WAL buffers might also force the creation of a new log segment, which takes even more time. Normally, WAL buffers should be written and flushed by a LogFlush request, which is made, for the most part, at transaction commit time to ensure that transaction records are flushed to permanent storage. On systems with high log output, LogFlush requests might not occur often enough to prevent LogInsert from having to do writes. On such systems one should increase the number of WAL buffers by modifying the configuration parameter wal_buffers. The default number of WAL buffers is 8. Increasing this value will correspondingly increase shared memory usage. When full_page_writes is set and the system is very busy, setting this value higher will help smooth response times during the period immediately following each checkpoint.

The commit_delay parameter defines for how many microseconds the server process will sleep after writing a commit record to the log with LogInsert but before performing a LogFlush. This delay allows other server processes to add their commit records to the log so as to have all of them flushed with a single log sync. No sleep will occur if fsync is not enabled, nor if fewer than commit_siblings other sessions are currently in active transactions; this avoids sleeping when it’s unlikely that any other session will commit soon. Note that on most platforms, the resolution of a sleep request is ten milliseconds, so that any nonzero commit_delay setting between 1 and 10000 microseconds would have the same effect. Good values for these parameters are not yet clear; experimentation is encouraged.

The wal_sync_method parameter determines how PostgreSQL will ask the kernel to force WAL updates out to disk. All the options should be the same in terms of reliability, with the exception of fsync_writethrough, which can sometimes force a flush of the disk cache even when other options do not do so. However, it’s quite platform-specific which one will be the fastest; you can test option speeds using the utility src/tools/fsync in the PostgreSQL source tree. Note that this parameter is irrelevant if fsync has been turned off.

Enabling the wal_debug configuration parameter (provided that PostgreSQL has been compiled with support for it) will result in each LogInsert and LogFlush WAL call being logged to the server log. This option might be replaced by a more general mechanism in the future.

### 28.5. WAL Internals

WAL is automatically enabled; no action is required from the administrator except ensuring that the disk-space requirements for the WAL logs are met, and that any necessary tuning is done (see Section 28.4).

WAL logs are stored in the directory `pg_xlog` under the data directory, as a set of segment files, normally each 16 MB in size. Each segment is divided into pages, normally 8 kB each. The log record headers are described in `access/xlog.h`; the record content is dependent on the type of
event that is being logged. Segment files are given ever-increasing numbers as names, starting at 000000010000000000000000. The numbers do not wrap, at present, but it should take a very very long time to exhaust the available stock of numbers.

It is of advantage if the log is located on another disk than the main database files. This can be achieved by moving the directory \texttt{pg\_xlog} to another location (while the server is shut down, of course) and creating a symbolic link from the original location in the main data directory to the new location.

The aim of WAL, to ensure that the log is written before database records are altered, can be subverted by disk drives that falsely report a successful write to the kernel, when in fact they have only cached the data and not yet stored it on the disk. A power failure in such a situation might still lead to irrecoverable data corruption. Administrators should try to ensure that disks holding PostgreSQL’s WAL log files do not make such false reports. (See Section 28.1.)

After a checkpoint has been made and the log flushed, the checkpoint’s position is saved in the file \texttt{pg\_control}. Therefore, when recovery is to be done, the server first reads \texttt{pg\_control} and then the checkpoint record; then it performs the REDO operation by scanning forward from the log position indicated in the checkpoint record. Because the entire content of data pages is saved in the log on the first page modification after a checkpoint, all pages changed since the checkpoint will be restored to a consistent state.

To deal with the case where \texttt{pg\_control} is corrupted, we should support the possibility of scanning existing log segments in reverse order — newest to oldest — in order to find the latest checkpoint. This has not been implemented yet. \texttt{pg\_control} is small enough (less than one disk page) that it is not subject to partial-write problems, and as of this writing there have been no reports of database failures due solely to inability to read \texttt{pg\_control} itself. So while it is theoretically a weak spot, \texttt{pg\_control} does not seem to be a problem in practice.
Chapter 29. Regression Tests

The regression tests are a comprehensive set of tests for the SQL implementation in PostgreSQL. They test standard SQL operations as well as the extended capabilities of PostgreSQL.

29.1. Running the Tests

The regression tests can be run against an already installed and running server, or using a temporary installation within the build tree. Furthermore, there is a “parallel” and a “sequential” mode for running the tests. The sequential method runs each test script in turn, whereas the parallel method starts up multiple server processes to run groups of tests in parallel. Parallel testing gives confidence that interprocess communication and locking are working correctly. For historical reasons, the sequential test is usually run against an existing installation and the parallel method against a temporary installation, but there are no technical reasons for this.

To run the regression tests after building but before installation, type

gmake check

in the top-level directory. (Or you can change to src/test/regress and run the command there.) This will first build several auxiliary files, such as some sample user-defined trigger functions, and then run the test driver script. At the end you should see something like

======================
All 100 tests passed.
======================

or otherwise a note about which tests failed. See Section 29.2 below before assuming that a “failure” represents a serious problem.

Because this test method runs a temporary server, it will not work when you are the root user (since the server will not start as root). If you already did the build as root, you do not have to start all over. Instead, make the regression test directory writable by some other user, log in as that user, and restart the tests. For example

root# chmod -R a+w src/test/regress
root# chmod -R a+w contrib/spi
root# su - joeuser
joeuser$ cd top-level build directory
joeuser$ gmake check

(The only possible “security risk” here is that other users might be able to alter the regression test results behind your back. Use common sense when managing user permissions.)

Alternatively, run the tests after installation.

If you have configured PostgreSQL to install into a location where an older PostgreSQL installation already exists, and you perform gmake check before installing the new version, you might find that the tests fail because the new programs try to use the already-installed shared libraries. (Typical symptoms are complaints about undefined symbols.) If you wish to run the tests before overwriting the old installation, you’ll need to build with configure --disable-rpath. It is not recommended that you use this option for the final installation, however.

The parallel regression test starts quite a few processes under your user ID. Presently, the maximum concurrency is twenty parallel test scripts, which means forty processes: there’s a server process
and a psql process for each test script. So if your system enforces a per-user limit on the number of processes, make sure this limit is at least fifty or so, else you might get random-seeming failures in the parallel test. If you are not in a position to raise the limit, you can cut down the degree of parallelism by setting the `MAX_CONNECTIONS` parameter. For example,

```
gmake MAX_CONNECTIONS=10 check
```

runs no more than ten tests concurrently.

To run the tests after installation (see Chapter 15), initialize a data area and start the server, as explained in Chapter 17, then type

```
gmake installcheck
```

or for a parallel test

```
gmake installcheck-parallel
```

The tests will expect to contact the server at the local host and the default port number, unless directed otherwise by `PGHOST` and `PGPORT` environment variables.

The source distribution also contains regression tests for the optional procedural languages and for some of the `contrib` modules. At present, these tests can be used only against an already-installed server. To run the tests for all procedural languages that have been built and installed, change to the `src/pl` directory of the build tree and type

```
gmake installcheck
```

You can also do this in any of the subdirectories of `src/pl` to run tests for just one procedural language. To run the tests for all `contrib` modules that have them, change to the `contrib` directory of the build tree and type

```
gmake installcheck
```

The `contrib` modules must have been built and installed first. You can also do this in a subdirectory of `contrib` to run the tests for just one module.

### 29.2. Test Evaluation

Some properly installed and fully functional PostgreSQL installations can “fail” some of these regression tests due to platform-specific artifacts such as varying floating-point representation and message wording. The tests are currently evaluated using a simple `diff` comparison against the outputs generated on a reference system, so the results are sensitive to small system differences. When a test is reported as “failed”, always examine the differences between expected and actual results; you might find that the differences are not significant. Nonetheless, we still strive to maintain accurate reference files across all supported platforms, so it can be expected that all tests pass.

The actual outputs of the regression tests are in files in the `src/test/regress/results` directory. The test script uses `diff` to compare each output file against the reference outputs stored in the `src/test/regress/expected` directory. Any differences are saved for your inspection in `src/test/regress/regression.diffs`. (Or you can run `diff` yourself, if you prefer.)

If for some reason a particular platform generates a “failure” for a given test, but inspection of the output convinces you that the result is valid, you can add a new comparison file to silence the failure report in future test runs. See Section 29.3 for details.
Chapter 29. Regression Tests

29.2.1. Error message differences

Some of the regression tests involve intentional invalid input values. Error messages can come from either the PostgreSQL code or from the host platform system routines. In the latter case, the messages can vary between platforms, but should reflect similar information. These differences in messages will result in a “failed” regression test that can be validated by inspection.

29.2.2. Locale differences

If you run the tests against an already-installed server that was initialized with a collation-order locale other than C, then there might be differences due to sort order and follow-up failures. The regression test suite is set up to handle this problem by providing alternative result files that together are known to handle a large number of locales.

29.2.3. Date and time differences

Most of the date and time results are dependent on the time zone environment. The reference files are generated for time zone PST8PDT (Berkeley, California), and there will be apparent failures if the tests are not run with that time zone setting. The regression test driver sets environment variable PGTZ to PST8PDT, which normally ensures proper results.

29.2.4. Floating-point differences

Some of the tests involve computing 64-bit floating-point numbers (double precision) from table columns. Differences in results involving mathematical functions of double precision columns have been observed. The float8 and geometry tests are particularly prone to small differences across platforms, or even with different compiler optimization options. Human eyeball comparison is needed to determine the real significance of these differences which are usually 10 places to the right of the decimal point.

Some systems display minus zero as -0, while others just show 0.

Some systems signal errors from pow() and exp() differently from the mechanism expected by the current PostgreSQL code.

29.2.5. Row ordering differences

You might see differences in which the same rows are output in a different order than what appears in the expected file. In most cases this is not, strictly speaking, a bug. Most of the regression test scripts are not so pedantic as to use an ORDER BY for every single SELECT, and so their result row orderings are not well-defined according to the letter of the SQL specification. In practice, since we are looking at the same queries being executed on the same data by the same software, we usually get the same result ordering on all platforms, and so the lack of ORDER BY isn’t a problem. Some queries do exhibit cross-platform ordering differences, however. When testing against an already-installed server, ordering differences can also be caused by non-C locale settings or non-default parameter settings, such as custom values of work_mem or the planner cost parameters.

Therefore, if you see an ordering difference, it’s not something to worry about, unless the query does have an ORDER BY that your result is violating. But please report it anyway, so that we can add an ORDER BY to that particular query and thereby eliminate the bogus “failure” in future releases.
You might wonder why we don’t order all the regression test queries explicitly to get rid of this issue once and for all. The reason is that that would make the regression tests less useful, not more, since they’d tend to exercise query plan types that produce ordered results to the exclusion of those that don’t.

29.2.6. Insufficient stack depth

If the `errors` test results in a server crash at the `select infinite_recurse()` command, it means that the platform’s limit on process stack size is smaller than the `max_stack_depth` parameter indicates. This can be fixed by running the server under a higher stack size limit (4MB is recommended with the default value of `max_stack_depth`). If you are unable to do that, an alternative is to reduce the value of `max_stack_depth`.

29.2.7. The “random” test

The `random` test script is intended to produce random results. In rare cases, this causes the random regression test to fail. Typing:

```
diff results/random.out expected/random.out
```

should produce only one or a few lines of differences. You need not worry unless the random test fails repeatedly.

29.3. Variant Comparison Files

Since some of the tests inherently produce environment-dependent results, we have provided ways to specify alternative “expected” result files. Each regression test can have several comparison files showing possible results on different platforms. There are two independent mechanisms for determining which comparison file is used for each test.

The first mechanism allows comparison files to be selected for specific platforms. There is a mapping file, `src/test/regress/resultmap`, that defines which comparison file to use for each platform. To eliminate bogus test “failures” for a particular platform, you first choose or make a variant result file, and then add a line to the `resultmap` file.

Each line in the mapping file is of the form

```
testname:output:platformpattern=comparisonfilename
```

The test name is just the name of the particular regression test module. The output value indicates which output file to check. For the standard regression tests, this is always `out`. The value corresponds to the file extension of the output file. The platform pattern is a pattern in the style of the Unix tool `expr` (that is, a regular expression with an implicit ^ anchor at the start). It is matched against the platform name as printed by `config.guess`. The comparison file name is the base name of the substitute result comparison file.

For example: some systems interpret very small floating-point values as zero, rather than reporting an underflow error. This causes a few differences in the `float8` regression test. Therefore, we provide a variant comparison file, `float8-small-is-zero.out`, which includes the results to be expected on these systems. To silence the bogus “failure” message on OpenBSD platforms, `resultmap` includes:
The second selection mechanism for variant comparison files is much more automatic: it simply uses the “best match” among several supplied comparison files. The regression test driver script considers both the standard comparison file for a test, testname.out, and variant files named testname_digit.out (where the digit is any single digit 0-9). If any such file is an exact match, the test is considered to pass; otherwise, the one that generates the shortest diff is used to create the failure report. (If resultmap includes an entry for the particular test, then the base testname is the substitute name given in resultmap.)

For example, for the char test, the comparison file char.out contains results that are expected in the C and POSIX locales, while the file char_1.out contains results sorted as they appear in many other locales.

The best-match mechanism was devised to cope with locale-dependent results, but it can be used in any situation where the test results cannot be predicted easily from the platform name alone. A limitation of this mechanism is that the test driver cannot tell which variant is actually “correct” for the current environment; it will just pick the variant that seems to work best. Therefore it is safest to use this mechanism only for variant results that you are willing to consider equally valid in all contexts.
IV. Client Interfaces

This part describes the client programming interfaces distributed with PostgreSQL. Each of these chapters can be read independently. Note that there are many other programming interfaces for client programs that are distributed separately and contain their own documentation (Appendix G lists some of the more popular ones). Readers of this part should be familiar with using SQL commands to manipulate and query the database (see Part II) and of course with the programming language that the interface uses.
Chapter 30. libpq - C Library

libpq is the C application programmer’s interface to PostgreSQL. libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server and to receive the results of these queries.

libpq is also the underlying engine for several other PostgreSQL application interfaces, including those written for C++, Perl, Python, Tcl and ECPG. So some aspects of libpq’s behavior will be important to you if you use one of those packages. In particular, Section 30.12, Section 30.13 and Section 30.16 describe behavior that is visible to the user of any application that uses libpq.

Some short programs are included at the end of this chapter (Section 30.19) to show how to write programs that use libpq. There are also several complete examples of libpq applications in the directory src/test/examples in the source code distribution.

Client programs that use libpq must include the header file libpq-fe.h and must link with the libpq library.

30.1. Database Connection Control Functions

The following functions deal with making a connection to a PostgreSQL backend server. An application program can have several backend connections open at one time. (One reason to do that is to access more than one database.) Each connection is represented by a PGconn object, which is obtained from the function PQconnectdb or PQsetdbLogin. Note that these functions will always return a non-null object pointer, unless perhaps there is too little memory even to allocate the PGconn object. The PQstatus function should be called to check whether a connection was successfully made before queries are sent via the connection object.

PQconnectdb

Makes a new connection to the database server.

\[
\text{PGconn } \ast \text{PQconnectdb}(\text{const char } \ast \text{conninfo});
\]

This function opens a new database connection using the parameters taken from the string conninfo. Unlike PQsetdbLogin below, the parameter set can be extended without changing the function signature, so use of this function (or its nonblocking analogues PQconnectStart and PQconnectPoll) is preferred for new application programming.

The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace. Each parameter setting is in the form keyword = value. Spaces around the equal sign are optional. To write an empty value or a value containing spaces, surround it with single quotes, e.g., keyword = ‘a value’. Single quotes and backslashes within the value must be escaped with a backslash, i.e., ‘ and \. 

The currently recognized parameter key words are:

host

Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default behavior when host is not specified is to connect to a Unix-domain socket in /tmp (or whatever socket directory was specified when Post-
PostgreSQL was built). On machines without Unix-domain sockets, the default is to connect to localhost.

**hostaddr**

Numeric IP address of host to connect to. This should be in the standard IPv4 address format, e.g., 172.28.40.9. If your machine supports IPv6, you can also use those addresses. TCP/IP communication is always used when a nonempty string is specified for this parameter.

Using `hostaddr` instead of `host` allows the application to avoid a host name look-up, which might be important in applications with time constraints. However, Kerberos and GSSAPI authentication requires the host name. The following therefore applies: If `host` is specified without `hostaddr`, a host name lookup occurs. If `hostaddr` is specified without `host`, the value for `hostaddr` gives the remote address. When Kerberos is used, a reverse name query occurs to obtain the host name for Kerberos. If both `host` and `hostaddr` are specified, the value for `hostaddr` gives the remote address; the value for `host` is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. (Note that authentication is likely to fail if `libpq` is passed a host name that is not the name of the machine at `hostaddr`.) Also, `host` rather than `hostaddr` is used to identify the connection in `~/.pgpass` (see Section 30.13).

Without either a host name or host address, `libpq` will connect using a local Unix-domain socket; or on machines without Unix-domain sockets, it will attempt to connect to localhost.

**port**

Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

**dbname**

The database name. Defaults to be the same as the user name.

**user**

PostgreSQL user name to connect as. Defaults to be the same as the operating system name of the user running the application.

**password**

Password to be used if the server demands password authentication.

**connect_timeout**

Maximum wait for connection, in seconds (write as a decimal integer string). Zero or not specified means wait indefinitely. It is not recommended to use a timeout of less than 2 seconds.

**options**

Command-line options to be sent to the server.

**tty**

Ignored (formerly, this specified where to send server debug output).

**sslmode**

This option determines whether or with what priority a SSL TCP/IP connection will be negotiated with the server. There are four modes: `disable` will attempt only an unencrypted SSL connection; `allow` will negotiate, trying first a non-SSL connection, then if that fails,
Chapter 30. libpq - C Library

trying an SSL connection; prefer (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; require will try only an SSL connection. sslmode is ignored for Unix domain socket communication.

If PostgreSQL is compiled without SSL support, using option require will cause an error, while options allow and prefer will be accepted but libpq will not in fact attempt an SSL connection.

requiresl

This option is deprecated in favor of the sslmode setting.

If set to 1, an SSL connection to the server is required (this is equivalent to sslmode require). libpq will then refuse to connect if the server does not accept an SSL connection. If set to 0 (default), libpq will negotiate the connection type with the server (equivalent to sslmode prefer). This option is only available if PostgreSQL is compiled with SSL support.

krbsrvname

Kerberos service name to use when authenticating with Kerberos 5 or GSSAPI. This must match the service name specified in the server configuration for Kerberos authentication to succeed. (See also Section 21.2.5 and Section 21.2.3.)

gsslib

GSS library to use for GSSAPI authentication. Only used on Windows. Set to gssapi to force libpq to use the GSSAPI library for authentication instead of the default SSPI.

service

Service name to use for additional parameters. It specifies a service name in pg_service.conf that holds additional connection parameters. This allows applications to specify only a service name so connection parameters can be centrally maintained. See Section 30.14.

If any parameter is unspecified, then the corresponding environment variable (see Section 30.12) is checked. If the environment variable is not set either, then the indicated built-in defaults are used.

PQsetdbLogin

Makes a new connection to the database server.

PGconn *PQsetdbLogin(const char *pghost,
                        const char *pgport,
                        const char *pgoptions,
                        const char *pgtty,
                        const char *dbName,
                        const char *login,
                        const char *pwd);

This is the predecessor of PQconnectdb with a fixed set of parameters. It has the same functionality except that the missing parameters will always take on default values. Write NULL or an empty string for any one of the fixed parameters that is to be defaulted.

If the dbName contains an = sign, it is taken as a conninfo string in exactly the same way as if it had been passed to PQconnectdb, and the remaining parameters are then applied as above.

PQsetdb

Makes a new connection to the database server.

PGconn *PQsetdb(char *pghost,
Chapter 30. libpq - C Library

This is a macro that calls PQsetdbLogin with null pointers for the login and pwd parameters. It is provided for backward compatibility with very old programs.

PQconnectStart

PQconnectPoll

Make a connection to the database server in a nonblocking manner.

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);

These two functions are used to open a connection to a database server such that your application’s thread of execution is not blocked on remote I/O whilst doing so. The point of this approach is that the waits for I/O to complete can occur in the application’s main loop, rather than down inside PQconnectdb, and so the application can manage this operation in parallel with other activities.

The database connection is made using the parameters taken from the string conninfo, passed to PQconnectStart. This string is in the same format as described above for PQconnectdb.

Neither PQconnectStart nor PQconnectPoll will block, so long as a number of restrictions are met:

- The hostaddr and host parameters are used appropriately to ensure that name and reverse name queries are not made. See the documentation of these parameters under PQconnectdb above for details.

- If you call PQtrace, ensure that the stream object into which you trace will not block.

- You ensure that the socket is in the appropriate state before calling PQconnectPoll, as described below.

To begin a nonblocking connection request, call conn = PQconnectStart("connection_info_string"). If conn is null, then libpq has been unable to allocate a new PGconn structure. Otherwise, a valid PGconn pointer is returned (though not yet representing a valid connection to the database). On return from PQconnectStart, call status = PQstatus(conn). If status equals CONNECTION_BAD, PQconnectStart has failed.

If PQconnectStart succeeds, the next stage is to poll libpq so that it can proceed with the connection sequence. Use PQsocket(conn) to obtain the descriptor of the socket underlying the database connection. Loop thus: If PQconnectPoll(conn) last returned PGRES_POLLING_READING, wait until the socket is ready to read (as indicated by select(), poll(), or similar system function). Then call PQconnectPoll(conn) again. Conversely, if PQconnectPoll(conn) last returned PGRES_POLLING_WRITING, wait until the socket is ready to write, then call PQconnectPoll(conn) again. If you have yet to call PQconnectPoll, i.e., just after the call to PQconnectStart, behave as if it last returned PGRES_POLLING_WRITING. Continue this loop until PQconnectPoll(conn) returns PGRES_POLLING_FAILED, indicating the connection procedure has failed, or PGRES_POLLING_OK, indicating the connection has been successfully made.
At any time during connection, the status of the connection can be checked by calling \texttt{PQstatus}. If this gives \texttt{CONNECTION\_BAD}, then the connection procedure has failed; if it gives \texttt{CONNECTION\_OK}, then the connection is ready. Both of these states are equally detectable from the return value of \texttt{PQconnectPoll}, described above. Other states might also occur during (and only during) an asynchronous connection procedure. These indicate the current stage of the connection procedure and might be useful to provide feedback to the user for example. These statuses are:

\begin{itemize}
  \item \texttt{CONNECTION\_STARTED}
    \begin{itemize}
      \item Waiting for connection to be made.
    \end{itemize}
  \item \texttt{CONNECTION\_MADE}
    \begin{itemize}
      \item Connection OK; waiting to send.
    \end{itemize}
  \item \texttt{CONNECTION\_AWAITING\_RESPONSE}
    \begin{itemize}
      \item Waiting for a response from the server.
    \end{itemize}
  \item \texttt{CONNECTION\_AUTH\_OK}
    \begin{itemize}
      \item Received authentication; waiting for backend start-up to finish.
    \end{itemize}
  \item \texttt{CONNECTION\_SSL\_STARTUP}
    \begin{itemize}
      \item Negotiating SSL encryption.
    \end{itemize}
  \item \texttt{CONNECTION\_SETENV}
    \begin{itemize}
      \item Negotiating environment-driven parameter settings.
    \end{itemize}
\end{itemize}

Note that, although these constants will remain (in order to maintain compatibility), an application should never rely upon these occurring in a particular order, or at all, or on the status always being one of these documented values. An application might do something like this:

\begin{verbatim}
switch(PQstatus(conn))
{
  case CONNECTION\_STARTED:
    feedback = "Connecting...";
    break;

  case CONNECTION\_MADE:
    feedback = "Connected to server...";
    break;

  ...

  default:
    feedback = "Connecting...";
}
\end{verbatim}

The \texttt{connect\_timeout} connection parameter is ignored when using \texttt{PQconnectPoll}; it is the application’s responsibility to decide whether an excessive amount of time has elapsed. Otherwise, \texttt{PQconnectStart} followed by a \texttt{PQconnectPoll} loop is equivalent to \texttt{PQconnectdb}.

Note that if \texttt{PQconnectStart} returns a non-null pointer, you must call \texttt{PQfinish} when you are finished with it, in order to dispose of the structure and any associated memory blocks. This must be done even if the connection attempt fails or is abandoned.
PQconndefaults

Returns the default connection options.

```c
PQconninfoOption *PQconndefaults(void);
```

```c
typedef struct
{
    char *keyword; /* The keyword of the option */
    char *envvar; /* Fallback environment variable name */
    char *compiled; /* Fallback compiled in default value */
    char *val; /* Option’s current value, or NULL */
    char *label; /* Label for field in connect dialog */
    char *dispchar; /* Character to display for this field in a connect dialog. Values are:
                     "" Display entered value as is
                    "*" Password field - hide value
                   "D" Debug option - don’t show by default */
    int dispsize; /* Field size in characters for dialog */
} PQconninfoOption;
```

Returns a connection options array. This can be used to determine all possible `PQconnectdb` options and their current default values. The return value points to an array of `PQconninfoOption` structures, which ends with an entry having a null keyword pointer. The null pointer is returned if memory could not be allocated. Note that the current default values (val fields) will depend on environment variables and other context. Callers must treat the connection options data as read-only.

After processing the options array, free it by passing it to `PQconninfoFree`. If this is not done, a small amount of memory is leaked for each call to `PQconndefaults`.

PQfinish

Closes the connection to the server. Also frees memory used by the `PGconn` object.

```c
void PQfinish(PGconn *conn);
```

Note that even if the server connection attempt fails (as indicated by `PQstatus`), the application should call `PQfinish` to free the memory used by the `PGconn` object. The `PGconn` pointer must not be used again after `PQfinish` has been called.

PQreset

Resets the communication channel to the server.

```c
void PQreset(PGconn *conn);
```

This function will close the connection to the server and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This might be useful for error recovery if a working connection is lost.

PQresetStart
PQresetPoll

Reset the communication channel to the server, in a nonblocking manner.

```c
int PQresetStart(PGconn *conn);
PostgresPollingStatusType PQresetPoll(PGconn *conn);
```
These functions will close the connection to the server and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This can be useful for error recovery if a working connection is lost. They differ from PQreset (above) in that they act in a nonblocking manner. These functions suffer from the same restrictions as PQconnectStart and PQconnectPoll.

To initiate a connection reset, call PQresetStart. If it returns 0, the reset has failed. If it returns 1, poll the reset using PQresetPoll in exactly the same way as you would create the connection using PQconnectPoll.

### 30.2. Connection Status Functions

These functions can be used to interrogate the status of an existing database connection object.

**Tip:** libpq application programmers should be careful to maintain the PGconn abstraction. Use the accessor functions described below to get at the contents of PGconn. Reference to internal PGconn fields using libpq-int.h is not recommended because they are subject to change in the future.

The following functions return parameter values established at connection. These values are fixed for the life of the PGconn object.

- **PQdb**

  Returns the database name of the connection.

  ```c
  char *PQdb(const PGconn *conn);
  ```

- **PQuser**

  Returns the user name of the connection.

  ```c
  char *PQuser(const PGconn *conn);
  ```

- **PQpass**

  Returns the password of the connection.

  ```c
  char *PQpass(const PGconn *conn);
  ```

- **PQhost**

  Returns the server host name of the connection.

  ```c
  char *PQhost(const PGconn *conn);
  ```

- **PQport**

  Returns the port of the connection.

  ```c
  char *PQport(const PGconn *conn);
  ```
Chapter 30. libpq - C Library

PQtty

Returns the debug TTY of the connection. (This is obsolete, since the server no longer pays attention to the TTY setting, but the function remains for backwards compatibility.)

char *PQtty(const PGconn *conn);

PQoptions

Returns the command-line options passed in the connection request.

char *PQoptions(const PGconn *conn);

The following functions return status data that can change as operations are executed on the PGconn object.

PQstatus

Returns the status of the connection.

ConnStatusType PQstatus(const PGconn *conn);

The status can be one of a number of values. However, only two of these are seen outside of an asynchronous connection procedure: CONNECTION_OK and CONNECTION_BAD. A good connection to the database has the status CONNECTION_OK. A failed connection attempt is signaled by status CONNECTION_BAD. Ordinarily, an OK status will remain so until PQfinish, but a communications failure might result in the status changing to CONNECTION_BAD prematurely. In that case the application could try to recover by calling PQreset.

See the entry for PQconnectStart and PQconnectPoll with regards to other status codes that might be seen.

PQtransactionStatus

Returns the current in-transaction status of the server.

PGTransactionStatusType PQtransactionStatus(const PGconn *conn);

The status can be PQTRANS_IDLE (currently idle), PQTRANS_ACTIVE (a command is in progress), PQTRANS_INTRANS (idle, in a valid transaction block), or PQTRANS_INERROR (idle, in a failed transaction block). PQTRANS_UNKNOWN is reported if the connection is bad. PQTRANS_ACTIVE is reported only when a query has been sent to the server and not yet completed.

Caution

PQtransactionStatus will give incorrect results when using a PostgreSQL 7.3 server that has the parameter autocommit set to off. The server-side autocommit feature has been deprecated and does not exist in later server versions.

PQparameterStatus

Looks up a current parameter setting of the server.

const char *PQparameterStatus(const PGconn *conn, const char *paramName);
Certain parameter values are reported by the server automatically at connection startup or whenever their values change. PQparameterStatus can be used to interrogate these settings. It returns the current value of a parameter if known, or NULL if the parameter is not known.

Parameters reported as of the current release include server_version, server_encoding, client_encoding, is_superuser, session_authorization, DateStyle, TimeZone, integer_datetimes, and standard_conforming_strings. (server_encoding, TimeZone, and integer_datetimes were not reported by releases before 8.0; standard_conforming_strings was not reported by releases before 8.1.) Note that server_version, server_encoding and integer_datetimes cannot change after startup.

Pre-3.0-protocol servers do not report parameter settings, but libpq includes logic to obtain values for server_version and client_encoding anyway. Applications are encouraged to use PQparameterStatus rather than ad hoc code to determine these values. (Beware however that on a pre-3.0 connection, changing client_encoding via SET after connection startup will not be reflected by PQparameterStatus.) For server_version, see also PQserverVersion, which returns the information in a numeric form that is much easier to compare against.

If no value for standard_conforming_strings is reported, applications can assume it is off, that is, backslashes are treated as escapes in string literals. Also, the presence of this parameter can be taken as an indication that the escape string syntax (E'...' ) is accepted.

Although the returned pointer is declared const, it in fact points to mutable storage associated with the PGconn structure. It is unwise to assume the pointer will remain valid across queries.

PQprotocolVersion
Interrogates the frontend/backend protocol being used.

    int PQprotocolVersion(const PGconn *conn);

Applications might wish to use this to determine whether certain features are supported. Currently, the possible values are 2 (2.0 protocol), 3 (3.0 protocol), or zero (connection bad). This will not change after connection startup is complete, but it could theoretically change during a connection reset. The 3.0 protocol will normally be used when communicating with PostgreSQL 7.4 or later servers; pre-7.4 servers support only protocol 2.0. (Protocol 1.0 is obsolete and not supported by libpq.)

PQserverVersion
Returns an integer representing the backend version.

    int PQserverVersion(const PGconn *conn);

Applications might use this to determine the version of the database server they are connected to. The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 8.1.5 will be returned as 80105, and version 8.2 will be returned as 80200 (leading zeroes are not shown). Zero is returned if the connection is bad.

PQerrorMessage
Returns the error message most recently generated by an operation on the connection.

    char *PQerrorMessage(const PGconn *conn);

Nearly all libpq functions will set a message for PQerrorMessage if they fail. Note that by libpq convention, a nonempty PQerrorMessage result will include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGconn handle is passed
to \texttt{PQfinish}. The result string should not be expected to remain the same across operations on the \texttt{PGconn} structure.

\textbf{PQsocket}

Obtains the file descriptor number of the connection socket to the server. A valid descriptor will be greater than or equal to 0; a result of -1 indicates that no server connection is currently open. (This will not change during normal operation, but could change during connection setup or reset.)

\begin{verbatim}
int PQsocket(const PGconn *conn);
\end{verbatim}

\textbf{PQbackendPID}

Returns the process ID (PID) of the backend server process handling this connection.

\begin{verbatim}
int PQbackendPID(const PGconn *conn);
\end{verbatim}

The backend PID is useful for debugging purposes and for comparison to \texttt{NOTIFY} messages (which include the PID of the notifying backend process). Note that the PID belongs to a process executing on the database server host, not the local host!

\textbf{PQconnectionNeedsPassword}

Returns true (1) if the connection authentication method required a password, but none was available. Returns false (0) if not.

\begin{verbatim}
int PQconnectionNeedsPassword(const PGconn *conn);
\end{verbatim}

This function can be applied after a failed connection attempt to decide whether to prompt the user for a password.

\textbf{PQconnectionUsedPassword}

Returns true (1) if the connection authentication method used a caller-supplied password. Returns false (0) if not.

\begin{verbatim}
int PQconnectionUsedPassword(const PGconn *conn);
\end{verbatim}

This function detects whether a password supplied to the connection function was actually used. Passwords obtained from other sources (such as the \texttt{.pgpass} file) are not considered caller-supplied.

\textbf{PQgetssl}

Returns the SSL structure used in the connection, or null if SSL is not in use.

\begin{verbatim}
SSL *PQgetssl(const PGconn *conn);
\end{verbatim}

This structure can be used to verify encryption levels, check server certificates, and more. Refer to the OpenSSL documentation for information about this structure.

You must define \texttt{USE_SSL} in order to get the correct prototype for this function. Doing this will also automatically include \texttt{ssl.h} from OpenSSL.
30.3. Command Execution Functions

Once a connection to a database server has been successfully established, the functions described here are used to perform SQL queries and commands.

30.3.1. Main Functions

PQexec

Submits a command to the server and waits for the result.

```c
PGresult *PQexec(PGconn *conn, const char *command);
```

Returns a `PGresult` pointer or possibly a null pointer. A non-null pointer will generally be returned except in out-of-memory conditions or serious errors such as inability to send the command to the server. If a null pointer is returned, it should be treated like a `PGRES_FATAL_ERROR` result. Use `PQerrorMessage` to get more information about such errors.

It is allowed to include multiple SQL commands (separated by semicolons) in the command string. Multiple queries sent in a single `PQexec` call are processed in a single transaction, unless there are explicit `BEGIN/COMMIT` commands included in the query string to divide it into multiple transactions. Note however that the returned `PGresult` structure describes only the result of the last command executed from the string. Should one of the commands fail, processing of the string stops with it and the returned `PGresult` describes the error condition.

PQexecParams

Submits a command to the server and waits for the result, with the ability to pass parameters separately from the SQL command text.

```c
PGresult *PQexecParams(PGconn *conn,
                         const char *command,
                         int nParams,
                         const Oid *paramTypes,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

`PQexecParams` is like `PQexec`, but offers additional functionality: parameter values can be specified separately from the command string proper, and query results can be requested in either text or binary format. `PQexecParams` is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The function arguments are:

- `conn`
  
The connection object to send the command through.

- `command`
  
The SQL command string to be executed. If parameters are used, they are referred to in the command string as `$1`, `$2`, etc.
nParams

The number of parameters supplied; it is the length of the arrays paramTypes[],
paramValues[], paramLengths[], and paramFormats[]. (The array pointers can be
NULL when nParams is zero.)

paramTypes[]

Specifies, by OID, the data types to be assigned to the parameter symbols. If paramTypes
is NULL, or any particular element in the array is zero, the server infers a data type for the
parameter symbol in the same way it would do for an untyped literal string.

paramValues[]

Specifies the actual values of the parameters. A null pointer in this array means the corre-
sponding parameter is null; otherwise the pointer points to a zero-terminated text string (for
text format) or binary data in the format expected by the server (for binary format).

paramLengths[]

Specifies the actual data lengths of binary-format parameters. It is ignored for null param-
eters and text-format parameters. The array pointer can be null when there are no binary
parameters.

paramFormats[]

Specifies whether parameters are text (put a zero in the array entry for the correspon-
ding parameter) or binary (put a one in the array entry for the corresponding parameter). If the
array pointer is null then all parameters are presumed to be text strings.

resultFormat

Specify zero to obtain results in text format, or one to obtain results in binary format. (There
is not currently a provision to obtain different result columns in different formats, although
that is possible in the underlying protocol.)

The primary advantage of PQexecParams over PQexec is that parameter values can be separated
from the command string, thus avoiding the need for tedious and error-prone quoting and escaping.

Unlike PQexec, PQexecParams allows at most one SQL command in the given string. (There can be
semicolons in it, but not more than one nonempty command.) This is a limitation of the underlying
protocol, but has some usefulness as an extra defense against SQL-injection attacks.

Tip: Specifying parameter types via OIDs is tedious, particularly if you prefer not to hard-wire
particular OID values into your program. However, you can avoid doing so even in cases where
the server by itself cannot determine the type of the parameter, or chooses a different type than
you want. In the SQL command text, attach an explicit cast to the parameter symbol to show what
data type you will send. For example:

SELECT * FROM mytable WHERE x = $1::bigint;

This forces parameter $1 to be treated as bigint, whereas by default it would be assigned the
same type as x. Forcing the parameter type decision, either this way or by specifying a numeric
type OID, is strongly recommended when sending parameter values in binary format, because
binary format has less redundancy than text format and so there is less chance that the server
will detect a type mismatch mistake for you.
PQprepare

Submits a request to create a prepared statement with the given parameters, and waits for com-
pletion.

PGresult *PQprepare(PGconn *conn,
    const char *stmtName,
    const char *query,
    int nParams,
    const Oid *paramTypes);

PQprepare creates a prepared statement for later execution with PQexecPrepared. This fea-
ture allows commands that will be used repeatedly to be parsed and planned just once, rather than
each time they are executed. PQprepare is supported only in protocol 3.0 and later connections;
it will fail when using protocol 2.0.

The function creates a prepared statement named stmtName from the query string, which must
contain a single SQL command. stmtName can be "" to create an unnamed statement, in which
case any pre-existing unnamed statement is automatically replaced; otherwise it is an error if the
statement name is already defined in the current session. If any parameters are used, they are
referred to in the query as $1, $2, etc. nParams is the number of parameters for which types
are pre-specified in the array paramTypes[]. (The array pointer can be NULL when nParams is
zero.) paramTypes[] specifies, by OID, the data types to be assigned to the parameter symbols.
If paramTypes is NULL, or any particular element in the array is zero, the server assigns a data
type to the parameter symbol in the same way it would do for an untyped literal string. Also, the
query can use parameter symbols with numbers higher than nParams; data types will be inferred
for these symbols as well. (See PQdescribePrepared for a means to find out what data types
were inferred.)

As with PQexec, the result is normally a PGresult object whose contents indicate server-side
success or failure. A null result indicates out-of-memory or inability to send the command at all.
Use PQerrorMessage to get more information about such errors.

 Prepared statements for use with PQexecPrepared can also be created by executing SQL PRE-
pare statements. Also, although there is no libpq function for deleting a prepared statement, the
SQL DEALLOCATE statement can be used for that purpose.

PQexecPrepared

Sends a request to execute a prepared statement with given parameters, and waits for the result.

PGresult *PQexecPrepared(PGconn *conn,
    const char *stmtName,
    int nParams,
    const char * const *paramValues,
    const int *paramLengths,
    const int *paramFormats,
    int resultFormat);

PQexecPrepared is like PQexecParams, but the command to be executed is specified by nam-
ing a previously-prepared statement, instead of giving a query string. This feature allows com-
mands that will be used repeatedly to be parsed and planned just once, rather than each time
they are executed. The statement must have been prepared previously in the current session.
PQexecPrepared is supported only in protocol 3.0 and later connections; it will fail when using
protocol 2.0.

The parameters are identical to PQexecParams, except that the name of a prepared statement is
given instead of a query string, and the paramTypes[] parameter is not present (it is not needed
since the prepared statement’s parameter types were determined when it was created).
**PQdescribePrepared**

Submits a request to obtain information about the specified prepared statement, and waits for completion.

```c
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

PQdescribePrepared allows an application to obtain information about a previously prepared statement. PQdescribePrepared is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

stmtName can be "" or NULL to reference the unnamed statement, otherwise it must be the name of an existing prepared statement. On success, a PGresult with status PGRES_COMMAND_OK is returned. The functions PQnparams and PQparamtype can be applied to this PGresult to obtain information about the parameters of the prepared statement, and the functions PQnfields, PQfname, PQftype, etc provide information about the result columns (if any) of the statement.

**PQdescribePortal**

Submits a request to obtain information about the specified portal, and waits for completion.

```c
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortal allows an application to obtain information about a previously created portal. (libpq does not provide any direct access to portals, but you can use this function to inspect the properties of a cursor created with a DECLARE CURSOR SQL command.) PQdescribePortal is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

portalName can be "" or NULL to reference the unnamed portal, otherwise it must be the name of an existing portal. On success, a PGresult with status PGRES_COMMAND_OK is returned. The functions PQnfields, PQfname, PQftype, etc can be applied to the PGresult to obtain information about the result columns (if any) of the portal.

The PGresult structure encapsulates the result returned by the server. libpq application programmers should be careful to maintain the PGresult abstraction. Use the accessor functions below to get at the contents of PGresult. Avoid directly referencing the fields of the PGresult structure because they are subject to change in the future.

**PQresultStatus**

Returns the result status of the command.

```c
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus can return one of the following values:

- **PGRES_EMPTY_QUERY**
  The string sent to the server was empty.

- **PGRES_COMMAND_OK**
  Successful completion of a command returning no data.

- **PGRES_TUPLES_OK**
  Successful completion of a command returning data (such as a SELECT or SHOW).

- **PGRES_COPY_OUT**
  Copy Out (from server) data transfer started.
Chapter 30. libpq - C Library

PGRES_COPY_IN

Copy In (to server) data transfer started.

PGRES_BAD_RESPONSE

The server’s response was not understood.

PGRES_NONFATAL_ERROR

A nonfatal error (a notice or warning) occurred.

PGRES_FATAL_ERROR

A fatal error occurred.

If the result status is PGRES_TUPLES_OK, then the functions described below can be used to retrieve the rows returned by the query. Note that a SELECT command that happens to retrieve zero rows still shows PGRES_TUPLES_OK. PGRES_COMMAND_OK is for commands that can never return rows (INSERT, UPDATE, etc.). A response of PGRES_EMPTY_QUERY might indicate a bug in the client software.

A result of status PGRES_NONFATAL_ERROR will never be returned directly by PQexec or other query execution functions; results of this kind are instead passed to the notice processor (see Section 30.11).

PQresStatus

Converts the enumerated type returned by PQresultStatus into a string constant describing the status code. The caller should not free the result.

    char *PQresStatus(ExecStatusType status);

PQresultErrorMessage

Returns the error message associated with the command, or an empty string if there was no error.

    char *PQresultErrorMessage(const PGresult *res);

If there was an error, the returned string will include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear. Immediately following a PQexec or PQgetResult call, PQerrorMessage (on the connection) will return the same string as PQresultErrorMessage (on the result). However, a PGresult will retain its error message until destroyed, whereas the connection’s error message will change when subsequent operations are done. Use PQresultErrorMessage when you want to know the status associated with a particular PGresult; use PQerrorMessage when you want to know the status from the latest operation on the connection.

PQresultErrorField

Returns an individual field of an error report.

    char *PQresultErrorField(const PGresult *res, int fieldcode);

fieldcode is an error field identifier; see the symbols listed below. NULL is returned if the PGresult is not an error or warning result, or does not include the specified field. Field values will normally not include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

The following field codes are available:
Chapter 30. libpq - C Library

PG_DIAG_SEVERITY

The severity; the field contents are ERROR, FATAL, or PANIC (in an error message), or WARNING, NOTICE, DEBUG, INFO, or LOG (in a notice message), or a localized translation of one of these. Always present.

PG_DIAG_SQLSTATE

The SQLSTATE code for the error. The SQLSTATE code identifies the type of error that has occurred; it can be used by front-end applications to perform specific operations (such as error handling) in response to a particular database error. For a list of the possible SQLSTATE codes, see Appendix A. This field is not localizable, and is always present.

PG_DIAG_MESSAGE_PRIMARY

The primary human-readable error message (typically one line). Always present.

PG_DIAG_MESSAGE_DETAIL

Detail: an optional secondary error message carrying more detail about the problem. Might run to multiple lines.

PG_DIAG_MESSAGE_HINT

Hint: an optional suggestion what to do about the problem. This is intended to differ from detail in that it offers advice (potentially inappropriate) rather than hard facts. Might run to multiple lines.

PG_DIAG_STATEMENT_POSITION

A string containing a decimal integer indicating an error cursor position as an index into the original statement string. The first character has index 1, and positions are measured in characters not bytes.

PG_DIAG_INTERNAL_POSITION

This is defined the same as the PG_DIAG_STATEMENT_POSITION field, but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client. The PG_DIAG_INTERNAL_QUERY field will always appear when this field appears.

PG_DIAG_INTERNAL_QUERY

The text of a failed internally-generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

PG_DIAG_CONTEXT

An indication of the context in which the error occurred. Presently this includes a call stack traceback of active procedural language functions and internally-generated queries. The trace is one entry per line, most recent first.

PG_DIAG_SOURCE_FILE

The file name of the source-code location where the error was reported.

PG_DIAG_SOURCE_LINE

The line number of the source-code location where the error was reported.

PG_DIAG_SOURCE_FUNCTION

The name of the source-code function reporting the error.
The client is responsible for formatting displayed information to meet its needs; in particular it should break long lines as needed. Newline characters appearing in the error message fields should be treated as paragraph breaks, not line breaks.

Errors generated internally by libpq will have severity and primary message, but typically no other fields. Errors returned by a pre-3.0-protocol server will include severity and primary message, and sometimes a detail message, but no other fields.

Note that error fields are only available from PGresult objects, not PGconn objects; there is no PQerrorField function.

PQclear

Frees the storage associated with a PGresult. Every command result should be freed via PQclear when it is no longer needed.

void PQclear(PGresult *res);

You can keep a PGresult object around for as long as you need it; it does not go away when you issue a new command, nor even if you close the connection. To get rid of it, you must call PQclear. Failure to do this will result in memory leaks in your application.

PQmakeEmptyPGresult

Constructs an empty PGresult object with the given status.

PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);

This is libpq’s internal function to allocate and initialize an empty PGresult object. This function returns NULL if memory could not be allocated. It is exported because some applications find it useful to generate result objects (particularly objects with error status) themselves. If conn is not null and status indicates an error, the current error message of the specified connection is copied into the PGresult. Note that PQclear should eventually be called on the object, just as with a PGresult returned by libpq itself.

### 30.3.2. Retrieving Query Result Information

These functions are used to extract information from a PGresult object that represents a successful query result (that is, one that has status PGRES_TUPLES_OK). They can also be used to extract information from a successful Describe operation: a Describe’s result has all the same column information that actual execution of the query would provide, but it has zero rows. For objects with other status values, these functions will act as though the result has zero rows and zero columns.

PQntuples

Returns the number of rows (tuples) in the query result. Because it returns an integer result, large result sets might overflow the return value on 32-bit operating systems.

int PQntuples(const PGresult *res);

PQnfields

Returns the number of columns (fields) in each row of the query result.

int PQnfields(const PGresult *res);
Chapter 30. libpq - C Library

PQfname

Returns the column name associated with the given column number. Column numbers start at 0. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

    char *PQfname(const PGresult *res,
                   int column_number);

NULL is returned if the column number is out of range.

PQfnumber

Returns the column number associated with the given column name.

    int PQfnumber(const PGresult *res,
                   const char *column_name);

-1 is returned if the given name does not match any column.

The given name is treated like an identifier in an SQL command, that is, it is downcased unless double-quoted. For example, given a query result generated from the SQL command:

    SELECT 1 AS FOO, 2 AS "BAR";

we would have the results:

    PQfname(res, 0)  foo
    PQfname(res, 1)  BAR
    PQfnumber(res, "FOO")  0
    PQfnumber(res, "foo")  0
    PQfnumber(res, "BAR") -1
    PQfnumber(res, "\"BAR\"")  1

PQftable

Returns the OID of the table from which the given column was fetched. Column numbers start at 0.

    Oid PQftable(const PGresult *res,
                 int column_number);

InvalidOid is returned if the column number is out of range, or if the specified column is not a simple reference to a table column, or when using pre-3.0 protocol. You can query the system table pg_class to determine exactly which table is referenced.

The type Oid and the constant InvalidOid will be defined when you include the libpq header file. They will both be some integer type.

PQftablecol

Returns the column number (within its table) of the column making up the specified query result column. Query-result column numbers start at 0, but table columns have nonzero numbers.

    int PQftablecol(const PGresult *res,
                    int column_number);

Zero is returned if the column number is out of range, or if the specified column is not a simple reference to a table column, or when using pre-3.0 protocol.
PQfformat

Returns the format code indicating the format of the given column. Column numbers start at 0.

\[
\text{int PQfformat(const PGresult \*res,} \\
\text{int column_number);} \\
\]

Format code zero indicates textual data representation, while format code one indicates binary representation. (Other codes are reserved for future definition.)

PQftype

Returns the data type associated with the given column number. The integer returned is the internal OID number of the type. Column numbers start at 0.

\[
\text{Oid PQftype(const PGresult \*res,} \\
\text{int column_number);} \\
\]

You can query the system table pg_type to obtain the names and properties of the various data types. The OIDs of the built-in data types are defined in the file src/include/catalog/pg_type.h in the source tree.

PQfmod

Returns the type modifier of the column associated with the given column number. Column numbers start at 0.

\[
\text{int PQfmod(const PGresult \*res,} \\
\text{int column_number);} \\
\]

The interpretation of modifier values is type-specific; they typically indicate precision or size limits. The value -1 is used to indicate "no information available". Most data types do not use modifiers, in which case the value is always -1.

PQfsize

Returns the size in bytes of the column associated with the given column number. Column numbers start at 0.

\[
\text{int PQfsize(const PGresult \*res,} \\
\text{int column_number);} \\
\]

PQfsize returns the space allocated for this column in a database row, in other words the size of the server’s internal representation of the data type. (Accordingly, it is not really very useful to clients.) A negative value indicates the data type is variable-length.

PQbinaryTuples

Returns 1 if the PGresult contains binary data and 0 if it contains text data.

\[
\text{int PQbinaryTuples(const PGresult \*res);} \\
\]

This function is deprecated (except for its use in connection with COPY), because it is possible for a single PGresult to contain text data in some columns and binary data in others. PQfformat is preferred. PQbinaryTuples returns 1 only if all columns of the result are binary (format 1).
PQgetvalue

Returns a single field value of one row of a PGresult. Row and column numbers start at 0. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

```c
char *PQgetvalue(const PGresult *res,
                  int row_number,
                  int column_number);
```

For data in text format, the value returned by PQgetvalue is a null-terminated character string representation of the field value. For data in binary format, the value is in the binary representation determined by the data type's typsend and typreceive functions. (The value is actually followed by a zero byte in this case too, but that is not ordinarily useful, since the value is likely to contain embedded nulls.)

An empty string is returned if the field value is null. See PQgetisnull to distinguish null values from empty-string values.

The pointer returned by PQgetvalue points to storage that is part of the PGresult structure. One should not modify the data it points to, and one must explicitly copy the data into other storage if it is to be used past the lifetime of the PGresult structure itself.

PQgetisnull

Tests a field for a null value. Row and column numbers start at 0.

```c
int PQgetisnull(const PGresult *res,
                 int row_number,
                 int column_number);
```

This function returns 1 if the field is null and 0 if it contains a non-null value. (Note that PQgetvalue will return an empty string, not a null pointer, for a null field.)

PQgetlength

Returns the actual length of a field value in bytes. Row and column numbers start at 0.

```c
int PQgetlength(const PGresult *res,
                 int row_number,
                 int column_number);
```

This is the actual data length for the particular data value, that is, the size of the object pointed to by PQgetvalue. For text data format this is the same as strlen(). For binary format this is essential information. Note that one should not rely on PQfsize to obtain the actual data length.

PQnparams

Returns the number of parameters of a prepared statement.

```c
int PQnparams(const PGresult *res);
```

This function is only useful when inspecting the result of PQdescribePrepared. For other types of queries it will return zero.

PQparamtype

Returns the data type of the indicated statement parameter. Parameter numbers start at 0.

```c
Oid PQparamtype(const PGresult *res, int param_number);
```
Chapter 30. libpq - C Library

This function is only useful when inspecting the result of PQdescribePrepared. For other
types of queries it will return zero.

PQprint

Prints out all the rows and, optionally, the column names to the specified output stream.

```
void PQprint(FILE *fout, /* output stream */
            const PGresult *res,
            const PQprintOpt *po);
```

```c
typedef struct {
    pqbool header;  /* print output field headings and row count */
    pqbool align;   /* fill align the fields */
    pqbool standard; /* old brain dead format */
    pqbool html3;   /* output HTML tables */
    pqbool expanded; /* expand tables */
    pqbool pager;   /* use pager for output if needed */
    char *fieldSep; /* field separator */
    char *tableOpt; /* attributes for HTML table element */
    char *caption;  /* HTML table caption */
    char **fieldName; /* null-terminated array of replacement field names */
} PQprintOpt;
```

This function was formerly used by psql to print query results, but this is no longer the case.
Note that it assumes all the data is in text format.

30.3.3. Retrieving Result Information for Other Commands

These functions are used to extract information from PGresult objects that are not SELECT results.

PQcmdStatus

Returns the command status tag from the SQL command that generated the PGresult.

```
char *PQcmdStatus(PGresult *res);
```

Commonly this is just the name of the command, but it might include additional data such as the
number of rows processed. The caller should not free the result directly. It will be freed when the
associated PGresult handle is passed to PQclear.

PQcmdTuples

Returns the number of rows affected by the SQL command.

```
char *PQcmdTuples(PGresult *res);
```

This function returns a string containing the number of rows affected by the SQL statement
that generated the PGresult. This function can only be used following the execution of an
INSERT, UPDATE, DELETE, MOVE, FETCH, or COPY statement, or an EXECUTE of a prepared
query that contains an INSERT, UPDATE, or DELETE statement. If the command that generated
the PGresult was anything else, PQcmdTuples returns an empty string. The caller should not
free the return value directly. It will be freed when the associated PGresult handle is passed to
PQclear.
Chapter 30. libpq - C Library

**PQoidValue**

Returns the OID of the inserted row, if the SQL command was an `INSERT` that inserted exactly one row into a table that has OIDs, or a `EXECUTE` of a prepared query containing a suitable `INSERT` statement. Otherwise, this function returns `InvalidOid`. This function will also return `InvalidOid` if the table affected by the `INSERT` statement does not contain OIDs.

```c
Oid PQoidValue(const PGresult *res);
```

**PQoidStatus**

Returns a string with the OID of the inserted row, if the SQL command was an `INSERT` that inserted exactly one row, or a `EXECUTE` of a prepared statement consisting of a suitable `INSERT`. (The string will be 0 if the `INSERT` did not insert exactly one row, or if the target table does not have OIDs.) If the command was not an `INSERT`, returns an empty string.

```c
char *PQoidStatus(const PGresult *res);
```

This function is deprecated in favor of `PQoidValue`. It is not thread-safe.

### 30.3.4. Escaping Strings for Inclusion in SQL Commands

`PQescapeStringConn` escapes a string for use within an SQL command. This is useful when inserting data values as literal constants in SQL commands. Certain characters (such as quotes and backslashes) must be escaped to prevent them from being interpreted specially by the SQL parser. `PQescapeStringConn` performs this operation.

**Tip:** It is especially important to do proper escaping when handling strings that were received from an untrustworthy source. Otherwise there is a security risk: you are vulnerable to “SQL injection” attacks wherein unwanted SQL commands are fed to your database.

Note that it is not necessary nor correct to do escaping when a data value is passed as a separate parameter in `PQexecParams` or its sibling routines.

```c
size_t PQescapeStringConn (PGconn *conn, char *to, const char *from, size_t length, int *error);
```

`PQescapeStringConn` writes an escaped version of the `from` string to the `to` buffer, escaping special characters so that they cannot cause any harm, and adding a terminating zero byte. The single quotes that must surround PostgreSQL string literals are not included in the result string; they should be provided in the SQL command that the result is inserted into. The parameter `from` points to the first character of the string that is to be escaped, and the `length` parameter gives the number of bytes in this string. A terminating zero byte is not required, and should not be counted in `length`. (If a terminating zero byte is found before `length` bytes are processed, `PQescapeStringConn` stops at the zero; the behavior is thus rather like `strncpy`) to shall point to a buffer that is able to hold at least one more byte than twice the value of `length`, otherwise the behavior is undefined. Behavior is likewise undefined if the `to` and `from` strings overlap.

If the `error` parameter is not NULL, then `*error` is set to zero on success, nonzero on error. Presently the only possible error conditions involve invalid multibyte encoding in the source string.
The output string is still generated on error, but it can be expected that the server will reject it as malformed. On error, a suitable message is stored in the `conn` object, whether or not `error` is NULL. 
PQescapeStringConn returns the number of bytes written to `to`, not including the terminating zero byte.

```c
size_t PQescapeString (char *to, const char *from, size_t length);
```

PQescapeString is an older, deprecated version of PQescapeStringConn; the difference is that it does not take `conn` or `error` parameters. Because of this, it cannot adjust its behavior depending on the connection properties (such as character encoding) and therefore it might give the wrong results. Also, it has no way to report error conditions.
PQescapeString can be used safely in single-threaded client programs that work with only one PostgreSQL connection at a time (in this case it can find out what it needs to know “behind the scenes”). In other contexts it is a security hazard and should be avoided in favor of PQescapeStringConn.

### 30.3.5. Escaping Binary Strings for Inclusion in SQL Commands

PQescapeByteaConn

Escapes binary data for use within an SQL command with the type `bytea`. As with PQescapeStringConn, this is only used when inserting data directly into an SQL command string.

```c
unsigned char *PQescapeByteaConn (PGconn *conn, const unsigned char *from, size_t from_length, size_t *to_length);
```

Certain byte values must be escaped (but all byte values can be escaped) when used as part of a `bytea` literal in an SQL statement. In general, to escape a byte, it is converted into the three digit octal number equal to the octet value, and preceded by usually two backslashes. The single quote (’) and backslash (\) characters have special alternative escape sequences. See Section 8.4 for more information. PQescapeByteaConn performs this operation, escaping only the minimally required bytes.

The `from` parameter points to the first byte of the string that is to be escaped, and the `from_length` parameter gives the number of bytes in this binary string. (A terminating zero byte is neither necessary nor counted.) The `to_length` parameter points to a variable that will hold the resultant escaped string length. This result string length includes the terminating zero byte of the result.

PQescapeByteaConn returns an escaped version of the `from` parameter binary string in memory allocated with `malloc()`. This memory must be freed using `PQfreemem()` when the result is no longer needed. The return string has all special characters replaced so that they can be properly processed by the PostgreSQL string literal parser, and the `bytea` input function. A terminating zero byte is also added. The single quotes that must surround PostgreSQL string literals are not part of the result string.
On error, a NULL pointer is returned, and a suitable error message is stored in the conn object. Currently, the only possible error is insufficient memory for the result string.

**PQescapeBytea**

*PQescapeBytea is an older, deprecated version of PQescapeByteaConn.*

```c
unsigned char *PQescapeBytea(const unsigned char *from, size_t from_length, size_t *to_length);
```

The only difference from PQescapeByteaConn is that PQescapeBytea does not take a PGconn parameter. Because of this, it cannot adjust its behavior depending on the connection properties (in particular, whether standard-conforming strings are enabled) and therefore it might give the wrong results. Also, it has no way to return an error message on failure.

PQescapeBytea can be used safely in single-threaded client programs that work with only one PostgreSQL connection at a time (in this case it can find out what it needs to know “behind the scenes”). In other contexts it is a security hazard and should be avoided in favor of PQescapeByteaConn.

**PQunescapeBytea**

Converts a string representation of binary data into binary data — the reverse of PQescapeBytea. This is needed when retrieving bytea data in text format, but not when retrieving it in binary format.

```c
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

The from parameter points to a string such as might be returned by PQgetvalue when applied to a bytea column. PQunescapeBytea converts this string representation into its binary representation. It returns a pointer to a buffer allocated with malloc(), or null on error, and puts the size of the buffer in to_length. The result must be freed using PQfreemem when it is no longer needed.

This conversion is not exactly the inverse of PQescapeBytea, because the string is not expected to be “escaped” when received from PQgetvalue. In particular this means there is no need for string quoting considerations, and so no need for a PGconn parameter.

**PQfreemem**

Frees memory allocated by libpq.

```c
void PQfreemem(void *ptr);
```

Frees memory allocated by libpq, particularly PQescapeByteaConn, PQescapeBytea, PQunescapeBytea, and PQnotifies. It is particularly important that this function, rather than free(), be used on Microsoft Windows. This is because allocating memory in a DLL and releasing it in the application works only if multithreaded/single-threaded, release/debug, and static/dynamic flags are the same for the DLL and the application. On non-Microsoft Windows platforms, this function is the same as the standard library function free().
30.4. Asynchronous Command Processing

The PQexec function is adequate for submitting commands in normal, synchronous applications. It has a couple of deficiencies, however, that can be of importance to some users:

- **PQexec** waits for the command to be completed. The application might have other work to do (such as maintaining a user interface), in which case it won’t want to block waiting for the response.
- Since the execution of the client application is suspended while it waits for the result, it is hard for the application to decide that it would like to try to cancel the ongoing command. (It can be done from a signal handler, but not otherwise.)
- PQexec can return only one PGresult structure. If the submitted command string contains multiple SQL commands, all but the last PGresult are discarded by PQexec.

Applications that do not like these limitations can instead use the underlying functions that PQexec is built from: PQsendQuery and PQgetResult. There are also PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared, and PQsendDescribePortal, which can be used with PQgetResult to duplicate the functionality of PQexecParams, PQprepare, PQexecPrepared, PQdescribePrepared, and PQdescribePortal respectively.

**PQsendQuery**

Submits a command to the server without waiting for the result(s). 1 is returned if the command was successfully dispatched and 0 if not (in which case, use PQerrorMessage to get more information about the failure).

```c
int PQsendQuery(PGconn *conn, const char *command);
```

After successfully calling **PQsendQuery**, call **PQgetResult** one or more times to obtain the results. **PQsendQuery** cannot be called again (on the same connection) until **PQgetResult** has returned a null pointer, indicating that the command is done.

**PQsendQueryParams**

Submits a command and separate parameters to the server without waiting for the result(s).

```c
int PQsendQueryParams(PGconn *conn, const char *command, int nParams, const Oid *paramTypes, const char * const *paramValues, const int *paramLengths, const int *paramFormats, int resultFormat);
```

This is equivalent to **PQsendQuery** except that query parameters can be specified separately from the query string. The function’s parameters are handled identically to **PQexecParams**. Like **PQexecParams**, it will not work on 2.0-protocol connections, and it allows only one command in the query string.

**PQsendPrepare**

Sends a request to create a prepared statement with the given parameters, without waiting for completion.

```c
int PQsendPrepare(PGconn *conn,`
const char *stmtName,
const char *query,
int nParams,
const Oid *paramTypes);  
This is an asynchronous version of PQprepare: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call PQgetResult to determine whether the server successfully created the prepared statement. The function’s parameters are handled identically to PQprepare. Like PQprepare, it will not work on 2.0-protocol connections.

PQsendQueryPrepared
Sends a request to execute a prepared statement with given parameters, without waiting for the result(s).
  
int PQsendQueryPrepared(PGconn *conn,
    const char *stmtName,
    int nParams,
    const char * const *paramValues,
    const int *paramLengths,
    const int *paramFormats,
    int resultFormat);  
This is similar to PQsendQueryParams, but the command to be executed is specified by naming a previously-prepared statement, instead of giving a query string. The function’s parameters are handled identically to PQexecPrepared. Like PQexecPrepared, it will not work on 2.0-protocol connections.

PQsendDescribePrepared
Submits a request to obtain information about the specified prepared statement, without waiting for completion.
  
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);  
This is an asynchronous version of PQdescribePrepared: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call PQgetResult to obtain the results. The function’s parameters are handled identically to PQdescribePrepared. Like PQdescribePrepared, it will not work on 2.0-protocol connections.

PQsendDescribePortal
Submits a request to obtain information about the specified portal, without waiting for completion.
  
int PQsendDescribePortal(PGconn *conn, const char *portalName);  
This is an asynchronous version of PQdescribePortal: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call PQgetResult to obtain the results. The function’s parameters are handled identically to PQdescribePortal. Like PQdescribePortal, it will not work on 2.0-protocol connections.

PQgetResult
Waits for the next result from a prior PQsendQuery, PQsendQueryParams, PQsendPrepare, or PQsendQueryPrepared call, and returns it. A null pointer is returned when the command is complete and there will be no more results.
  
PGresult *PQgetResult(PGconn *conn);
**PQgetResult** must be called repeatedly until it returns a null pointer, indicating that the command is done. (If called when no command is active, **PQgetResult** will just return a null pointer at once.) Each non-null result from **PQgetResult** should be processed using the same **PGresult** accessor functions previously described. Don’t forget to free each result object with **PQclear** when done with it. Note that **PQgetResult** will block only if a command is active and the necessary response data has not yet been read by **PQconsumeInput**.

Using **PQsendQuery** and **PQgetResult** solves one of **PQexec**’s problems: If a command string contains multiple SQL commands, the results of those commands can be obtained individually. (This allows a simple form of overlapped processing, by the way: the client can be handling the results of one command while the server is still working on later queries in the same command string.) However, calling **PQgetResult** will still cause the client to block until the server completes the next SQL command. This can be avoided by proper use of two more functions:

**PQconsumeInput**

If input is available from the server, consume it.

```c
int PQconsumeInput(PGconn *conn);
```

**PQconsumeInput** normally returns 1 indicating “no error”, but returns 0 if there was some kind of trouble (in which case **PQerrorMessage** can be consulted). Note that the result does not say whether any input data was actually collected. After calling **PQconsumeInput**, the application can check **PQisBusy** and/or **PQnotifies** to see if their state has changed.

**PQconsumeInput** can be called even if the application is not prepared to deal with a result or notification just yet. The function will read available data and save it in a buffer, thereby causing a **select()** read-ready indication to go away. The application can thus use **PQconsumeInput** to clear the **select()** condition immediately, and then examine the results at leisure.

**PQisBusy**

Returns 1 if a command is busy, that is, **PQgetResult** would block waiting for input. A 0 return indicates that **PQgetResult** can be called with assurance of not blocking.

```c
int PQisBusy(PGconn *conn);
```

**PQisBusy** will not itself attempt to read data from the server; therefore **PQconsumeInput** must be invoked first, or the busy state will never end.

A typical application using these functions will have a main loop that uses **select()** or **poll()** to wait for all the conditions that it must respond to. One of the conditions will be input available from the server, which in terms of **select()** means readable data on the file descriptor identified by **PQsocket**. When the main loop detects input ready, it should call **PQconsumeInput** to read the input. It can then call **PQisBusy**, followed by **PQgetResult** if **PQisBusy** returns false (0). It can also call **PQnotifies** to detect **NOTIFY** messages (see Section 30.7).

A client that uses **PQsendQuery/PQgetResult** can also attempt to cancel a command that is still being processed by the server; see Section 30.5. But regardless of the return value of **PQcancel**, the application must continue with the normal result-reading sequence using **PQgetResult**. A successful cancellation will simply cause the command to terminate sooner than it would have otherwise.

By using the functions described above, it is possible to avoid blocking while waiting for input from the database server. However, it is still possible that the application will block waiting to send output to the server. This is relatively uncommon but can happen if very long SQL commands or data values are
sent. (It is much more probable if the application sends data via \COPY IN, however.) To prevent this possibility and achieve completely nonblocking database operation, the following additional functions can be used.

**PQsetnonblocking**

Sets the nonblocking status of the connection.

```c
int PQsetnonblocking(PGconn *conn, int arg);
```

Sets the state of the connection to nonblocking if `arg` is 1, or blocking if `arg` is 0. Returns 0 if OK, -1 if error.

In the nonblocking state, calls to `PQsendQuery`, `PQputline`, `PQputnbytes`, and `PQendcopy` will not block but instead return an error if they need to be called again.

Note that `PQexec` does not honor nonblocking mode; if it is called, it will act in blocking fashion anyway.

**PQisnonblocking**

Returns the blocking status of the database connection.

```c
int PQisnonblocking(const PGconn *conn);
```

Returns 1 if the connection is set to nonblocking mode and 0 if blocking.

**PQflush**

Attempts to flush any queued output data to the server. Returns 0 if successful (or if the send queue is empty), -1 if it failed for some reason, or 1 if it was unable to send all the data in the send queue yet (this case can only occur if the connection is nonblocking).

```c
int PQflush(PGconn *conn);
```

After sending any command or data on a nonblocking connection, call `PQflush`. If it returns 1, wait for the socket to be write-ready and call it again; repeat until it returns 0. Once `PQflush` returns 0, wait for the socket to be read-ready and then read the response as described above.

### 30.5. Cancelling Queries in Progress

A client application can request cancellation of a command that is still being processed by the server, using the functions described in this section.

**PQgetCancel**

Creates a data structure containing the information needed to cancel a command issued through a particular database connection.

```c
PGcancel *PQgetCancel(PGconn *conn);
```

`PQgetCancel` creates a `PGcancel` object given a `PGconn` connection object. It will return NULL if the given `conn` is NULL or an invalid connection. The `PGcancel` object is an opaque structure that is not meant to be accessed directly by the application; it can only be passed to `PQcancel` or `PQfreeCancel`. 

---

**503**
Chapter 30. libpq - C Library

PQfreeCancel

Frees a data structure created by PQgetCancel.

```c
void PQfreeCancel(PGcancel *cancel);
```

PQfreeCancel frees a data object previously created by PQgetCancel.

PQcancel

Requests that the server abandon processing of the current command.

```c
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

The return value is 1 if the cancel request was successfully dispatched and 0 if not. If not, `errbuf` is filled with an error message explaining why not. `errbuf` must be a char array of size `errbufsize` (the recommended size is 256 bytes).

Successful dispatch is no guarantee that the request will have any effect, however. If the cancellation is effective, the current command will terminate early and return an error result. If the cancellation fails (say, because the server was already done processing the command), then there will be no visible result at all.

PQcancel can safely be invoked from a signal handler, if the `errbuf` is a local variable in the signal handler. The `PGcancel` object is read-only as far as PQcancel is concerned, so it can also be invoked from a thread that is separate from the one manipulating the `PGconn` object.

PQrequestCancel

Requests that the server abandon processing of the current command.

```c
int PQrequestCancel(PGconn *conn);
```

PQrequestCancel is a deprecated variant of PQcancel. It operates directly on the `PGconn` object, and in case of failure stores the error message in the `PGconn` object (whence it can be retrieved by PQerrorMessage). Although the functionality is the same, this approach creates hazards for multiple-thread programs and signal handlers, since it is possible that overwriting the `PGconn`’s error message will mess up the operation currently in progress on the connection.

30.6. The Fast-Path Interface

PostgreSQL provides a fast-path interface to send simple function calls to the server.

**Tip:** This interface is somewhat obsolete, as one can achieve similar performance and greater functionality by setting up a prepared statement to define the function call. Then, executing the statement with binary transmission of parameters and results substitutes for a fast-path function call.

The function PQfn requests execution of a server function via the fast-path interface:

```c
PGresult *PQfn(PGconn *conn,  
   int fnid,  
   int *result_buf,  
   int *result_len,  
   ...)  
```
typedef struct {
    int len;
    int isint;
    union {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;

The fnid argument is the OID of the function to be executed. args and nargs define the parameters to be passed to the function; they must match the declared function argument list. When the isint field of a parameter structure is true, the u.integer value is sent to the server as an integer of the indicated length (this must be 1, 2, or 4 bytes); proper byte-swapping occurs. When isint is false, the indicated number of bytes at *u.ptr are sent with no processing; the data must be in the format expected by the server for binary transmission of the function’s argument data type. result_buf is the buffer in which to place the return value. The caller must have allocated sufficient space to store the return value. (There is no check!) The actual result length will be returned in the integer pointed to by result_len. If a 1, 2, or 4-byte integer result is expected, set result_is_int to 1, otherwise set it to 0. Setting result_is_int to 1 causes libpq to byte-swap the value if necessary, so that it is delivered as a proper int value for the client machine. When result_is_int is 0, the binary-format byte string sent by the server is returned unmodified.

PQfn always returns a valid PGresult pointer. The result status should be checked before the result is used. The caller is responsible for freeing the PGresult with PQclear when it is no longer needed.

Note that it is not possible to handle null arguments, null results, nor set-valued results when using this interface.

30.7. Asynchronous Notification

PostgreSQL offers asynchronous notification via the LISTEN and NOTIFY commands. A client session registers its interest in a particular notification condition with the LISTEN command (and can stop listening with the UNLISTEN command). All sessions listening on a particular condition will be notified asynchronously when a NOTIFY command with that condition name is executed by any session. No additional information is passed from the notifier to the listener. Thus, typically, any actual data that needs to be communicated is transferred through a database table. Commonly, the condition name is the same as the associated table, but it is not necessary for there to be any associated table.

libpq applications submit LISTEN and UNLISTEN commands as ordinary SQL commands. The arrival of NOTIFY messages can subsequently be detected by calling PQnotifies.

The function PQnotifies returns the next notification from a list of unhandled notification messages received from the server. It returns a null pointer if there are no pending notifications. Once a notification is returned from PQnotifies, it is considered handled and will be removed from the list of notifications.

PQnotifies(PQconn *conn);
typedef struct pgNotify {
    char *relname; /* notification condition name */
    int be_pid;    /* process ID of notifying server process */
    char *extra;  /* notification parameter */
} PGnotify;

After processing a PGnotify object returned by PQnotifies, be sure to free it with PQfreemem. It is sufficient to free the PGnotify pointer; the relname and extra fields do not represent separate allocations. (At present, the extra field is unused and will always point to an empty string.)

Example 30-2 gives a sample program that illustrates the use of asynchronous notification.

PQnotifies does not actually read data from the server; it just returns messages previously absorbed by another libpq function. In prior releases of libpq, the only way to ensure timely receipt of NOTIFY messages was to constantly submit commands, even empty ones, and then check PQnotifies after each PQexec. While this still works, it is deprecated as a waste of processing power.

A better way to check for NOTIFY messages when you have no useful commands to execute is to call PQconsumeInput, then check PQnotifies. You can use select() to wait for data to arrive from the server, thereby using no CPU power unless there is something to do. (See PQsocket to obtain the file descriptor number to use with select().) Note that this will work OK whether you submit commands with PQsendQuery/PQgetResult or simply use PQexec. You should, however, remember to check PQnotifies after each PQgetResult or PQexec, to see if any notifications came in during the processing of the command.

### 30.8. Functions Associated with the COPY Command

The COPY command in PostgreSQL has options to read from or write to the network connection used by libpq. The functions described in this section allow applications to take advantage of this capability by supplying or consuming copied data.

The overall process is that the application first issues the SQL COPY command via PQexec or one of the equivalent functions. The response to this (if there is no error in the command) will be a PGresult object bearing a status code of PGRES_COPY_OUT or PGRES_COPY_IN (depending on the specified copy direction). The application should then use the functions of this section to receive or transmit data rows. When the data transfer is complete, another PGresult object is returned to indicate success or failure of the transfer. Its status will be PGRES_COMMAND_OK for success or PGRES_FATAL_ERROR if some problem was encountered. At this point further SQL commands can be issued via PQexec. (It is not possible to execute other SQL commands using the same connection while the COPY operation is in progress.)

If a COPY command is issued via PQexec in a string that could contain additional commands, the application must continue fetching results via PQgetResult after completing the COPY sequence. Only when PQgetResult returns NULL is it certain that the PQexec command string is done and it is safe to issue more commands.

The functions of this section should be executed only after obtaining a result status of PGRES_COPY_OUT or PGRES_COPY_IN from PQexec or PQgetResult.

A PGresult object bearing one of these status values carries some additional data about the COPY operation that is starting. This additional data is available using functions that are also used in connection with query results:
Chapter 30. libpq - C Library

PQnfields

Returns the number of columns (fields) to be copied.

PQbinaryTuples

0 indicates the overall copy format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary. See COPY for more information.

PQfformat

Returns the format code (0 for text, 1 for binary) associated with each column of the copy operation. The per-column format codes will always be zero when the overall copy format is textual, but the binary format can support both text and binary columns. (However, as of the current implementation of COPY, only binary columns appear in a binary copy; so the per-column formats always match the overall format at present.)

Note: These additional data values are only available when using protocol 3.0. When using protocol 2.0, all these functions will return 0.

30.8.1. Functions for Sending COPY Data

These functions are used to send data during COPY FROM STDIN. They will fail if called when the connection is not in COPY_IN state.

PQputCopyData

Sends data to the server during COPY_IN state.

```c
int PQputCopyData(PGconn *conn, const char *buffer, int nbytes);
```

Transmits the COPY data in the specified buffer, of length nbytes, to the server. The result is 1 if the data was sent, zero if it was not sent because the attempt would block (this case is only possible if the connection is in nonblocking mode), or -1 if an error occurred. (Use PQerrorMessage to retrieve details if the return value is -1. If the value is zero, wait for write-ready and try again.)

The application can divide the COPY data stream into buffer loads of any convenient size. Buffer-load boundaries have no semantic significance when sending. The contents of the data stream must match the data format expected by the COPY command; see COPY for details.

PQputCopyEnd

Sends end-of-data indication to the server during COPY_IN state.

```c
int PQputCopyEnd(PGconn *conn, const char *errmsg);
```

Ends the COPY_IN operation successfully if errmsg is NULL. If errmsg is not NULL then the COPY is forced to fail, with the string pointed to by errmsg used as the error message. (One should not assume that this exact error message will come back from the server, however,
as the server might have already failed the COPY for its own reasons. Also note that the option to
force failure does not work when using pre-3.0-protocol connections.)

The result is 1 if the termination data was sent, zero if it was not sent because the attempt would
block (this case is only possible if the connection is in nonblocking mode), or -1 if an error
occurred. (Use PQerrorMessage to retrieve details if the return value is -1. If the value is zero,
wait for write-ready and try again.)

After successfully calling PQputCopyEnd, call PQgetResult to obtain the final result status of
the COPY command. One can wait for this result to be available in the usual way. Then return to
normal operation.

### 30.8.2. Functions for Receiving COPY Data

These functions are used to receive data during COPY TO STDOUT. They will fail if called when the
collection is not in COPY_OUT state.

**PQgetCopyData**

Receives data from the server during COPY_OUT state.

```c
int PQgetCopyData(PGconn *conn,
                 char **buffer,
                 int async);
```

Attempts to obtain another row of data from the server during a COPY. Data is always returned
one row at a time; if only a partial row is available, it is not returned. Successful return of a
data row involves allocating a chunk of memory to hold the data. The buffer parameter must be
non-NULL. *buffer is set to point to the allocated memory, or to NULL in cases where no buffer
is returned. A non-NULL result buffer must be freed using PQfreemem when no longer needed.

When a row is successfully returned, the return value is the number of data bytes in the row (this
will always be greater than zero). The returned string is always null-terminated, though this is
probably only useful for textual COPY. A result of zero indicates that the COPY is still in progress,
but no row is yet available (this is only possible when async is true). A result of -1 indicates that
the COPY is done. A result of -2 indicates that an error occurred (consult PQerrorMessage for
the reason).

When async is true (not zero), PQgetCopyData will not block waiting for input; it will return
zero if the COPY is still in progress but no complete row is available. (In this case wait for read-
ready and then call PQconsumeInput before calling PQgetCopyData again.) When async is
false (zero), PQgetCopyData will block until data is available or the operation completes.

After PQgetCopyData returns -1, call PQgetResult to obtain the final result status of the COPY
command. One can wait for this result to be available in the usual way. Then return to normal
operation.

### 30.8.3. Obsolete Functions for COPY

These functions represent older methods of handling COPY. Although they still work, they are depre-
cated due to poor error handling, inconvenient methods of detecting end-of-data, and lack of support
for binary or nonblocking transfers.
Chapter 30. libpq - C Library

PQgetline

Reads a newline-terminated line of characters (transmitted by the server) into a buffer string of size length.

```c
int PQgetline(PGconn *conn,
               char *buffer,
               int length);
```

This function copies up to length-1 characters into the buffer and converts the terminating newline into a zero byte. PQgetline returns EOF at the end of input, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read.

Note that the application must check to see if a new line consists of the two characters \., which indicates that the server has finished sending the results of the COPY command. If the application might receive lines that are more than length-1 characters long, care is needed to be sure it recognizes the \. line correctly (and does not, for example, mistake the end of a long data line for a terminator line).

PQgetlineAsync

Reads a row of COPY data (transmitted by the server) into a buffer without blocking.

```c
int PQgetlineAsync(PGconn *conn,
                    char *buffer,
                    int bufsize);
```

This function is similar to PQgetline, but it can be used by applications that must read COPY data asynchronously, that is, without blocking. Having issued the COPY command and gotten a PGRES_COPY_OUT response, the application should call PQconsumeInput and PQgetlineAsync until the end-of-data signal is detected.

Unlike PQgetline, this function takes responsibility for detecting end-of-data.

On each call, PQgetlineAsync will return data if a complete data row is available in libpq’s input buffer. Otherwise, no data is returned until the rest of the row arrives. The function returns -1 if the end-of-copy-data marker has been recognized, or 0 if no data is available, or a positive number giving the number of bytes of data returned. If -1 is returned, the caller must next call PQendcopy, and then return to normal processing.

The data returned will not extend beyond a data-row boundary. If possible a whole row will be returned at one time. But if the buffer offered by the caller is too small to hold a row sent by the server, then a partial data row will be returned. With textual data this can be detected by testing whether the last returned byte is \n or not. (In a binary COPY, actual parsing of the COPY data format will be needed to make the equivalent determination.) The returned string is not null-terminated. (If you want to add a terminating null, be sure to pass a bufsize one smaller than the room actually available.)

PQputline

Sends a null-terminated string to the server. Returns 0 if OK and EOF if unable to send the string.

```c
int PQputline(PGconn *conn,
              const char *string);
```

The COPY data stream sent by a series of calls to PQputline has the same format as that returned by PQgetlineAsync, except that applications are not obliged to send exactly one data row per PQputline call; it is okay to send a partial line or multiple lines per call.
Note: Before PostgreSQL protocol 3.0, it was necessary for the application to explicitly send the two characters \.. as a final line to indicate to the server that it had finished sending `COPY` data. While this still works, it is deprecated and the special meaning of \.. can be expected to be removed in a future release. It is sufficient to call PQendcopy after having sent the actual data.

PQputnbytes
Sends a non-null-terminated string to the server. Returns 0 if OK and EOF if unable to send the string.

```c
int PQputnbytes(PGconn *conn,
    const char *buffer,
    int nbytes);
```

This is exactly like PQputline, except that the data buffer need not be null-terminated since the number of bytes to send is specified directly. Use this procedure when sending binary data.

PQendcopy
Synchronizes with the server.

```c
int PQendcopy(PGconn *conn);
```

This function waits until the server has finished the copying. It should either be issued when the last string has been sent to the server using PQputline or when the last string has been received from the server using PGgetline. It must be issued or the server will get “out of sync” with the client. Upon return from this function, the server is ready to receive the next SQL command. The return value is 0 on successful completion, nonzero otherwise. (Use PQerrorMessage to retrieve details if the return value is nonzero.)

When using PQgetResult, the application should respond to a PGRES_COPY_OUT result by executing PQgetline repeatedly, followed by PQendcopy after the terminator line is seen. It should then return to the PQgetResult loop until PQgetResult returns a null pointer. Similarly a PGRES_COPY_IN result is processed by a series of PQputline calls followed by PQendcopy, then return to the PQgetResult loop. This arrangement will ensure that a COPY command embedded in a series of SQL commands will be executed correctly.

Older applications are likely to submit a COPY via PQexec and assume that the transaction is done after PQendcopy. This will work correctly only if the COPY is the only SQL command in the command string.

### 30.9. Control Functions

These functions control miscellaneous details of libpq’s behavior.

PQclientEncoding
Returns the client encoding.

```c
int PQclientEncoding(const PGconn *conn);
```

Note that it returns the encoding ID, not a symbolic string such as EUC_JP. To convert an encoding ID to an encoding name, you can use:
char *pg_encoding_to_char(int encoding_id);

PQsetClientEncoding

Sets the client encoding.

int PQsetClientEncoding(PGconn *conn, const char *encoding);

cconn is a connection to the server, and encoding is the encoding you want to use. If the function
successfully sets the encoding, it returns 0, otherwise -1. The current encoding for this connection
 can be determined by using PQclientEncoding.

PQsetErrorVerbosity

Determines the verbosity of messages returned by PQerrorMessage and
PQresultErrorMessage.

typedef enum {
  PQERRORS_TERSE,
  PQERRORS_DEFAULT,
  PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);

PQsetErrorVerbosity sets the verbosity mode, returning the connection’s previous setting.
In TERSE mode, returned messages include severity, primary text, and position only; this will
normally fit on a single line. The default mode produces messages that include the above plus
any detail, hint, or context fields (these might span multiple lines). The VERBOSE mode includes
all available fields. Changing the verbosity does not affect the messages available from already-
existing PGresult objects, only subsequently-created ones.

PQtrace

Enables tracing of the client/server communication to a debugging file stream.

void PQtrace(PGconn *conn, FILE *stream);

Note: On Windows, if the libpq library and an application are compiled with different
flags, this function call will crash the application because the internal representation of
the FILE pointers differ. Specifically, multithreaded/single-threaded, release/debug, and
static/dynamic flags should be the same for the library and all applications using that library.

PQuntrace

Disables tracing started by PQtrace.

void PQuntrace(PGconn *conn);

30.10. Miscellaneous Functions

As always, there are some functions that just don’t fit anywhere.
Chapter 30. libpq - C Library

PQencryptPassword

Prepares the encrypted form of a PostgreSQL password.

```c
char * PQencryptPassword(const char *passwd, const char *user);
```

This function is intended to be used by client applications that wish to send commands like
`ALTER USER joe PASSWORD 'pwd'`. It is good practice not to send the original cleartext password
in such a command, because it might be exposed in command logs, activity displays, and
so on. Instead, use this function to convert the password to encrypted form before it is sent. The
arguments are the cleartext password, and the SQL name of the user it is for. The return value is
a string allocated by `malloc`, or `NULL` if out of memory. The caller can assume the string doesn’t
contain any special characters that would require escaping. Use `PQfreemem` to free the result
when done with it.

30.11. Notice Processing

Notice and warning messages generated by the server are not returned by the query execution func-
tions, since they do not imply failure of the query. Instead they are passed to a notice handling func-
tion, and execution continues normally after the handler returns. The default notice handling function
prints the message on `stderr`, but the application can override this behavior by supplying its own
handling function.

For historical reasons, there are two levels of notice handling, called the notice receiver and notice
processor. The default behavior is for the notice receiver to format the notice and pass a string to the
notice processor for printing. However, an application that chooses to provide its own notice receiver
will typically ignore the notice processor layer and just do all the work in the notice receiver.

The function `PQsetNoticeReceiver` sets or examines the current notice receiver for a connection
object. Similarly, `PQsetNoticeProcessor` sets or examines the current notice processor.

```c
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                     PQnoticeReceiver proc,
                     void *arg);

datatype void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

Each of these functions returns the previous notice receiver or processor function pointer, and sets the
new value. If you supply a null function pointer, no action is taken, but the current pointer is returned.

When a notice or warning message is received from the server, or generated internally by libpq, the
notice receiver function is called. It is passed the message in the form of a `PGRES_NONFATAL_ERROR
PGresult`. (This allows the receiver to extract individual fields using `PQresultErrorField`, or the
complete preformatted message using `PQresultErrorMessage`.) The same void pointer passed to
`PQsetNoticeReceiver` is also passed. (This pointer can be used to access application-specific state
if needed.)
The default notice receiver simply extracts the message (using PQresultErrorMessage) and passes it to the notice processor.

The notice processor is responsible for handling a notice or warning message given in text form. It is passed the string text of the message (including a trailing newline), plus a void pointer that is the same one passed to PQsetNoticeProcessor. (This pointer can be used to access application-specific state if needed.)

The default notice processor is simply:

```c
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

Once you have set a notice receiver or processor, you should expect that that function could be called as long as either the PGconn object or PGresult objects made from it exist. At creation of a PGresult, the PGconn’s current notice handling pointers are copied into the PGresult for possible use by functions like PQgetvalue.

### 30.12. Environment Variables

The following environment variables can be used to select default connection parameter values, which will be used by PQconnectdb, PQsetdbLogin and PQsetdb if no value is directly specified by the calling code. These are useful to avoid hard-coding database connection information into simple client applications, for example.

- **PGHOST** sets the database server name. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is then the name of the directory in which the socket file is stored (in a default installation setup this would be /tmp).
- **PGHOSTADDR** specifies the numeric IP address of the database server. This can be set instead of or in addition to PGHOST to avoid DNS lookup overhead. See the documentation of these parameters, under PQconnectdb above, for details on their interaction.
  
  When neither PGHOST nor PGHOSTADDR is set, the default behavior is to connect using a local Unix-domain socket; or on machines without Unix-domain sockets, libpq will attempt to connect to localhost.
- **PGPORT** sets the TCP port number or Unix-domain socket file extension for communicating with the PostgreSQL server.
- **PGDATABASE** sets the PostgreSQL database name.
- **PGUSER** sets the user name used to connect to the database.
- **PGPASSWORD** sets the password used if the server demands password authentication. Use of this environment variable is not recommended for security reasons (some operating systems allow non-root users to see process environment variables via ps); instead consider using the ~/.pgpass file (see Section 30.13).
- **PGPASSFILE** specifies the name of the password file to use for lookups. If not set, it defaults to ~/.pgpass (see Section 30.13).
• **PGSERVICE** sets the service name to be looked up in `pg_service.conf`. This offers a shorthand way of setting all the parameters.

• **PGREALM** sets the Kerberos realm to use with PostgreSQL, if it is different from the local realm. If `PGREALM` is set, libpq applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the server.

• **PGOPTIONS** sets additional run-time options for the PostgreSQL server.

• **PGSSLMODE** determines whether and with what priority an SSL connection will be negotiated with the server. There are four modes: `disable` will attempt only an unencrypted SSL connection; `allow` will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; `prefer` (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; `require` will try only an SSL connection. If PostgreSQL is compiled without SSL support, using option `require` will cause an error, while options `allow` and `prefer` will be accepted but libpq will not in fact attempt an SSL connection.

• **PGREQUIRESSL** sets whether or not the connection must be made over SSL. If set to “1”, libpq will refuse to connect if the server does not accept an SSL connection (equivalent to `sslmode prefer`). This option is deprecated in favor of the `sslmode` setting, and is only available if PostgreSQL is compiled with SSL support.

• **PGSSLKEY** specifies the hardware token that stores the secret key for the client certificate. The value of this variable should consist of a colon-separated engine name (engines are OpenSSL loadable modules) and an engine-specific key identifier. If this is not set, the secret key must be kept in a file.

• **PGKRBSRVNAME** sets the Kerberos service name to use when authenticating with Kerberos 5 or GSSAPI.

• **PGGSSLIB** sets the GSS library to use for GSSAPI authentication.

• **PGCONNECT_TIMEOUT** sets the maximum number of seconds that libpq will wait when attempting to connect to the PostgreSQL server. If unset or set to zero, libpq will wait indefinitely. It is not recommended to set the timeout to less than 2 seconds.

The following environment variables can be used to specify default behavior for each PostgreSQL session. (See also the `ALTER USER` and `ALTER DATABASE` commands for ways to set default behavior on a per-user or per-database basis.)

• **PGDATESTYLE** sets the default style of date/time representation. (Equivalent to `SET datestyle TO ...`)

• **PGTZ** sets the default time zone. (Equivalent to `SET timezone TO ...`)

• **PGCLIENTENCODING** sets the default client character set encoding. (Equivalent to `SET client_encoding TO ...`)

• **PGGEOQ** sets the default mode for the genetic query optimizer. (Equivalent to `SET geqo TO ...`)

Refer to the SQL command `SET` for information on correct values for these environment variables.

The following environment variables determine internal behavior of libpq; they override compiled-in defaults.

• **PGSYSCONFDIR** sets the directory containing the `pg_service.conf` file.

• **PGLOCALEDIR** sets the directory containing the `locale` files for message internationalization.
30.13. The Password File

The file .pgpass in a user’s home directory or the file referenced by PGPASSFILE can contain passwords to be used if the connection requires a password (and no password has been specified otherwise). On Microsoft Windows the file is named %APPDATA%\postgresql\pgpass.conf (where %APPDATA% refers to the Application Data subdirectory in the user’s profile).

This file should contain lines of the following format:

```
hostname:port:database:username:password
```

Each of the first four fields can be a literal value, or *, which matches anything. The password field from the first line that matches the current connection parameters will be used. (Therefore, put more-specific entries first when you are using wildcards.) If an entry needs to contain : or \, escape this character with \\. A host name of localhost matches both TCP (host name localhost) and Unix domain socket (pghost empty or the default socket directory) connections coming from the local machine.

On Unix systems, the permissions on .pgpass must disallow any access to world or group; achieve this by the command chmod 0600 ~/.pgpass. If the permissions are less strict than this, the file will be ignored. On Microsoft Windows, it is assumed that the file is stored in a directory that is secure, so no special permissions check is made.

30.14. The Connection Service File

The connection service file allows libpq connection parameters to be associated with a single service name. That service name can then be specified by a libpq connection, and the associated settings will be used. This allows connection parameters to be modified without requiring a recompile of the libpq application. The service name can also be specified using the PGSERVICE environment variable.

To use this feature, copy share/pg_service.conf.sample to etc/pg_service.conf and edit the file to add service names and parameters. This file can be used for client-only installs too. The file’s location can also be specified by the PGSYSCONFDIR environment variable.

30.15. LDAP Lookup of Connection Parameters

If libpq has been compiled with LDAP support (option --with-ldap for configure) it is possible to retrieve connection options like host or dbname via LDAP from a central server. The advantage is that if the connection parameters for a database change, the connection information doesn’t have to be updated on all client machines.

LDAP connection parameter lookup uses the connection service file pg_service.conf (see Section 30.14). A line in a pg_service.conf stanza that starts with ldap:// will be recognized as an LDAP URL and an LDAP query will be performed. The result must be a list of keyword = value pairs which will be used to set connection options. The URL must conform to RFC 1959 and be of the form

```
ldap://[hostname[:port]]/search_base?attribute?search_scope?filter
```

where hostname defaults to localhost and port defaults to 389.
Processing of `pg_service.conf` is terminated after a successful LDAP lookup, but is continued if the LDAP server cannot be contacted. This is to provide a fallback with further LDAP URL lines that point to different LDAP servers, classical `keyword = value` pairs, or default connection options. If you would rather get an error message in this case, add a syntactically incorrect line after the LDAP URL.

A sample LDAP entry that has been created with the LDIF file

```plaintext
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:groupOfUniqueNames
cn=mydatabase
uniqueMember:host=dbserver.mycompany.com
uniqueMember:port=5439
uniqueMember:dbname=mydb
uniqueMember:user=mydb_user
uniqueMember:sslmode=require
```

might be queried with the following LDAP URL:

```plaintext
ldap://ldap.mycompany.com/dc=mycompany,dc=com?uniqueMember?one?(cn=mydatabase)
```

You can also mix regular service file entries with LDAP lookups. A complete example for a stanza in `pg_service.conf` would be:

```plaintext
# only host and port are stored in LDAP, specify dbname and user explicitly
[customerdb]
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*),
```

**30.16. SSL Support**

PostgreSQL has native support for using SSL connections to encrypt client/server communications for increased security. See Section 17.8 for details about the server-side SSL functionality.

`libpq` reads the system-wide OpenSSL configuration file. By default, this file is named `openssl.cnf` and is located in the directory reported by `openssl version -d`. This default can be overridden by setting environment variable `OPENSSL_CONF` to the name of the desired configuration file.

To verify the server certificate is trustworthy, place certificates of the certificate authorities (CA) you trust in the file `~/.postgresql/root.crt` in the user’s home directory. (On Microsoft Windows the file is named `%APPDATA%\postgresql\root.crt`.) `libpq` will then verify that the server’s certificate is signed by one of the trusted certificate authorities. The SSL connection will fail if the server does not present a trusted certificate. Certificate Revocation List (CRL) entries are also checked if the file `~/.postgresql/root.crl` exists (`%APPDATA%\postgresql\root.crl` on Microsoft Windows).
Chapter 30. libpq - C Library

If the server requests a trusted client certificate, libpq will send the certificate stored in file
~/.postgresql/postgresql.crt in the user’s home directory. The certificate must be signed
by one of the certificate authorities (CA) trusted by the server. A matching private key file
~/.postgresql/postgresql.key must also be present, unless the secret key for the certificate is
stored in a hardware token, as specified by PGSSLKEY. (On Microsoft Windows these files are named
%APPDATA%\postgresql\postgresql.crt and %APPDATA%\postgresql\postgresql.key.)
The private key file must not be world-readable.

If the environment variable PGSSLKEY is set, its value should consist of a colon-separated engine
name and key identifier. In this case, libpq will load the specified engine, i.e. the OpenSSL module
which supports special hardware, and reference the key with the specified identifier. Identifiers are
eengine-specific. Typically, cryptography hardware tokens do not reveal secret keys to the application.
Instead, applications delegate all cryptography operations which require the secret key to the hardware
token.

If you are using SSL inside your application (in addition to inside libpq), you can use
PQinitSSL(int) to tell libpq that the SSL library has already been initialized by your application.

Table 30-1. Libpq/Client SSL File Usage

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>~/.postgresql/postgresql.crt</td>
<td>client certificate</td>
<td>requested by server</td>
</tr>
<tr>
<td>~/.postgresql/postgresql.key</td>
<td>client private key</td>
<td>proves client certificate sent by owner; does not indicate certificate owner is trustworthy</td>
</tr>
<tr>
<td>~/.postgresql/root.crt</td>
<td>trusted certificate authorities</td>
<td>checks server certificate is signed by a trusted certificate authority</td>
</tr>
<tr>
<td>~/.postgresql/root.crl</td>
<td>certificates revoked by</td>
<td>server certificate must not be on certificate authorities</td>
</tr>
</tbody>
</table>

30.17. Behavior in Threaded Programs

libpq is reentrant and thread-safe if the configure command-line option
--enable-thread-safety was used when the PostgreSQL distribution was built. In addition, you
might need to use additional compiler command-line options when you compile your application
code. Refer to your system’s documentation for information about how to build thread-enabled
applications, or look in src/Makefile.global for PTHREAD_CFLAGS and PTHREAD_LIBS. This
function allows the querying of libpq’s thread-safe status:

PQisthreadsafe

Returns the thread safety status of the libpq library.

int PQisthreadsafe();

Returns 1 if the libpq is thread-safe and 0 if it is not.

One thread restriction is that no two threads attempt to manipulate the same PGconn object at the
same time. In particular, you cannot issue concurrent commands from different threads through the
same connection object. (If you need to run concurrent commands, use multiple connections.)

PGresult objects are read-only after creation, and so can be passed around freely between threads.

The deprecated functions PQrequestCancel and PQoidStatus are not thread-safe and should not be used in multithread programs. PQrequestCancel can be replaced by PQcancel. PQoidStatus can be replaced by PQoidValue.

If you are using Kerberos inside your application (in addition to inside libpq), you will need to do locking around Kerberos calls because Kerberos functions are not thread-safe. See function PQregisterThreadLock in the libpq source code for a way to do cooperative locking between libpq and your application.

If you experience problems with threaded applications, run the program in src/tools/thread to see if your platform has thread-unsafe functions. This program is run by configure, but for binary distributions your library might not match the library used to build the binaries.

### 30.18. Building libpq Programs

To build (i.e., compile and link) a program using libpq you need to do all of the following things:

- Include the libpq-fe.h header file:

  ```
  #include <libpq-fe.h>
  ```

  If you failed to do that then you will normally get error messages from your compiler similar to

  ```
  foo.c:34: 'PGconn' undeclared (first use in this function)
  foo.c:35: 'PGresult' undeclared (first use in this function)
  foo.c:54: 'CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: 'PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: 'PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- Point your compiler to the directory where the PostgreSQL header files were installed, by supplying the `-I` directory option to your compiler. (In some cases the compiler will look into the directory in question by default, so you can omit this option.) For instance, your compile command line could look like:

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  If you are using makefiles then add the option to the `CPPFLAGS` variable:

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  If there is any chance that your program might be compiled by other users then you should not hardcode the directory location like that. Instead, you can run the utility `pg_config` to find out where the header files are on the local system:

  ```
  $ pg_config --includedir /usr/local/include
  ```

  Failure to specify the correct option to the compiler will result in an error message such as

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- When linking the final program, specify the option `-lpq` so that the libpq library gets pulled in, as well as the option `-L` directory to point the compiler to the directory where the libpq library resides. (Again, the compiler will search some directories by default.) For maximum portability, put the `-L` option before the `-lpq` option. For example:
Chapter 30. libpq - C Library

cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq

You can find out the library directory using `pg_config` as well:

$ pg_config --libdir
/usr/local/pgsql/lib

Error messages that point to problems in this area could look like the following.

testlibpq.o: In function ‘main’:
testlibpq.o(.text+0x60): undefined reference to ‘PQsetdbLogin’
testlibpq.o(.text+0x71): undefined reference to ‘PQstatus’
testlibpq.o(.text+0xa4): undefined reference to ‘PQerrorMessage’
This means you forgot `-lpq`.
/usr/bin/ld: cannot find `-lpq`
This means you forgot the `-L` option or did not specify the right directory.

30.19. Example Programs

These examples and others can be found in the directory `src/test/examples` in the source code distribution.

Example 30-1. libpq Example Program 1

```c
/*
 * testlibpq.c
 *
 * Test the C version of libpq, the PostgreSQL frontend library.
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn *conn;
    PGresult *res;
    int nFields;
    int i,
     j;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
```
if (argc > 1)
    conninfo = argv[1];
else
    conninfo = "dbname = postgres";

/* Make a connection to the database */
conn = PQconnectdb(conninfo);

/* Check to see that the backend connection was successfully made */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}

/* Our test case here involves using a cursor, for which we must be inside
 * a transaction block. We could do the whole thing with a single
 * PQexec() of "select * from pg_database", but that's too trivial to make
 * a good example.
 */

/* Start a transaction block */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* Should PQclear PGresult whenever it is no longer needed to avoid memory
 * leaks
 */
PQclear(res);

/* Fetch rows from pg_database, the system catalog of databases */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
Chapter 30. libpq - C Library

Example 30-2. libpq Example Program 2

```c
/*
 * testlibpq2.c
 * Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 * NOTIFY TBL2;
 * Repeat four times to get this program to exit.
 *
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
 * (provided in src/test/examples/testlibpq2.sql):
 *  * CREATE TABLE TBL1 (i int4);
 *  * CREATE TABLE TBL2 (i int4);
 *  * CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *  * (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *  *
 *  * and do this four times:
 *  *
 *  * INSERT INTO TBL1 VALUES (10);
 *  */

#include <stdio.h>
```
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
  PQfinish(conn);
  exit(1);
}

int
main(int argc, char **argv)
{
  const char *conninfo;
  PGconn  *conn;
  PGresult *res;
  PGnotify *notify;
  int nnotifies;

  /*
   * If the user supplies a parameter on the command line, use it as the
   * conninfo string; otherwise default to setting dbname=postgres and using
   * environment variables or defaults for all other connection parameters.
   */
  if (argc > 1)
    conninfo = argv[1];
  else
    conninfo = "dbname = postgres";

  /* Make a connection to the database */
  conn = PQconnectdb(conninfo);

  /* Check to see that the backend connection was successfully made */
  if (PQstatus(conn) != CONNECTION_OK)
    {
      fprintf(stderr, "Connection to database failed: %s", 
PQerrorMessage(conn));
      exit_nicely(conn);
    }

  /*
   * Issue LISTEN command to enable notifications from the rule’s NOTIFY.
   */
  res = PQexec(conn, "LISTEN TBL2");
  if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
      fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
      PQclear(res);
      exit_nicely(conn);
    }

  /*
   * should PQclear PGresult whenever it is no longer needed to avoid memory
   * leaks
   */
PQclear(res);

/* Quit after four notifies are received. */
nnotifies = 0;
while (nnotifies < 4)
{
    /*
     * Sleep until something happens on the connection. We use select(2)
     * to wait for input, but you could also use poll() or similar
     * facilities.
     */
    int sock;
    FD_ZERO(&input_mask);
    sock = PQsocket(conn);
    if (sock < 0)
        /* shouldn’t happen */
        break;
    FD_SET(sock, &input_mask);
    if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
    {
        fprintf(stderr, "select() failed: %s\n", strerror(errno));
        exit_nicely(conn);
    }
    /* Now check for input */
    PQconsumeInput(conn);
    while ((notify = PQnotifies(conn)) != NULL)
    {
        fprintf(stderr,
                "ASYNC NOTIFY of ‘%s’ received from backend pid %d\n",
                notify->relname, notify->be_pid);
        PQfreemem(notify);
        nnotifies++;
    }
    fprintf(stderr, "Done.\n");
}

Example 30-3. libpq Example Program 3

/*
 * testlibpq3.c
 * Test out-of-line parameters and binary I/O.
 * Before running this, populate a database with the following commands
 * (provided in src/test/examples/testlibpq3.sql):
 */
CREATE TABLE test1 (i int4, t text, b bytea);

INSERT INTO test1 values (1, 'joe''s place', '\000\001\002\003\004');
INSERT INTO test1 values (2, 'ho there', '\004\003\002\001\000');

The expected output is:

tuple 0: got
  i = (4 bytes) 1
  t = (11 bytes) 'joe’s place'
  b = (5 bytes) \000\001\002\003\004

tuple 0: got
  i = (4 bytes) 2
  t = (8 bytes) 'ho there'
  b = (5 bytes) \004\003\002\001\000

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

#include <netinet/in.h>
#include <arpa/inet.h>

static void
exit_nicely(PGconn *conn)
{
  PQfinish(conn);
  exit(1);
}

/* This function prints a query result that is a binary-format fetch from */
/* a table defined as in the comment above. We split it out because the */
/* main() function uses it twice. */
static void
show_binary_results(PGresult *res)
{
  int i,
      j;
  int i_fnum,
      t_fnum,
      b_fnum;

  /* Use PQfnumber to avoid assumptions about field order in result */
  i_fnum = PQfnumber(res, "i");
  t_fnum = PQfnumber(res, "t");
  b_fnum = PQfnumber(res, "b");

  for (i = 0; i < PQntuples(res); i++)
  {
Chapter 30. libpq - C Library

char *iptr;
char *tptr;
char *bptr;
int blen;
int ival;

/* Get the field values (we ignore possibility they are null!) */
iptr = PQgetvalue(res, i, i_fnum);
tptr = PQgetvalue(res, i, t_fnum);
bptr = PQgetvalue(res, i, b_fnum);

/*
 * The binary representation of INT4 is in network byte order, which
 * we'd better coerce to the local byte order.
 */
ival = ntohl(*((uint32_t *) iptr));

/*
 * The binary representation of TEXT is, well, text, and since libpq
 * was nice enough to append a zero byte to it, it'll work just fine
 * as a C string.
 *
 * The binary representation of BYTEA is a bunch of bytes, which could
 * include embedded nulls so we have to pay attention to field length.
 */
blen = PQgetlength(res, i, b_fnum);

printf("tuple \$d: got\n", i);
printf(" i = (%d bytes) %d\n",
       PQgetlength(res, i, i_fnum), ival);
printf(" t = (%d bytes) \%s\n",
       PQgetlength(res, i, t_fnum), tptr);
printf(" b = (%d bytes) ", blen);
for (j = 0; j < blen; j++)
    printf("\%03o", bptr[j]);
printf("\n\n");
}

int main(int argc, char **argv)
{
    const char *conninfo;
    PGconn *conn;
    PGresult *res;
    const char *paramValues[1];
    int paramLengths[1];
    int paramFormats[1];
    uint32_t binaryIntVal;

    /*
    * If the user supplies a parameter on the command line, use it as the
    * conninfo string; otherwise default to setting dbname=postgres and using
    * environment variables or defaults for all other connection parameters.
    */
    if (argc > 1)
        conninfo = argv[1];
else
    conninfo = "dbname = postgres";

/* Make a connection to the database */
conn = PQconnectdb(conninfo);

/* Check to see that the backend connection was successfully made */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
        PQerrorMessage(conn));
    exit_nicely(conn);
}

/* The point of this program is to illustrate use of PQexecParams() with
 * out-of-line parameters, as well as binary transmission of data.
 * This first example transmits the parameters as text, but receives the
 * results in binary format. By using out-of-line parameters we can
 * avoid a lot of tedious mucking about with quoting and escaping, even
 * though the data is text. Notice how we don’t have to do anything
 * special with the quote mark in the parameter value.
 */

/* Here is our out-of-line parameter value */
paramValues[0] = "joe’s place";

res = PQexecParams(conn,
    "SELECT * FROM test1 WHERE t = $1",
    1,    /* one param */
    NULL,  /* let the backend deduce param type */
    paramValues,
    NULL,  /* don’t need param lengths since text */
    NULL,  /* default to all text params */
    1);  /* ask for binary results */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/*
 * In this second example we transmit an integer parameter in binary
 * form, and again retrieve the results in binary form.
 * Although we tell PQexecParams we are letting the backend deduce
 * parameter type, we really force the decision by casting the parameter
 * symbol in the query text. This is a good safety measure when sending
 * binary parameters.
 */
/* Convert integer value "2" to network byte order */
binaryIntVal = htonl((uint32_t) 2);

/* Set up parameter arrays for PQexecParams */
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);
paramFormats[0] = 1; /* binary */

res = PQexecParams(conn,
"SELECT * FROM test1 WHERE i = $1::int4",
1, /* one param */
NULL, /* let the backend deduce param type */
paramValues,
paramLengths,
paramFormats,
1); /* ask for binary results */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);
PQclear(res);

/* close the connection to the database and cleanup */
PQfinish(conn);

return 0;

Chapter 31. Large Objects

PostgreSQL has a *large object* facility, which provides stream-style access to user data that is stored in a special large-object structure. Streaming access is useful when working with data values that are too large to manipulate conveniently as a whole.

This chapter describes the implementation and the programming and query language interfaces to PostgreSQL large object data. We use the libpq C library for the examples in this chapter, but most programming interfaces native to PostgreSQL support equivalent functionality. Other interfaces might use the large object interface internally to provide generic support for large values. This is not described here.

31.1. Introduction

All large objects are placed in a single system table called `pg_largeobject`. PostgreSQL also supports a storage system called “TOAST” that automatically stores values larger than a single database page into a secondary storage area per table. This makes the large object facility partially obsolete. One remaining advantage of the large object facility is that it allows values up to 2 GB in size, whereas TOASTed fields can be at most 1 GB. Also, large objects can be randomly modified using a read/write API that is more efficient than performing such operations using TOAST.

31.2. Implementation Features

The large object implementation breaks large objects up into “chunks” and stores the chunks in rows in the database. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes.

31.3. Client Interfaces

This section describes the facilities that PostgreSQL client interface libraries provide for accessing large objects. All large object manipulation using these functions must take place within an SQL transaction block. The PostgreSQL large object interface is modeled after the Unix file-system interface, with analogues of `open`, `read`, `write`, `lseek`, etc.

Client applications which use the large object interface in libpq should include the header file `libpq/libpq-fs.h` and link with the libpq library.

31.3.1. Creating a Large Object

The function

```c
Oid lo_creat(PGconn *conn, int mode);
```

creates a new large object. The return value is the OID that was assigned to the new large object, or `InvalidOid` (zero) on failure. `mode` is unused and ignored as of PostgreSQL 8.1; however, for backwards compatibility with earlier releases it is best to set it to `INV_READ`, `INV_WRITE`, or `INV_READ | INV_WRITE`. (These symbolic constants are defined in the header file `libpq/libpq-fs.h`.)
Chapter 31. Large Objects

31.3.2. Importing a Large Object

To import an operating system file as a large object, call

```c
Oid lo_import(PGconn *conn, const char *filename);
```

*filename* specifies the operating system name of the file to be imported as a large object. The return value is the OID that was assigned to the new large object, or InvalidOid (zero) on failure. Note that the file is read by the client interface library, not by the server; so it must exist in the client file system and be readable by the client application.

31.3.3. Exporting a Large Object

To export a large object into an operating system file, call

```c
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

The *lobjId* argument specifies the OID of the large object to export and the *filename* argument specifies the operating system name of the file. Note that the file is written by the client interface library, not by the server. Returns 1 on success, -1 on failure.

31.3.4. Opening an Existing Large Object

To open an existing large object for reading or writing, call

```c
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

The *lobjId* argument specifies the OID of the large object to open. The *mode* bits control whether the object is opened for reading (INV_READ), writing (INV_WRITE), or both. (These symbolic constants
A large object cannot be opened before it is created. `lo_open` returns a (non-negative) large object descriptor for later use in `lo_read`, `lo_write`, `lo_lseek`, `lo_tell`, and `lo_close`. The descriptor is only valid for the duration of the current transaction. On failure, -1 is returned.

The server currently does not distinguish between modes `INV_WRITE` and `INV_READ | INV_WRITE`: you are allowed to read from the descriptor in either case. However there is a significant difference between these modes and `INV_READ` alone: with `INV_READ` you cannot write on the descriptor, and the data read from it will reflect the contents of the large object at the time of the transaction snapshot that was active when `lo_open` was executed, regardless of later writes by this or other transactions. Reading from a descriptor opened with `INV_WRITE` returns data that reflects all writes of other committed transactions as well as writes of the current transaction. This is similar to the behavior of `SERIALIZABLE` versus `READ_COMMITTED` transaction modes for ordinary SQL `SELECT` commands.

An example:

```c
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

### 31.3.5. Writing Data to a Large Object

The function

```c
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

writes `len` bytes from `buf` to large object descriptor `fd`. The `fd` argument must have been returned by a previous `lo_open`. The number of bytes actually written is returned. In the event of an error, the return value is negative.

### 31.3.6. Reading Data from a Large Object

The function

```c
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

reads `len` bytes from large object descriptor `fd` into `buf`. The `fd` argument must have been returned by a previous `lo_open`. The number of bytes actually read is returned. In the event of an error, the return value is negative.

### 31.3.7. Seeking in a Large Object

To change the current read or write location associated with a large object descriptor, call

```c
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

This function moves the current location pointer for the large object descriptor identified by `fd` to the new location specified by `offset`. The valid values for `whence` are `SEEK_SET` (seek from object start), `SEEK_CUR` (seek from current position), and `SEEK_END` (seek from object end). The return value is the new location pointer, or -1 on error.
31.3.8. Obtaining the Seek Position of a Large Object

To obtain the current read or write location of a large object descriptor, call

```c
int lo_tell(PGconn *conn, int fd);
```

If there is an error, the return value is negative.

31.3.9. Truncating a Large Object

To truncate a large object to a given length, call

```c
int lo_truncate(PGconn *conn, int fd, size_t len);
```

truncates the large object descriptor `fd` to length `len`. The `fd` argument must have been returned by a previous `lo_open`. If `len` is greater than the current large object length, the large object is extended with null bytes (`\0`).

The file offset is not changed.

On success `lo_truncate` returns zero. On error, the return value is negative.

`lo_truncate` is new as of PostgreSQL 8.3; if this function is run against an older server version, it will fail and return a negative value.

31.3.10. Closing a Large Object Descriptor

A large object descriptor can be closed by calling

```c
int lo_close(PGconn *conn, int fd);
```

where `fd` is a large object descriptor returned by `lo_open`. On success, `lo_close` returns zero. On error, the return value is negative.

Any large object descriptors that remain open at the end of a transaction will be closed automatically.

31.3.11. Removing a Large Object

To remove a large object from the database, call

```c
int lo_unlink(PGconn *conn, Oid lobjId);
```

The `lobjId` argument specifies the OID of the large object to remove. Returns 1 if successful, -1 on failure.

31.4. Server-Side Functions

There are server-side functions callable from SQL that correspond to each of the client-side functions described above; indeed, for the most part the client-side functions are simply interfaces to the equiva-
lent server-side functions. The ones that are actually useful to call via SQL commands are `lo_creat`, `lo_create`, `lo_unlink`, `lo_import`, and `lo_export`. Here are examples of their use:

```sql
CREATE TABLE image (  
    name text,  
    raster oid  
);  
SELECT lo_creat(-1);  -- returns OID of new, empty large object
SELECT lo_create(43213);  -- attempts to create large object with OID 43213
SELECT lo_unlink(173454);  -- deletes large object with OID 173454
INSERT INTO image (name, raster)  
    VALUES ('beautiful image', lo_import('/etc/motd'));
SELECT lo_export(image.raster, '/tmp/motd') FROM image  
    WHERE name = 'beautiful image';
```

The server-side `lo_import` and `lo_export` functions behave considerably differently from their client-side analogs. These two functions read and write files in the server’s file system, using the permissions of the database’s owning user. Therefore, their use is restricted to superusers. In contrast, the client-side import and export functions read and write files in the client’s file system, using the permissions of the client program. The client-side functions can be used by any PostgreSQL user.

### 31.5. Example Program

Example 31-1 is a sample program which shows how the large object interface in libpq can be used. Parts of the program are commented out but are left in the source for the reader’s benefit. This program can also be found in `src/test/examples/testlo.c` in the source distribution.

**Example 31-1. Large Objects with libpq Example Program**

```c
/*--------------------------------------------------------------
*    testlo.c--
*    test using large objects with libpq
*    Copyright (c) 1994, Regents of the University of California
*    --------------------------------------------------------------
*/
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"
#define BUFSIZE 1024

/* importFile
* import file "in_filename" into database as large object "lobj0id"
* */
```
Chapter 31. Large Objects

```c
Oid
importFile(PGconn *conn, char *filename)
{
    Oid lobjId;
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes,
        tmp;
    int fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {
        /* error */
        fprintf(stderr, "cannot open unix file \%s\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int lobj_fd;
    char *buf;
    int nbytes;
    int nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "cannot open large object \%d\n",

lobjId);
}
lo_lseek(conn, lobj_fd, start, SEEK_SET);
buf = malloc(len + 1);

nread = 0;
while (len - nread > 0)
{
    nbytes = lo_read(conn, lobj_fd, buf, len - nread);
    buf[nbytes] = ' ';  
    fprintf(stderr, ">>>")
    fprintf(stderr, "%s", buf);
    nread += nbytes;
}
free(buf);
fprintf(stderr, "\n");
lo_close(conn, lobj_fd);
}

void overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int lobj_fd;
    char *buf;
    int nbytes;
    int nwritten;
    int i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "cannot open large object %d\n", lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';
    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/ * exportFile
 * export large object "lobjOid" to file "out_filename"
 *
Chapter 31. Large Objects

```c
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes,
        tmp;
    int fd;

    /*
    * open the large object
    */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "cannot open large object %d\n",
                lobjId);
    }

    /*
    * open the file to be written to
    */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {
        /* error */
        fprintf(stderr, "cannot open unix file %s\n",
                filename);
    }

    /*
    * read in from the inversion file and write to the Unix file
    */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                    filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
```
main(int argc, char **argv) {
    char *in_filename,
        *out_filename;
    char *database;
    Oid  lobjOid;
    PGconn *conn;
    PGresult *res;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n", argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /* set up the connection */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) == CONNECTION_BAD) {
        fprintf(stderr, "Connection to database '%s' failed.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file %s\n", in_filename);
    lobjOid = lo_import(conn, in_filename);
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's\n");
    overwrite(conn, lobjOid, 1000, 1000);

    printf("exporting large object to file %s\n", out_filename);
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
Chapter 32. ECPG - Embedded SQL in C

This chapter describes the embedded SQL package for PostgreSQL. It was written by Linus Tolke (<linus@epact.se>) and Michael Meskes (<meskes@postgresql.org>). Originally it was written to work with C. It also works with C++, but it does not recognize all C++ constructs yet.

This documentation is quite incomplete. But since this interface is standardized, additional information can be found in many resources about SQL.

32.1. The Concept

An embedded SQL program consists of code written in an ordinary programming language, in this case C, mixed with SQL commands in specially marked sections. To build the program, the source code is first passed through the embedded SQL preprocessor, which converts it to an ordinary C program, and afterwards it can be processed by a C compiler.

Embedded SQL has advantages over other methods for handling SQL commands from C code. First, it takes care of the tedious passing of information to and from variables in your C program. Second, the SQL code in the program is checked at build time for syntactical correctness. Third, embedded SQL in C is specified in the SQL standard and supported by many other SQL database systems. The PostgreSQL implementation is designed to match this standard as much as possible, and it is usually possible to port embedded SQL programs written for other SQL databases to PostgreSQL with relative ease.

As already stated, programs written for the embedded SQL interface are normal C programs with special code inserted to perform database-related actions. This special code always has the form:

EXEC SQL ...;

These statements syntactically take the place of a C statement. Depending on the particular statement, they can appear at the global level or within a function. Embedded SQL statements follow the case-sensitivity rules of normal SQL code, and not those of C.

The following sections explain all the embedded SQL statements.

32.2. Connecting to the Database Server

One connects to a database using the following statement:

EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];

The target can be specified in the following ways:

- dbname[@hostname][:port]
- tcp:postgresql://hostname[:port]/dbname[?options]
- unix:postgresql://hostname[:port]/dbname[?options]
- an SQL string literal containing one of the above forms
- a reference to a character variable containing one of the above forms (see examples)
- DEFAULT
If you specify the connection target literally (that is, not through a variable reference) and you don’t quote the value, then the case-insensitivity rules of normal SQL are applied. In that case you can also double-quote the individual parameters separately as needed. In practice, it is probably less error-prone to use a (single-quoted) string literal or a variable reference. The connection target DEFAULT initiates a connection to the default database under the default user name. No separate user name or connection name can be specified in that case.

There are also different ways to specify the user name:

- `username`
- `username/password`
- `username IDENTIFIED BY password`
- `username USING password`

As above, the parameters `username` and `password` can be an SQL identifier, an SQL string literal, or a reference to a character variable.

The connection-name is used to handle multiple connections in one program. It can be omitted if a program uses only one connection. The most recently opened connection becomes the current connection, which is used by default when an SQL statement is to be executed (see later in this chapter).

Here are some examples of `CONNECT` statements:

```sql
EXEC SQL CONNECT TO mydb@sql.mydomain.com;
EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;
EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
EXEC SQL END DECLARE SECTION;
EXEC SQL CONNECT TO :target USER :user;
```

The last form makes use of the variant referred to above as character variable reference. You will see in later sections how C variables can be used in SQL statements when you prefix them with a colon. Be advised that the format of the connection target is not specified in the SQL standard. So if you want to develop portable applications, you might want to use something based on the last example above to encapsulate the connection target string somewhere.

### 32.3. Closing a Connection

To close a connection, use the following statement:

```sql
EXEC SQL DISCONNECT [connection];
```

The `connection` can be specified in the following ways:

- `connection-name`
- `DEFAULT`
If no connection name is specified, the current connection is closed.
It is good style that an application always explicitly disconnect from every connection it opened.

32.4. Running SQL Commands

Any SQL command can be run from within an embedded SQL application. Below are some examples of how to do that.

Creating a table:

EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;

Inserting rows:

EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;

Deleting rows:

EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;

Single-row select:

EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';

Select using cursors:

EXEC SQL DECLARE foo_bar CURSOR FOR
SELECT number, ascii FROM foo
ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
... EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;

Updates:

EXEC SQL UPDATE foo
SET ascii = 'foobar'
WHERE number = 9999;
EXEC SQL COMMIT;
Chapter 32. ECPG - Embedded SQL in C

The tokens of the form :something are host variables, that is, they refer to variables in the C program. They are explained in Section 32.6.

In the default mode, statements are committed only when EXEC SQL COMMIT is issued. The embedded SQL interface also supports autocommit of transactions (similar to libpq behavior) via the -t command-line option to ecpg (see below) or via the EXEC SQL SET AUTOCOMMIT TO ON statement. In autocommit mode, each command is automatically committed unless it is inside an explicit transaction block. This mode can be explicitly turned off using EXEC SQL SET AUTOCOMMIT TO OFF.

32.5. Choosing a Connection

The SQL statements shown in the previous section are executed on the current connection, that is, the most recently opened one. If an application needs to manage multiple connections, then there are two ways to handle this.

The first option is to explicitly choose a connection for each SQL statement, for example:

EXEC SQL AT connection-name SELECT ...;

This option is particularly suitable if the application needs to use several connections in mixed order. If your application uses multiple threads of execution, they cannot share a connection concurrently. You must either explicitly control access to the connection (using mutexes) or use a connection for each thread. If each thread uses its own connection, you will need to use the AT clause to specify which connection the thread will use.

The second option is to execute a statement to switch the current connection. That statement is:

EXEC SQL SET CONNECTION connection-name;

This option is particularly convenient if many statements are to be executed on the same connection. It is not thread-aware.

32.6. Using Host Variables

In Section 32.4 you saw how you can execute SQL statements from an embedded SQL program. Some of those statements only used fixed values and did not provide a way to insert user-supplied values into statements or have the program process the values returned by the query. Those kinds of statements are not really useful in real applications. This section explains in detail how you can pass data between your C program and the embedded SQL statements using a simple mechanism called host variables. In an embedded SQL program we consider the SQL statements to be guests in the C program code which is the host language. Therefore the variables of the C program are called host variables.

32.6.1. Overview

Passing data between the C program and the SQL statements is particularly simple in embedded SQL. Instead of having the program paste the data into the statement, which entails various complications,
such as properly quoting the value, you can simply write the name of a C variable into the SQL statement, prefixed by a colon. For example:

```sql
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

This statements refers to two C variables named `v1` and `v2` and also uses a regular SQL string literal, to illustrate that you are not restricted to use one kind of data or the other.

This style of inserting C variables in SQL statements works anywhere a value expression is expected in an SQL statement.

### 32.6.2. Declare Sections

To pass data from the program to the database, for example as parameters in a query, or to pass data from the database back to the program, the C variables that are intended to contain this data need to be declared in specially marked sections, so the embedded SQL preprocessor is made aware of them.

This section starts with:

```sql
EXEC SQL BEGIN DECLARE SECTION;
```

and ends with:

```sql
EXEC SQL END DECLARE SECTION;
```

Between those lines, there must be normal C variable declarations, such as:

```c
int x = 4;
char foo[16], bar[16];
```

As you can see, you can optionally assign an initial value to the variable. The variable’s scope is determined by the location of its declaring section within the program. You can also declare variables with the following syntax which implicitly creates a declare section:

```sql
EXEC SQL int i = 4;
```

You can have as many declare sections in a program as you like.

The declarations are also echoed to the output file as normal C variables, so there’s no need to declare them again. Variables that are not intended to be used in SQL commands can be declared normally outside these special sections.

The definition of a structure or union also must be listed inside a DECLARE section. Otherwise the preprocessor cannot handle these types since it does not know the definition.

### 32.6.3. Different types of host variables

As a host variable you can also use arrays, typedefs, structs and pointers. Moreover there are special types of host variables that exist only in ECPG.

A few examples on host variables:
Arrays

One of the most common uses of an array declaration is probably the allocation of a char array as in:

EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;

Note that you have to take care of the length for yourself. If you use this host variable as the target variable of a query which returns a string with more than 49 characters, a buffer overflow occurs.

Typedefs

Use the typedef keyword to map new types to already existing types.

EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;

Note that you could also use:

EXEC SQL TYPE serial_t IS long;

This declaration does not need to be part of a declare section.

Pointers

You can declare pointers to the most common types. Note however that you cannot use pointers as target variables of queries without auto-allocation. See Section 32.10 for more information on auto-allocation.

EXEC SQL BEGIN DECLARE SECTION;
    int  *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;

Special types of variables

ECPG contains some special types that help you to interact easily with data from the SQL server. For example it has implemented support for the varchar, numeric, date, timestamp, and interval types. Section 32.8 contains basic functions to deal with those types, such that you do not need to send a query to the SQL server just for adding an interval to a timestamp for example.

The special type VARCHAR is converted into a named struct for every variable. A declaration like:

VARCHAR var[180];

is converted into:

struct varchar_var { int len; char arr[180]; } var;

This structure is suitable for interfacing with SQL datums of type varchar.

32.6.4. SELECT INTO and FETCH INTO

Now you should be able to pass data generated by your program into an SQL command. But how do you retrieve the results of a query? For that purpose, embedded SQL provides special variants of the usual commands SELECT and FETCH. These commands have a special INTO clause that specifies which host variables the retrieved values are to be stored in.
Chapter 32. ECPG - Embedded SQL in C

Here is an example:

```
/*
 * assume this table:
 * CREATE TABLE test1 (a int, b varchar(50));
 */
EXEC SQL BEGIN DECLARE SECTION;
    int v1;
    VARCHAR v2;
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

So the INTO clause appears between the select list and the FROM clause. The number of elements in the select list and the list after INTO (also called the target list) must be equal.

Here is an example using the command FETCH:

```
EXEC SQL BEGIN DECLARE SECTION;
    int v1;
    VARCHAR v2;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

... do {
    ... 
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

Here the INTO clause appears after all the normal clauses.

Both of these methods only allow retrieving one row at a time. If you need to process result sets that potentially contain more than one row, you need to use a cursor, as shown in the second example.

### 32.6.5. Indicators

The examples above do not handle null values. In fact, the retrieval examples will raise an error if they fetch a null value from the database. To be able to pass null values to the database or retrieve null values from the database, you need to append a second host variable specification to each host variable that contains data. This second host variable is called the indicator and contains a flag that tells whether the datum is null, in which case the value of the real host variable is ignored. Here is an example that handles the retrieval of null values correctly:

```
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR val;
    int val_ind;
EXEC SQL END DECLARE SECTION;
```
EXEC SQL SELECT b INTO :val :val_ind FROM test1;

The indicator variable \( \text{val\_ind} \) will be zero if the value was not null, and it will be negative if the value was null.

The indicator has another function: if the indicator value is positive, it means that the value is not null, but it was truncated when it was stored in the host variable.

### 32.7. Dynamic SQL

In many cases, the particular SQL statements that an application has to execute are known at the time the application is written. In some cases, however, the SQL statements are composed at run time or provided by an external source. In these cases you cannot embed the SQL statements directly into the C source code, but there is a facility that allows you to call arbitrary SQL statements that you provide in a string variable.

The simplest way to execute an arbitrary SQL statement is to use the command `EXECUTE IMMEDIATE`. For example:

```c
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE IMMEDIATE :stmt;
```

You cannot execute statements that retrieve data (e.g., `SELECT`) this way.

A more powerful way to execute arbitrary SQL statements is to prepare them once and execute the prepared statement as often as you like. It is also possible to prepare a generalized version of a statement and then execute specific versions of it by substituting parameters. When preparing the statement, write question marks where you want to substitute parameters later. For example:

```c
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE mystmt FROM :stmt;
...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

If the statement you are executing returns values, then add an `INTO` clause:

```c
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3;
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE mystmt FROM :stmt;
...
EXEC SQL EXECUTE mystmt INTO v1, v2, v3 USING 37;
```
Chapter 32. ECPG - Embedded SQL in C

An **EXECUTE** command can have an **INTO** clause, a **USING** clause, both, or neither.

When you don’t need the prepared statement anymore, you should deallocate it:

EXEC SQL DEALLOCATE PREPARE name;

### 32.8. pgtypes library

The pgtypes library maps PostgreSQL database types to C equivalents that can be used in C programs. It also offers functions to do basic calculations with those types within C, i.e. without the help of the PostgreSQL server. See the following example:

```sql
EXEC SQL BEGIN DECLARE SECTION;
    date date1;
    timestamp ts1, tsout;
    interval iv1;
    char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
free(out);
```

### 32.8.1. The numeric type

The numeric type offers to do calculations with arbitrary precision. See Section 8.1 for the equivalent type in the PostgreSQL server. Because of the arbitrary precision this variable needs to be able to expand and shrink dynamically. That’s why you can only create variables on the heap by means of the **PGTYPESnumeric_new** and **PGTYPESnumeric_free** functions. The decimal type, which is similar but limited in the precision, can be created on the stack as well as on the heap.

The following functions can be used to work with the numeric type:

**PGTYPESnumeric_new**

Request a pointer to a newly allocated numeric variable.

```c
numeric *PGTYPESnumeric_new(void);
```

**PGTYPESnumeric_free**

Free a numeric type, release all of its memory.

```c
void PGTYPESnumeric_free(numeric *var);
```

**PGTYPESnumeric_from_asc**

Parse a numeric type from its string notation.

```c
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```
Valid formats are for example: \(-2.794, +3.44, 592.49E07\) or \(-32.84e-4\). If the value could be parsed successfully, a valid pointer is returned, else the NULL pointer. At the moment ecpg always parses the complete string and so it currently does not support to store the address of the first invalid character in \(*\text{endptr}\). You can safely set \text{endptr}\ to NULL.

\textbf{PGTYPESnumeric_to_asc}

Returns a pointer to a string allocated by \text{malloc} that contains the string representation of the numeric type \text{num}.

\begin{verbatim}
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
The numeric value will be printed with \text{dscale} decimal digits, with rounding applied if necessary.
\end{verbatim}

\textbf{PGTYPESnumeric_add}

Add two numeric variables into a third one.

\begin{verbatim}
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
The function adds the variables \text{var1} and \text{var2} into the result variable \text{result}. The function returns 0 on success and -1 in case of error.
\end{verbatim}

\textbf{PGTYPESnumeric_sub}

Subtract two numeric variables and return the result in a third one.

\begin{verbatim}
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
The function subtracts the variable \text{var2} from the variable \text{var1}. The result of the operation is stored in the variable \text{result}. The function returns 0 on success and -1 in case of error.
\end{verbatim}

\textbf{PGTYPESnumeric_mul}

Multiply two numeric variables and return the result in a third one.

\begin{verbatim}
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
The function multiplies the variables \text{var1} and \text{var2}. The result of the operation is stored in the variable \text{result}. The function returns 0 on success and -1 in case of error.
\end{verbatim}

\textbf{PGTYPESnumeric_div}

Divide two numeric variables and return the result in a third one.

\begin{verbatim}
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
The function divides the variables \text{var1} by \text{var2}. The result of the operation is stored in the variable \text{result}. The function returns 0 on success and -1 in case of error.
\end{verbatim}

\textbf{PGTYPESnumeric_cmp}

Compare two numeric variables.

\begin{verbatim}
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
This function compares two numeric variables. In case of error, \text{INT\_MAX} is returned. On success, the function returns one of three possible results:
\begin{itemize}
  \item 1, if \text{var1} is bigger than \text{var2}
  \item -1, if \text{var1} is smaller than \text{var2}
  \item 0, if \text{var1} and \text{var2} are equal
\end{itemize}
\end{verbatim}

\textbf{PGTYPESnumeric_from_int}

Convert an int variable to a numeric variable.

\begin{verbatim}
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
\end{verbatim}
Chapter 32. ECPG - Embedded SQL in C

This function accepts a variable of type signed int and stores it in the numeric variable var. Upon success, 0 is returned and -1 in case of a failure.

PGTYPESnumeric_from_long

Convert a long int variable to a numeric variable.

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

This function accepts a variable of type signed long int and stores it in the numeric variable var. Upon success, 0 is returned and -1 in case of a failure.

PGTYPESnumeric_copy

Copy over one numeric variable into another one.

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

This function copies over the value of the variable that src points to into the variable that dst points to. It returns 0 on success and -1 if an error occurs.

PGTYPESnumeric_from_double

Convert a variable of type double to a numeric.

```
int PGTYPESnumeric_from_double(double d, numeric *dst);
```

This function accepts a variable of type double and stores the result in the variable that dst points to. It returns 0 on success and -1 if an error occurs.

PGTYPESnumeric_to_double

Convert a variable of type numeric to double.

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

The function converts the numeric value from the variable that nv points to into the double variable that dp points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable errno will be set to PGTYPES_NUM_OVERFLOW additionally.

PGTYPESnumeric_to_int

Convert a variable of type numeric to int.

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

The function converts the numeric value from the variable that nv points to into the integer variable that ip points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable errno will be set to PGTYPES_NUM_OVERFLOW additionally.

PGTYPESnumeric_to_long

Convert a variable of type numeric to long.

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

The function converts the numeric value from the variable that nv points to into the long integer variable that lp points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable errno will be set to PGTYPES_NUM_OVERFLOW additionally.

PGTYPESnumeric_to_decimal

Convert a variable of type numeric to decimal.

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

The function converts the numeric value from the variable that src points to into the decimal variable that dst points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable errno will be set to PGTYPES_NUM_OVERFLOW additionally.
PGTYPESnumeric_from_decimal

Convert a variable of type decimal to numeric.

int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);

The function converts the decimal value from the variable that src points to into the numeric variable that dst points to. It returns 0 on success and -1 if an error occurs. Since the decimal type is implemented as a limited version of the numeric type, overflow cannot occur with this conversion.

32.8.2. The date type

The date type in C enables your programs to deal with data of the SQL type date. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the date type:

PGTYPESdate_from_timestamp

Extract the date part from a timestamp.

date PGTYPESdate_from_timestamp(timestamp dt);

The function receives a timestamp as its only argument and returns the extracted date part from this timestamp.

PGTYPESdate_from_asc

Parse a date from its textual representation.

date PGTYPESdate_from_asc(char *str, char **endptr);

The function receives a C char* string str and a pointer to a C char* string endptr. At the moment ecpg always parses the complete string and so it currently does not support to store the address of the first invalid character in *endptr. You can safely set endptr to NULL.

Note that the function always assumes MDY-formatted dates and there is currently no variable to change that within ecpg.

The following input formats are allowed:

Table 32-1. Valid input formats for PGTYPESdate_from_asc

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 8, 1999</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>1999-01-08</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>1/8/1999</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>1/18/1999</td>
<td>January 18, 1999</td>
</tr>
<tr>
<td>01/02/03</td>
<td>February 1, 2003</td>
</tr>
<tr>
<td>1999-Jan-08</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>Jan-08-1999</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>08-Jan-1999</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>99-Jan-08</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>08-Jan-99</td>
<td>January 8, 1999</td>
</tr>
<tr>
<td>08-Jan-06</td>
<td>January 8, 2006</td>
</tr>
<tr>
<td>Jan-08-99</td>
<td>January 8, 1999</td>
</tr>
</tbody>
</table>
Chapter 32. ECPG - Embedded SQL in C

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>19990108</td>
<td>ISO 8601; January 8, 1999</td>
</tr>
<tr>
<td>990108</td>
<td>ISO 8601; January 8, 1999</td>
</tr>
<tr>
<td>1999.008</td>
<td>year and day of year</td>
</tr>
<tr>
<td>J2451187</td>
<td>Julian day</td>
</tr>
<tr>
<td>January 8, 99 BC</td>
<td>year 99 before the Common Era</td>
</tr>
</tbody>
</table>

PGTYPESdate_to_asc

Return the textual representation of a date variable.

```c
char *PGTYPESdate_to_asc(date dDate);
```

The function receives the date `dDate` as its only parameter. It will output the date in the form `1999-01-18`, i.e. in the `YYYY-MM-DD` format.

PGTYPESdate_julmdy

Extract the values for the day, the month and the year from a variable of type date.

```c
void PGTYPESdate_julmdy(date d, int *mdy);
```

The function receives the date `d` and a pointer to an array of 3 integer values `mdy`. The variable name indicates the sequential order: `mdy[0]` will be set to contain the number of the month, `mdy[1]` will be set to the value of the day and `mdy[2]` will contain the year.

PGTYPESdate_mdyjul

Create a date value from an array of 3 integers that specify the day, the month and the year of the date.

```c
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

The function receives the array of the 3 integers (`mdy`) as its first argument and as its second argument a pointer to a variable of type date that should hold the result of the operation.

PGTYPESdate_dayofweek

Return a number representing the day of the week for a date value.

```c
int PGTYPESdate_dayofweek(date d);
```

The function receives the date variable `d` as its only argument and returns an integer that indicates the day of the week for this date.

- 0 - Sunday
- 1 - Monday
- 2 - Tuesday
- 3 - Wednesday
- 4 - Thursday
- 5 - Friday
- 6 - Saturday

PGTYPESdate_today

Get the current date.

```c
void PGTYPESdate_today(date *d);
```

The function receives a pointer to a date variable (`d`) that it sets to the current date.
PGTYPESdate_fmt_asc

Convert a variable of type date to its textual representation using a format mask.

```c
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

The function receives the date to convert (dDate), the format mask (fmtstring) and the string that will hold the textual representation of the date (outbuf).

On success, 0 is returned and a negative value if an error occurred.

The following literals are the field specifiers you can use:

- `dd` - The number of the day of the month.
- `mm` - The number of the month of the year.
- `yy` - The number of the year as a two digit number.
- `yyyy` - The number of the year as a four digit number.
- `ddd` - The name of the day (abbreviated).
- `mmm` - The name of the month (abbreviated).

All other characters are copied 1:1 to the output string.

The following table indicates a few possible formats. This will give you an idea of how to use this function. All output lines are based on the same date: November, 23rd, 1959.

<table>
<thead>
<tr>
<th>fmt</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mnddyy</td>
<td>112359</td>
</tr>
<tr>
<td>ddmmyy</td>
<td>231159</td>
</tr>
<tr>
<td>yymmdd</td>
<td>591123</td>
</tr>
<tr>
<td>yy/mm/dd</td>
<td>59/11/23</td>
</tr>
<tr>
<td>yy mm dd</td>
<td>59 11 23</td>
</tr>
<tr>
<td>yyyy dd mm</td>
<td>1959 23 11</td>
</tr>
<tr>
<td>.mm.yyyy.dd.</td>
<td>.11.1959.23.</td>
</tr>
<tr>
<td>mmm. dd, yyyy</td>
<td>Nov. 23, 1959</td>
</tr>
<tr>
<td>mmm dd yyyy</td>
<td>Nov 23 1959</td>
</tr>
<tr>
<td>yyyy dd mm</td>
<td>1959 23 11</td>
</tr>
<tr>
<td>(ddd) mmm. dd, yyyy</td>
<td>Mon, Nov. 23, 1959</td>
</tr>
</tbody>
</table>

PGTYPESdate_defmt_asc

Use a format mask to convert a C char* string to a value of type date.

```c
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

The function receives a pointer to the date value that should hold the result of the operation (d), the format mask to use for parsing the date (fmt) and the C char* string containing the textual representation of the date (str). The textual representation is expected to match the format mask. However you do not need to have a 1:1 mapping of the string to the format mask. The function only analyzes the sequential order and looks for the literals `yy` or `yyyy` that indicate the position of the year, `mm` to indicate the position of the month and `dd` to indicate the position of the day.
The following table indicates a few possible formats. This will give you an idea of how to use this function.

Table 32-3. Valid input formats for \texttt{rdefmtdate}

<table>
<thead>
<tr>
<th>fmt</th>
<th>str</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddmmyy</td>
<td>21-2-54</td>
<td>1954-02-21</td>
</tr>
<tr>
<td>ddmmyy</td>
<td>2-12-54</td>
<td>1954-12-02</td>
</tr>
<tr>
<td>ddmmyy</td>
<td>20111954</td>
<td>1954-11-20</td>
</tr>
<tr>
<td>ddmmyy</td>
<td>130464</td>
<td>1964-04-13</td>
</tr>
<tr>
<td>mmm.dd.yyyy</td>
<td>MAR-12-1967</td>
<td>1967-03-12</td>
</tr>
<tr>
<td>yy/mm/dd</td>
<td>1954, February 3rd</td>
<td>1954-02-03</td>
</tr>
<tr>
<td>mmm.dd.yyyy</td>
<td>041269</td>
<td>1969-04-12</td>
</tr>
<tr>
<td>yy/mm/dd</td>
<td>In the year 2525, in</td>
<td>2525-07-28</td>
</tr>
<tr>
<td></td>
<td>the month of July,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mankind will be alive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>on the 28th day</td>
<td></td>
</tr>
<tr>
<td>dd-mm-yy</td>
<td>I said on the 28th of</td>
<td>2525-07-28</td>
</tr>
<tr>
<td></td>
<td>July in the year 2525</td>
<td></td>
</tr>
<tr>
<td>mmm.dd.yyyy</td>
<td>9/14/58</td>
<td>1958-09-14</td>
</tr>
<tr>
<td>yy/mm/dd</td>
<td>47/03/29</td>
<td>1947-03-29</td>
</tr>
<tr>
<td>mmm_dd.yyyy</td>
<td>oct 28 1975</td>
<td>1975-10-28</td>
</tr>
<tr>
<td>mmddyy</td>
<td>Nov 14th, 1985</td>
<td>1985-11-14</td>
</tr>
</tbody>
</table>

32.8.3. The timestamp type

The timestamp type in C enables your programs to deal with data of the SQL type timestamp. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the timestamp type:

\texttt{PGTYPEStimestamp\_from\_asc}

Parse a timestamp from its textual representation into a timestamp variable.

\begin{verbatim}
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
\end{verbatim}

The function receives the string to parse (\texttt{str}) and a pointer to a C char* (\texttt{endptr}). At the moment ecpg always parses the complete string and so it currently does not support to store the address of the first invalid character in \texttt{*endptr}. You can safely set \texttt{endptr} to NULL.

The function returns the parsed timestamp on success. On error, \texttt{PGTYPESInvalidTimestamp} is returned and \texttt{errno} is set to \texttt{PGTYPES\_TS\_BAD\_TIMESTAMP}. See \texttt{PGTYPESInvalidTimestamp} for important notes on this value.

In general, the input string can contain any combination of an allowed date specification, a whitespace character and an allowed time specification. Note that timezones are not supported by ecpg. It can parse them but does not apply any calculation as the PostgreSQL server does for example. Timezone specifiers are silently discarded.
The following table contains a few examples for input strings:

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999-01-08 04:05:06</td>
<td>1999-01-08 04:05:06</td>
</tr>
<tr>
<td>January 8 04:05:06 1999 PST</td>
<td>1999-01-08 04:05:06</td>
</tr>
<tr>
<td>1999-Jan-08 04:05:06.789-8</td>
<td>1999-01-08 04:05:06.789 (time zone specifier ignored)</td>
</tr>
<tr>
<td>J2451187 04:05-08:00</td>
<td>1999-01-08 04:05:00 (time zone specifier ignored)</td>
</tr>
</tbody>
</table>

**PGTYPEStimestamp_to_asc**

Converts a date to a C char* string.

```c
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

The function receives the timestamp `tstamp` as its only argument and returns an allocated string that contains the textual representation of the timestamp.

**PGTYPEStimestamp_current**

Retrieve the current timestamp.

```c
void PGTYPEStimestamp_current(timestamp *ts);
```

The function retrieves the current timestamp and saves it into the timestamp variable that `ts` points to.

**PGTYPEStimestamp_fmt_asc**

Convert a timestamp variable to a C char* using a format mask.

```c
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

The function receives a pointer to the timestamp to convert as its first argument (`ts`), a pointer to the output buffer (`output`), the maximal length that has been allocated for the output buffer (`str_len`) and the format mask to use for the conversion (`fmtstr`).

Upon success, the function returns 0 and a negative value if an error occurred.

You can use the following format specifiers for the format mask. The format specifiers are the same ones that are used in the `strftime` function in libc. Any non-format specifier will be copied into the output buffer.

- `%A` - is replaced by national representation of the full weekday name.
- `%a` - is replaced by national representation of the abbreviated weekday name.
- `%B` - is replaced by national representation of the full month name.
- `%b` - is replaced by national representation of the abbreviated month name.
- `%C` - is replaced by (year / 100) as decimal number; single digits are preceded by a zero.
- `%c` - is replaced by national representation of time and date.
- `%d` - is equivalent to `%m/%d/%y.`
- `%e` - is replaced by the day of the month as a decimal number (01-31).
- `%F` - POSIX locale extensions. The sequences `%Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %OM %OS %Ou %OU %OV %OW %Oy` are supposed to provide alternative representations.
Additionally %OB implemented to represent alternative months names (used standalone, without day mentioned).

- %e - is replaced by the day of month as a decimal number (1-31); single digits are preceded by a blank.
- %F - is equivalent to %Y-%m-%d.
- %G - is replaced by a year as a decimal number with century. This year is the one that contains the greater part of the week (Monday as the first day of the week).
- %g - is replaced by the same year as in %G, but as a decimal number without century (00-99).
- %H - is replaced by the hour (24-hour clock) as a decimal number (00-23).
- %h - the same as %b.
- %I - is replaced by the hour (12-hour clock) as a decimal number (01-12).
- %j - is replaced by the day of the year as a decimal number (001-366).
- %k - is replaced by the hour (24-hour clock) as a decimal number (0-23); single digits are preceded by a blank.
- %l - is replaced by the hour (12-hour clock) as a decimal number (1-12); single digits are preceded by a blank.
- %M - is replaced by the minute as a decimal number (00-59).
- %m - is replaced by the month as a decimal number (01-12).
- %n - is replaced by a newline.
- %O* - the same as %E*.
- %p - is replaced by national representation of either "ante meridiem" or "post meridiem" as appropriate.
- %R - is equivalent to %H:%M.
- %T - is equivalent to %H:%M:%S
- %t - is replaced by a tab.
- %u - is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
- %w - is replaced by the weekday (Monday as the first day of the week) as a decimal number (1-7).
- %V - is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (01-53). If the week containing January 1 has four or more days in the new year, then it is week 1; otherwise it is the last week of the previous year, and the next week is week 1.
- %v - is equivalent to %e-%b-%Y.
- %W - is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
- %w - is replaced by the weekday (Sunday as the first day of the week) as a decimal number (0-6).
Chapter 32. ECPG - Embedded SQL in C

• %X - is replaced by national representation of the time.
• %x - is replaced by national representation of the date.
• %Y - is replaced by the year with century as a decimal number.
• %y - is replaced by the year without century as a decimal number (00-99).
• %z - is replaced by the time zone name.
• %Z - is replaced by the time zone offset from UTC; a leading plus sign stands for east of UTC, a minus sign for west of UTC, hours and minutes follow with two digits each and no delimiter between them (common form for RFC 822 date headers).
• +% - is replaced by national representation of the date and time.
• %-- - GNU libc extension. Do not do any padding when performing numerical outputs.
• %*- - GNU libc extension. Explicitly specify space for padding.
• %0*- - GNU libc extension. Explicitly specify zero for padding.
• %% - is replaced by %.

PGTYPEStimestamp_sub
Subtract one timestamp from another one and save the result in a variable of type interval.
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
The function will subtract the timestamp variable that ts2 points to from the timestamp variable that ts1 points to and will store the result in the interval variable that iv points to.
Upon success, the function returns 0 and a negative value if an error occurred.

PGTYPEStimestamp_defmt_asc
Parse a timestamp value from its textual representation using a formatting mask.
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
The function receives the textual representation of a timestamp in the variable str as well as the formatting mask to use in the variable fmt. The result will be stored in the variable that d points to.
If the formatting mask fmt is NULL, the function will fall back to the default formatting mask which is %Y-%m-%d %H:%M:%S.
This is the reverse function to PGTYPEStimestamp_fmt_asc. See the documentation there in order to find out about the possible formatting mask entries.

PGTYPEStimestamp_add_interval
Add an interval variable to a timestamp variable.
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
The function receives a pointer to a timestamp variable tin and a pointer to an interval variable span. It adds the interval to the timestamp and saves the resulting timestamp in the variable that tout points to.
Upon success, the function returns 0 and a negative value if an error occurred.

PGTYPEStimestamp_sub_interval
Subtract an interval variable from a timestamp variable.
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
The function subtracts the interval variable that span points to from the timestamp variable that tin points to and saves the result into the variable that tout points to.
Upon success, the function returns 0 and a negative value if an error occurred.

### 32.8.4. The interval type

The interval type in C enables your programs to deal with data of the SQL type interval. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the interval type:

- **PGTYPESinterval_new**
  
  Return a pointer to a newly allocated interval variable.
  
  ```c
  interval *PGTYPESinterval_new(void);
  ```

- **PGTYPESinterval_free**
  
  Release the memory of a previously allocated interval variable.
  
  ```c
  void PGTYPESinterval_new(interval *intvl);
  ```

- **PGTYPESinterval_from_asc**
  
  Parse an interval from its textual representation.
  
  ```c
  interval *PGTYPESinterval_from_asc(char *str, char **endptr);
  ```

  The function parses the input string `str` and returns a pointer to an allocated interval variable. At the moment ecpg always parses the complete string and so it currently does not support to store the address of the first invalid character in `endptr`. You can safely set `endptr` to `NULL`.

- **PGTYPESinterval_to_asc**
  
  Convert a variable of type interval to its textual representation.
  
  ```c
  char *PGTYPESinterval_to_asc(interval *span);
  ```

  The function converts the interval variable that `span` points to into a C char*. The output looks like this example: @ 1 day 12 hours 59 mins 10 secs.

- **PGTYPESinterval_copy**
  
  Copy a variable of type interval.
  
  ```c
  int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
  ```

  The function copies the interval variable that `intvlsrc` points to into the variable that `intvldest` points to. Note that you need to allocate the memory for the destination variable before.

### 32.8.5. The decimal type

The decimal type is similar to the numeric type. However it is limited to a maximal precision of 30 significant digits. In contrast to the numeric type which can be created on the heap only, the decimal type can be created either on the stack or on the heap (by means of the functions **PGTYPESdecimal_new()** and **PGTYPESdecimal_free()**). There are a lot of other functions that deal with the decimal type in the Informix compatibility mode described in Section 32.9.

The following functions can be used to work with the decimal type and are not only contained in the `libcompat` library.
Chapter 32. ECPG - Embedded SQL in C

PGTYPESdecimal_new
   Request a pointer to a newly allocated decimal variable.
   
   decimal *PGTYPESdecimal_new(void);

PGTYPESdecimal_free
   Free a decimal type, release all of its memory.
   
   void PGTYPESdecimal_free(decimal *var);

32.8.6. errno values of pgtypeslib

PGTYPES_NUM_BAD_NUMERIC
   An argument should contain a numeric variable (or point to a numeric variable) but in fact its
   in-memory representation was invalid.

PGTYPES_NUM_OVERFLOW
   An overflow occurred. Since the numeric type can deal with almost arbitrary precision, convert-
   ing a numeric variable into other types might cause overflow.

PGTYPES_NUM_OVERFLOW
   An underflow occurred. Since the numeric type can deal with almost arbitrary precision, con-
   verting a numeric variable into other types might cause underflow.

PGTYPES_NUM_DIVIDE_ZERO
   A division by zero has been attempted.

PGTYPES_DATE_BAD_DATE

PGTYPES_DATE_ERR_EARGS

PGTYPES_DATE_ERR_ENOSHORTDATE

PGTYPES_DATE_ERR_ENOTDMY

PGTYPES_DATE_BAD_DAY

PGTYPES_DATE_BAD_MONTH

PGTYPES_DATE_BAD_DATE

PGTYPES_TS_BAD_TIMESTAMP

32.8.7. Special constants of pgtypeslib

PGTYPESInvalidTimestamp
   A value of type timestamp representing an invalid time stamp. This is returned by the function
   PGTYPEStimestamp_from_asc on parse error. Note that due to the internal representation of
the timestamp datatype, `PGTYPESInvalidTimestamp` is also a valid timestamp at the same time. It is set to 1899-12-31 23:59:59. In order to detect errors, make sure that your application does not only test for `PGTYPESInvalidTimestamp` but also for `errno != 0` after each call to `PGTYPEStimestamp_from_asc`.

### 32.9. Informix compatibility mode

ecpg can be run in a so-called Informix compatibility mode. If this mode is active, it tries to behave as if it were the Informix precompiler for Informix E/SQL. Generally spoken this will allow you to use the dollar sign instead of the `EXEC SQL` primitive to introduce embedded SQL commands:

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

There are two compatibility modes: INFORMIX, INFORMIX_SE

When linking programs that use this compatibility mode, remember to link against `libcompat` that is shipped with ecpg.

Besides the previously explained syntactic sugar, the Informix compatibility mode ports some functions for input, output and transformation of data as well as embedded SQL statements known from E/SQL to ecpg.

Informix compatibility mode is closely connected to the pgtypeslib library of ecpg. pgtypeslib maps SQL data types to data types within the C host program and most of the additional functions of the Informix compatibility mode allow you to operate on those C host program types. Note however that the extent of the compatibility is limited. It does not try to copy Informix behaviour; it allows you to do more or less the same operations and gives you functions that have the same name and the same basic behavior but it is no drop-in replacement if you are using Informix at the moment. Moreover, some of the data types are different. For example, PostgreSQL’s datetime and interval types do not know about ranges like for example `YEAR TO MINUTE` so you won’t find support in ecpg for that either.

### 32.9.1. Additional embedded SQL statements

**CLOSE DATABASE**

This statement closes the current connection. In fact, this is a synonym for ecpg’s `DISCONNECT CURRENT`:

```
$CLOSE DATABASE;  /* close the current connection */
EXEC SQL CLOSE DATABASE;
```
### 32.9.2. Additional functions

#### decadd

Add two decimal type values.

```c
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

The function receives a pointer to the first operand of type decimal (`arg1`), a pointer to the second operand of type decimal (`arg2`) and a pointer to a value of type decimal that will contain the sum (`sum`). On success, the function returns 0. ECPG_INFORMIX_NUM_OVERFLOW is returned in case of overflow and ECPG_INFORMIX_NUM_UNDERFLOW in case of underflow. -1 is returned for other failures and errno is set to the respective errno number of the pgtypeslib.

#### deccmp

Compare two variables of type decimal.

```c
int deccmp(decimal *arg1, decimal *arg2);
```

The function receives a pointer to the first decimal value (`arg1`), a pointer to the second decimal value (`arg2`) and returns an integer value that indicates which is the bigger value.

- 1, if the value that `arg1` points to is bigger than the value that `arg2` points to
- -1, if the value that `arg1` points to is smaller than the value that `arg2` points to
- 0, if the value that `arg1` points to and the value that `arg2` points to are equal

#### deccopy

Copy a decimal value.

```c
void deccopy(decimal *src, decimal *target);
```

The function receives a pointer to the decimal value that should be copied as the first argument (`src`) and a pointer to the target structure of type decimal (`target`) as the second argument.

#### deccvasc

Convert a value from its ASCII representation into a decimal type.

```c
int deccvasc(char *cp, int len, decimal *np);
```

The function receives a pointer to string that contains the string representation of the number to be converted (`cp`) as well as its length (`len`). `np` is a pointer to the decimal value that saves the result of the operation.

Valid formats are for example: `-2.794`, `+3.44`, `592.49E07` or `-32.84e-4`.

The function returns 0 on success. If overflow or underflow occurred, ECPG_INFORMIX_NUM_OVERFLOW or ECPG_INFORMIX_NUM_UNDERFLOW is returned. If the ASCII representation could not be parsed, ECPG_INFORMIX_BAD_NUMERIC is returned or ECPG_INFORMIX_BAD_EXPONENT if this problem occurred while parsing the exponent.

#### deccvdbl

Convert a value of type double to a value of type decimal.

```c
int deccvdbl(double dbl, decimal *np);
```

The function receives the variable of type double that should be converted as its first argument (`dbl`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.
deccvint

Convert a value of type int to a value of type decimal.

```c
int deccvint(int in, decimal *np);
```

The function receives the variable of type int that should be converted as its first argument (`in`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.

deccvlong

Convert a value of type long to a value of type decimal.

```c
int deccvlong(long lng, decimal *np);
```

The function receives the variable of type long that should be converted as its first argument (`lng`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.

decdiv

Divide two variables of type decimal.

```c
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1/n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the division fails. If overflow or underflow occurred, the function returns ECPG_INFORMIX_NUM_OVERFLOW or ECPG_INFORMIX_NUM_UNDERFLOW respectively. If an attempt to divide by zero is observed, the function returns ECPG_INFORMIX_DIVIDE_ZERO.

decmul

Multiply two decimal values.

```c
int decmul(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1*n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the multiplication fails. If overflow or underflow occurred, the function returns ECPG_INFORMIX_NUM_OVERFLOW or ECPG_INFORMIX_NUM_UNDERFLOW respectively.

decsub

Subtract one decimal value from another.

```c
int decsub(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1-n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the subtraction fails. If overflow or underflow occurred, the function returns ECPG_INFORMIX_NUM_OVERFLOW or ECPG_INFORMIX_NUM_UNDERFLOW respectively.
dectoasc

Convert a variable of type decimal to its ASCII representation in a C char* string.

```c
int dectoasc(decimal *np, char *cp, int len, int right)
```

The function receives a pointer to a variable of type decimal (`np`) that it converts to its textual representation. `cp` is the buffer that should hold the result of the operation. The parameter `right` specifies, how many digits right of the decimal point should be included in the output. The result will be rounded to this number of decimal digits. Setting `right` to -1 indicates that all available decimal digits should be included in the output. If the length of the output buffer, which is indicated by `len` is not sufficient to hold the textual representation including the trailing NUL character, only a single * character is stored in the result and -1 is returned.

The function returns either -1 if the buffer `cp` was too small or `ECPG_INFORMIX_OUT_OF_MEMORY` if memory was exhausted.

dectodbl

Convert a variable of type decimal to a double.

```c
int dectodbl(decimal *np, double *dblp);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the double variable that should hold the result of the operation (`dblp`).

On success, 0 is returned and a negative value if the conversion failed.

dectoint

Convert a variable to type decimal to an integer.

```c
int dectoint(decimal *np, int *ip);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the integer variable that should hold the result of the operation (`ip`).

On success, 0 is returned and a negative value if the conversion failed. If an overflow occurred, `ECPG_INFORMIX_NUM_OVERFLOW` is returned.

Note that the `ecpg` implementation differs from the Informix implementation. Informix limits an integer to the range from -32767 to 32767, while the limits in the `ecpg` implementation depend on the architecture (`-INT_MAX .. INT_MAX`).

dectolong

Convert a variable to type decimal to a long integer.

```c
int dectolong(decimal *np, long *lngp);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the long variable that should hold the result of the operation (`lngp`).

On success, 0 is returned and a negative value if the conversion failed. If an overflow occurred, `ECPG_INFORMIX_NUM_OVERFLOW` is returned.

Note that the `ecpg` implementation differs from the Informix implementation. Informix limits a long integer to the range from -2,147,483,647 to 2,147,483,647, while the limits in the `ecpg` implementation depend on the architecture (`-LONG_MAX .. LONG_MAX`).

rdatestr

Converts a date to a C char* string.

```c
int rdatestr(date d, char *str);
```

The function receives two arguments, the first one is the date to convert (`d`) and the second one is a pointer to the target string. The output format is always `yyyy-mm-dd`, so you need to allocate at least 11 bytes (including the NUL-terminator) for the string.
The function returns 0 on success and a negative value in case of error.

Note that ecpg’s implementation differs from the Informix implementation. In Informix the format can be influenced by setting environment variables. In ecpg however, you cannot change the output format.

rstrdate
Parse the textual representation of a date.

```c
int rstrdate(char *str, date *d);
```
The function receives the textual representation of the date to convert (str) and a pointer to a variable of type date (d). This function does not allow you to specify a format mask. It uses the default format mask of Informix which is mm/dd/yyyy. Internally, this function is implemented by means of rdefmtdate. Therefore, rstrdate is not faster and if you have the choice you should opt for rdefmtdate which allows you to specify the format mask explicitly.

The function returns the same values as rdefmtdate.

rtoday
Get the current date.

```c
void rtoday(date *d);
```
The function receives a pointer to a date variable (d) that it sets to the current date.

Internally this function uses the PGTYPESdate_today function.

rjulmdy
Extract the values for the day, the month and the year from a variable of type date.

```c
int rjulmdy(date d, short mdy[3]);
```
The function receives the date d and a pointer to an array of 3 short integer values mdy. The variable name indicates the sequential order: mdy[0] will be set to contain the number of the month, mdy[1] will be set to the value of the day and mdy[2] will contain the year.

The function always returns 0 at the moment.

Internally the function uses the PGTYPESdate_julmdy function.

rdefmtdate
Use a format mask to convert a character string to a value of type date.

```c
int rdefmtdate(date *d, char *fmt, char *str);
```
The function receives a pointer to the date value that should hold the result of the operation (d), the format mask to use for parsing the date (fmt) and the C char* string containing the textual representation of the date (str). The textual representation is expected to match the format mask. However you do not need to have a 1:1 mapping of the string to the format mask. The function only analyzes the sequential order and looks for the literals yy or yyyy that indicate the position of the year, mm to indicate the position of the month and dd to indicate the position of the day.

The function returns the following values:

- 0 - The function terminated successfully.
- ECPG_INFORMIX_ENOSHORTDATE - The date does not contain delimiters between day, month and year. In this case the input string must be exactly 6 or 8 bytes long but isn’t.
- ECPG_INFORMIX_ENOTDMY - The format string did not correctly indicate the sequential order of year, month and day.
- ECPG_INFORMIX_BAD_DAY - The input string does not contain a valid day.
- ECPG_INFORMIX_BAD_MONTH - The input string does not contain a valid month.
Chapter 32. ECPG - Embedded SQL in C

- ECPG_INFORMIX_BAD_YEAR - The input string does not contain a valid year.

Internally this function is implemented to use the `PGTYPESdate_defmt_asc` function. See the reference there for a table of example input.

rfmtdate
Convert a variable of type date to its textual representation using a format mask.

```c
int rfmtdate(date d, char *fmt, char *str);
```

The function receives the date to convert (`d`), the format mask (`fmt`) and the string that will hold the textual representation of the date (`str`).

On success, 0 is returned and a negative value if an error occurred.

Internally this function uses the `PGTYPESdate_fmt_asc` function, see the reference there for examples.

rmdyjul
Create a date value from an array of 3 short integers that specify the day, the month and the year of the date.

```c
int rmdyjul(short mdy[3], date *d);
```

The function receives the array of the 3 short integers (`mdy`) and a pointer to a variable of type `date` that should hold the result of the operation.

Currently the function returns always 0.

Internally the function is implemented to use the function `PGTYPESdate_mdyjul`.

rdayofweek
Return a number representing the day of the week for a date value.

```c
int rdayofweek(date d);
```

The function receives the date variable `d` as its only argument and returns an integer that indicates the day of the week for this date.

- 0 - Sunday
- 1 - Monday
- 2 - Tuesday
- 3 - Wednesday
- 4 - Thursday
- 5 - Friday
- 6 - Saturday

Internally the function is implemented to use the function `PGTYPESdate_dayofweek`.

dtcurrent
Retrieve the current timestamp.

```c
void dtcurrent(timestamp *ts);
```

The function retrieves the current timestamp and saves it into the timestamp variable that `ts` points to.
Chapter 32. ECPG - Embedded SQL in C

**dtcvasc**

Parses a timestamp from its textual representation in ANSI standard into a timestamp variable.

```c
int dtcvasc(char *str, timestamp *ts);
```

The function receives the string to parse (`str`) and a pointer to the timestamp variable that should hold the result of the operation (`ts`).

The function returns 0 on success and a negative value in case of error.

Internally this function uses the `PGTYPEStimestamp_from_asc` function. See the reference there for a table with example inputs.

**dtcvfmtasc**

Parses a timestamp from its textual representation in ANSI standard using a format mask into a timestamp variable.

```c
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

The function receives the string to parse (`inbuf`), the format mask to use (`fmtstr`) and a pointer to the timestamp variable that should hold the result of the operation (`ts`).

This function is implemented by means of the `PGTYPEStimestamp_defmt_asc`. See the documentation there for a list of format specifiers that can be used.

The function returns 0 on success and a negative value in case of error.

**dtsub**

Subtract one timestamp from another and return a variable of type interval.

```c
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

The function will subtract the timestamp variable that `ts2` points to from the timestamp variable that `ts1` points to and will store the result in the interval variable that `iv` points to.

Upon success, the function returns 0 and a negative value if an error occurred.

**dttoasc**

Convert a timestamp variable to a C char* string.

```c
int dttoasc(timestamp *ts, char *output);
```

The function receives a pointer to the timestamp variable to convert (`ts`) and the string that should hold the result of the operation (`output`). It converts `ts` to its textual representation in the ANSI SQL standard which is defined to be `YYYY-MM-DD HH:MM:SS`.

Upon success, the function returns 0 and a negative value if an error occurred.

**dttofmtasc**

Convert a timestamp variable to a C char* using a format mask.

```c
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

The function receives a pointer to the timestamp variable to convert as its first argument (`ts`), a pointer to the output buffer (`output`), the maximal length that has been allocated for the output buffer (`str_len`) and the format mask to use for the conversion (`fmtstr`).

Upon success, the function returns 0 and a negative value if an error occurred.

Internally, this function uses the `PGTYPEStimestamp_fmt_asc` function. See the reference there for information on what format mask specifiers can be used.

**intoasc**

Convert an interval variable to a C char* string.

```c
int intoasc(interval *i, char *str);
```
The function receives a pointer to the interval variable to convert (i) and the string that should hold the result of the operation str). It converts i to its textual representation in the ANSI SQL standard which is defined to be YYYY-MM-DD HH:MM:SS.

Upon success, the function returns 0 and a negative value if an error occurred.

\textbf{rfmtlong}

Convert a long integer value to its textual representation using a format mask.

\begin{verbatim}
int rfmtlong(long lng_val, char *fmt, char *outbuf);
\end{verbatim}

The function receives the long value lng_val, the format mask fmt and a pointer to the output buffer outbuf. It converts the long value according to the format mask to its textual representation.

The format mask can be composed of the following format specifying characters:

\begin{itemize}
  \item * (asterisk) - if this position would be blank otherwise, fill it with an asterisk.
  \item & (ampersand) - if this position would be blank otherwise, fill it with a zero.
  \item # - turn leading zeroes into blanks.
  \item < - left-justify the number in the string.
  \item , (comma) - group numbers of four or more digits into groups of three digits separated by a comma.
  \item . (period) - this character separates the whole-number part of the number from the fractional part.
  \item - (minus) - the minus sign appears if the number is a negative value.
  \item + (plus) - the plus sign appears if the number is a positive value.
  \item ( - this replaces the minus sign in front of the negative number. The minus sign will not appear.
  \item ) - this character replaces the minus and is printed behind the negative value.
  \item $ - the currency symbol.
\end{itemize}

\textbf{rupshift}

Convert a string to upper case.

\begin{verbatim}
void rupshift(char *str);
\end{verbatim}

The function receives a pointer to the string and transforms every lower case character to upper case.

\textbf{byleng}

Return the number of characters in a string without counting trailing blanks.

\begin{verbatim}
int byleng(char *str, int len);
\end{verbatim}

The function expects a fixed-length string as its first argument (str) and its length as its second argument (len). It returns the number of significant characters, that is the length of the string without trailing blanks.

\textbf{ldchar}

Copy a fixed-length string into a null-terminated string.

\begin{verbatim}
void ldchar(char *src, int len, char *dest);
\end{verbatim}

The function receives the fixed-length string to copy (src), its length (len) and a pointer to the destination memory (dest). Note that you need to reserve at least len+1 bytes for the string.
that dest points to. The function copies at most len bytes to the new location (less if the source string has trailing blanks) and adds the null-terminator.

\[ \text{rgetmsg} \]

int rgetmsg(int msgnum, char *s, int maxsize);
This function exists but is not implemented at the moment!

\[ \text{rtypalign} \]

int rtypalign(int offset, int type);
This function exists but is not implemented at the moment!

\[ \text{rtypmsize} \]

int rtypmsize(int type, int len);
This function exists but is not implemented at the moment!

\[ \text{rtypwidth} \]

int rtypwidth(int sqltype, int sqllen);
This function exists but is not implemented at the moment!

\[ \text{rsetnull} \]

Set a variable to NULL.
int rsetnull(int t, char *ptr);
The function receives an integer that indicates the type of the variable and a pointer to the variable itself that is casted to a C char* pointer.

The following types exist:
- CCHARTYPE - For a variable of type char or char*
- CSHORTTYPE - For a variable of type short int
- CINTTYPE - For a variable of type int
- CBOOLTYPE - For a variable of type boolean
- CFLOATTYPE - For a variable of type float
- CLONGTYPE - For a variable of type long
- CDOUBLETYPE - For a variable of type double
- CDECIMALTYPE - For a variable of type decimal
- CDATETYPE - For a variable of type date
- CDTIMETYPE - For a variable of type timestamp

Here is an example of a call to this function:

```c
$char c[] = "abc \n";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```
risnull

Test if a variable is NULL.

```c
int risnull(int t, char *ptr);
```

The function receives the type of the variable to test (t) as well a pointer to this variable (ptr). Note that the latter needs to be casted to a char*. See the function `rsetnull` for a list of possible variable types.

Here is an example of how to use this function:

```c
$char c[] = "abc ";
$short s = 17;
<int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

### 32.9.3. Additional constants

Note that all constants here describe errors and all of them are defined to represent negative values. In the descriptions of the different constants you can also find the value that the constants represent in the current implementation. However you should not rely on this number. You can however rely on the fact all of them are defined to represent negative values.

- **ECPG_INFORMIX_NUM_OVERFLOW**
  
  Functions return this value if an overflow occurred in a calculation. Internally it is defined to -1200 (the Informix definition).

- **ECPG_INFORMIX_NUM_UNDERFLOW**
  
  Functions return this value if an underflow occurred in a calculation. Internally it is defined to -1201 (the Informix definition).

- **ECPG_INFORMIX_DIVIDE_ZERO**
  
  Functions return this value if an attempt to divide by zero is observed. Internally it is defined to -1202 (the Informix definition).

- **ECPG_INFORMIX_BAD_YEAR**
  
  Functions return this value if a bad value for a year was found while parsing a date. Internally it is defined to -1204 (the Informix definition).

- **ECPG_INFORMIX_BAD_MONTH**
  
  Functions return this value if a bad value for a month was found while parsing a date. Internally it is defined to -1205 (the Informix definition).

- **ECPG_INFORMIX_BAD_DAY**
  
  Functions return this value if a bad value for a day was found while parsing a date. Internally it is defined to -1206 (the Informix definition).

- **ECPG_INFORMIX_ENOSHORTDATE**
  
  Functions return this value if a parsing routine needs a short date representation but did not get the date string in the right length. Internally it is defined to -1209 (the Informix definition).
Chapter 32. ECPG - Embedded SQL in C

32.10. Using SQL Descriptor Areas

An SQL descriptor area is a more sophisticated method for processing the result of a SELECT or FETCH statement. An SQL descriptor area groups the data of one row of data together with metadata items into one data structure. The metadata is particularly useful when executing dynamic SQL statements, where the nature of the result columns might not be known ahead of time.

An SQL descriptor area consists of a header, which contains information concerning the entire descriptor, and one or more item descriptor areas, which basically each describe one column in the result row.

Before you can use an SQL descriptor area, you need to allocate one:

EXEC SQL ALLOCATE DESCRIPTOR identifier;

The identifier serves as the “variable name” of the descriptor area. When you don’t need the descriptor anymore, you should deallocate it:

EXEC SQL DEALLOCATE DESCRIPTOR identifier;

To use a descriptor area, specify it as the storage target in an INTO clause, instead of listing host variables:
EXEC SQL FETCH NEXT FROM mycursor INTO DESCRIPTOR mydesc;

Now how do you get the data out of the descriptor area? You can think of the descriptor area as a structure with named fields. To retrieve the value of a field from the header and store it into a host variable, use the following command:

EXEC SQL GET DESCRIPTOR name :hostvar = field;

Currently, there is only one header field defined: COUNT, which tells how many item descriptor areas exist (that is, how many columns are contained in the result). The host variable needs to be of an integer type. To get a field from the item descriptor area, use the following command:

EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;

num can be a literal integer or a host variable containing an integer. Possible fields are:

CARDINALITY (integer)
   number of rows in the result set

DATA
   actual data item (therefore, the data type of this field depends on the query)

DATETIME_INTERVAL_CODE (integer)
   not implemented

DATETIME_INTERVAL_PRECISION (integer)
   not implemented

INDICATOR (integer)
   the indicator (indicating a null value or a value truncation)

KEY_MEMBER (integer)
   not implemented

LENGTH (integer)
   length of the datum in characters

NAME (string)
   name of the column

NULLABLE (integer)
   not implemented

OCTET_LENGTH (integer)
   length of the character representation of the datum in bytes

PRECISION (integer)
   precision (for type numeric)

RETURNED_LENGTH (integer)
   length of the datum in characters
32.11. Error Handling

This section describes how you can handle exceptional conditions and warnings in an embedded SQL program. There are several nonexclusive facilities for this.

32.11.1. Setting Callbacks

One simple method to catch errors and warnings is to set a specific action to be executed whenever a particular condition occurs. In general:

EXEC SQL WHENEVER condition action;

*condition* can be one of the following:

- **SQLERROR**
  The specified action is called whenever an error occurs during the execution of an SQL statement.

- **SQLWARNING**
  The specified action is called whenever a warning occurs during the execution of an SQL statement.

- **NOT FOUND**
  The specified action is called whenever an SQL statement retrieves or affects zero rows. (This condition is not an error, but you might be interested in handling it specially.)

*action* can be one of the following:

- **CONTINUE**
  This effectively means that the condition is ignored. This is the default.

- **GOTO label**

  Jump to the specified label (using a C goto statement).

- **SQLPRINT**

  Print a message to standard error. This is useful for simple programs or during prototyping. The details of the message cannot be configured.
STOP

Call `exit(1)`, which will terminate the program.

DO BREAK

Execute the C statement `break`. This should only be used in loops or `switch` statements.

CALL `name (args)`
DO `name (args)`

Call the specified C functions with the specified arguments.

The SQL standard only provides for the actions `CONTINUE` and `GOTO` (and `GO TO`).

Here is an example that you might want to use in a simple program. It prints a simple message when a warning occurs and aborts the program when an error happens:

```sql
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

The statement `EXEC SQL WHENEVER` is a directive of the SQL preprocessor, not a C statement. The error or warning actions that it sets apply to all embedded SQL statements that appear below the point where the handler is set, unless a different action was set for the same condition between the first `EXEC SQL WHENEVER` and the SQL statement causing the condition, regardless of the flow of control in the C program. So neither of the two following C program excerpts will have the desired effect:

```c
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```c
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```c
static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```
32.11.2. sqlca

For more powerful error handling, the embedded SQL interface provides a global variable with the name sqlca that has the following structure:

```c
struct {
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct {
        int sqlerrml;
        char sqlerrmc[70];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

(In a multithreaded program, every thread automatically gets its own copy of sqlca. This works similarly to the handling of the standard C global variable errno.)

sqlca covers both warnings and errors. If multiple warnings or errors occur during the execution of a statement, then sqlca will only contain information about the last one.

If no error occurred in the last SQL statement, sqlca.sqlcode will be 0 and sqlca.sqlstate will be "00000". If a warning or error occurred, then sqlca.sqlcode will be negative and sqlca.sqlstate will be different from "00000". A positive sqlca.sqlcode indicates a harmless condition, such as that the last query returned zero rows. sqlcode and sqlstate are two different error code schemes; details appear below.

If the last SQL statement was successful, then sqlca.sqlerrd[1] contains the OID of the processed row, if applicable, and sqlca.sqlerrd[2] contains the number of processed or returned rows, if applicable to the command.

In case of an error or warning, sqlca.sqlerrm.sqlerrmc will contain a string that describes the error. The field sqlca.sqlerrm.sqlerrml contains the length of the error message that is stored in sqlca.sqlerrm.sqlerrmc (the result of strlen()), not really interesting for a C programmer). Note that some messages are too long to fit in the fixed-size sqlerrmc array; they will be truncated.

In case of a warning, sqlca.sqlwarn[2] is set to W. (In all other cases, it is set to something different from W.) If sqlca.sqlwarn[1] is set to W, then a value was truncated when it was stored in a host variable. sqlca.sqlwarn[0] is set to W if any of the other elements are set to indicate a warning.

The fields sqlcaid, sqlcabc, sqlerrp, and the remaining elements of sqlerrd and sqlwarn currently contain no useful information.

The structure sqlca is not defined in the SQL standard, but is implemented in several other SQL database systems. The definitions are similar at the core, but if you want to write portable applications, then you should investigate the different implementations carefully.

32.11.3. SQLSTATE vs SQLCODE

The fields sqlca.sqlstate and sqlca.sqlcode are two different schemes that provide error codes. Both are derived from the SQL standard, but SQLCODE has been marked deprecated in the
SQL-92 edition of the standard and has been dropped in later editions. Therefore, new applications are strongly encouraged to use SQLSTATE.

SQLSTATE is a five-character array. The five characters contain digits or upper-case letters that represent codes of various error and warning conditions. SQLSTATE has a hierarchical scheme: the first two characters indicate the general class of the condition, the last three characters indicate a subclass of the general condition. A successful state is indicated by the code 00000. The SQLSTATE codes are for the most part defined in the SQL standard. The PostgreSQL server natively supports SQLSTATE error codes; therefore a high degree of consistency can be achieved by using this error code scheme throughout all applications. For further information see Appendix A.

SQLCODE, the deprecated error code scheme, is a simple integer. A value of 0 indicates success, a positive value indicates success with additional information, a negative value indicates an error. The SQL standard only defines the positive value +100, which indicates that the last command returned or affected zero rows, and no specific negative values. Therefore, this scheme can only achieve poor portability and does not have a hierarchical code assignment. Historically, the embedded SQL processor for PostgreSQL has assigned some specific SQLCODE values for its use, which are listed below with their numeric value and their symbolic name. Remember that these are not portable to other SQL implementations. To simplify the porting of applications to the SQLSTATE scheme, the corresponding SQLSTATE is also listed. There is, however, no one-to-one or one-to-many mapping between the two schemes (indeed it is many-to-many), so you should consult the global SQLSTATE listing in Appendix A in each case.

These are the assigned SQLCODE values:

-12 (ECPG_OUT_OF_MEMORY)
   Indicates that your virtual memory is exhausted. (SQLSTATE YE001)

-200 (ECPG_UNSUPPORTED)
   Indicates the preprocessor has generated something that the library does not know about. Perhaps you are running incompatible versions of the preprocessor and the library. (SQLSTATE YE002)

-201 (ECPG_TOO_MANY_ARGUMENTS)
   This means that the command specified more host variables than the command expected. (SQLSTATE 07001 or 07002)

-202 (ECPG_TOO_FEW_ARGUMENTS)
   This means that the command specified fewer host variables than the command expected. (SQLSTATE 07001 or 07002)

-203 (ECPG_TOO_MANY_MATCHES)
   This means a query has returned multiple rows but the statement was only prepared to store one result row (for example, because the specified variables are not arrays). (SQLSTATE 21000)

-204 (ECPG_INT_FORMAT)
   The host variable is of type int and the datum in the database is of a different type and contains a value that cannot be interpreted as an int. The library uses strtol() for this conversion. (SQLSTATE 42804)

-205 (ECPG_UINT_FORMAT)
   The host variable is of type unsigned int and the datum in the database is of a different type and contains a value that cannot be interpreted as an unsigned int. The library uses strtoul() for this conversion. (SQLSTATE 42804)
The host variable is of type `float` and the datum in the database is of another type and contains a value that cannot be interpreted as a `float`. The library uses `strtod()` for this conversion. (SQLSTATE 42804)

This means the host variable is of type `bool` and the datum in the database is neither `'t'` nor `'f'`. (SQLSTATE 42804)

The statement sent to the PostgreSQL server was empty. (This cannot normally happen in an embedded SQL program, so it might point to an internal error.) (SQLSTATE YE002)

A null value was returned and no null indicator variable was supplied. (SQLSTATE 22002)

An ordinary variable was used in a place that requires an array. (SQLSTATE 42804)

The database returned an ordinary variable in a place that requires array value. (SQLSTATE 42804)

The program tried to access a connection that does not exist. (SQLSTATE 08003)

The program tried to access a connection that does exist but is not open. (This is an internal error.) (SQLSTATE YE002)

The statement you are trying to use has not been prepared. (SQLSTATE 26000)

The descriptor specified was not found. The statement you are trying to use has not been prepared. (SQLSTATE 33000)

The descriptor index specified was out of range. (SQLSTATE 07009)

An invalid descriptor item was requested. (This is an internal error.) (SQLSTATE YE002)

During the execution of a dynamic statement, the database returned a numeric value and the host variable was not numeric. (SQLSTATE 07006)

During the execution of a dynamic statement, the database returned a non-numeric value and the host variable was numeric. (SQLSTATE 07006)

Some error caused by the PostgreSQL server. The message contains the error message from the PostgreSQL server.
The PostgreSQL server signaled that we cannot start, commit, or rollback the transaction. (SQLSTATE 08007)

The connection attempt to the database did not succeed. (SQLSTATE 08001)

This is a harmless condition indicating that the last command retrieved or processed zero rows, or that you are at the end of the cursor. (SQLSTATE 02000)

### 32.12. Preprocessor directives

#### 32.12.1. Including files

To include an external file into your embedded SQL program, use:

```sql
EXEC SQL INCLUDE filename;
```

The embedded SQL preprocessor will look for a file named `filename.h`, preprocess it, and include it in the resulting C output. Thus, embedded SQL statements in the included file are handled correctly.

Note that this is **not** the same as:

```c
#include <filename.h>
```

because this file would not be subject to SQL command preprocessing. Naturally, you can continue to use the C `#include` directive to include other header files.

**Note:** The include file name is case-sensitive, even though the rest of the `EXEC SQL INCLUDE` command follows the normal SQL case-sensitivity rules.

#### 32.12.2. The `#define` and `#undef` directives

Similar to the directive `#define` that is known from C, embedded SQL has a similar concept:

```sql
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

So you can define a name:

```sql
EXEC SQL DEFINE HAVE_FEATURE;
```

And you can also define constants:

```sql
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```
Use `undef` to remove a previous definition:

```c
EXEC SQL UNDEF MYNUMBER;
```

Of course you can continue to use the C versions `#define` and `#undef` in your embedded SQL program. The difference is where your defined values get evaluated. If you use `EXEC SQL DEFINE` then the ecpg preprocessor evaluates the defines and substitutes the values. For example if you write:

```c
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

then ecpg will already do the substitution and your C compiler will never see any name or identifier `MYNUMBER`. Note that you cannot use `#define` for a constant that you are going to use in an embedded SQL query because in this case the embedded SQL precompiler is not able to see this declaration.

### 32.12.3. ifdef, ifndef, else, elif and endif directives

You can use the following directives to compile code sections conditionally:

```c
EXEC SQL ifdef name;

Checks a `name` and processes subsequent lines if `name` has been created with `EXEC SQL define name`.

EXEC SQL ifndef name;

Checks a `name` and processes subsequent lines if `name` has not been created with `EXEC SQL define name`.

EXEC SQL else;

Starts processing an alternative section to a section introduced by either `EXEC SQL ifdef name` or `EXEC SQL ifndef name`.

EXEC SQL elif name;

Checks `name` and starts an alternative section if `name` has been created with `EXEC SQL define name`.

EXEC SQL endif;

Ends an alternative section.

Example:

```c
exec sql ifdef TZVAR;
exec sql SET TIMEZONE TO 'GMT';
exec sql elif TZNAME;
exec sql SET TIMEZONE TO TZNAME;
exec sql else;
exec sql SET TIMEZONE TO TZVAR;
exec sql endif;
```
32.13. Processing Embedded SQL Programs

Now that you have an idea how to form embedded SQL C programs, you probably want to know how to compile them. Before compiling you run the file through the embedded SQL C preprocessor, which converts the SQL statements you used to special function calls. After compiling, you must link with a special library that contains the needed functions. These functions fetch information from the arguments, perform the SQL command using the libpq interface, and put the result in the arguments specified for output.

The preprocessor program is called ecpg and is included in a normal PostgreSQL installation. Embedded SQL programs are typically named with an extension .pgc. If you have a program file called progl.pgc, you can preprocess it by simply calling:

ecpg progl.pgc

This will create a file called progl.c. If your input files do not follow the suggested naming pattern, you can specify the output file explicitly using the -o option.

The preprocessed file can be compiled normally, for example:

cc -c progl.c

The generated C source files include header files from the PostgreSQL installation, so if you installed PostgreSQL in a location that is not searched by default, you have to add an option such as -I/usr/local/pgsql/include to the compilation command line.

To link an embedded SQL program, you need to include the libecpg library, like so:

cc -o myprog prog1.o prog2.o ... -lepg

Again, you might have to add an option like -L/usr/local/pgsql/lib to that command line.

If you manage the build process of a larger project using make, it might be convenient to include the following implicit rule to your makefiles:

ECPG = ecpg
%.c: %.pgc
  $(ECPG) $<

The complete syntax of the ecpg command is detailed in ecpg.

The ecpg library is thread-safe if it is built using the --enable-thread-safety command-line option to configure. (You might need to use other threading command-line options to compile your client code.)

32.14. Library Functions

The libecpg library primarily contains “hidden” functions that are used to implement the functionality expressed by the embedded SQL commands. But there are some functions that can usefully be called directly. Note that this makes your code unportable.

- ECPGdebug(int on, FILE *stream) turns on debug logging if called with the first argument non-zero. Debug logging is done on stream. The log contains all SQL statements with all the
input variables inserted, and the results from the PostgreSQL server. This can be very useful when searching for errors in your SQL statements.

**Note:** On Windows, if the ecpg libraries and an application are compiled with different flags, this function call will crash the application because the internal representation of the `FILE` pointers differ. Specifically, multithreaded/single-threaded, release/debug, and static/dynamic flags should be the same for the library and all applications using that library.

- ECPGstatus(int lineno, const char* connection_name) returns true if you are connected to a database and false if not. `connection_name` can be NULL if a single connection is being used.

# 32.15. Internals

This section explains how ECPG works internally. This information can occasionally be useful to help users understand how to use ECPG.

The first four lines written by ecpg to the output are fixed lines. Two are comments and two are include lines necessary to interface to the library. Then the preprocessor reads through the file and writes output. Normally it just echoes everything to the output.

When it sees an `EXEC SQL` statement, it intervenes and changes it. The command starts with `EXEC` and ends with `;`. Everything in between is treated as an SQL statement and parsed for variable substitution.

Variable substitution occurs when a symbol starts with a colon (:). The variable with that name is looked up among the variables that were previously declared within a `EXEC SQL DECLARE` section.

The most important function in the library is `ECPGdo`, which takes care of executing most commands. It takes a variable number of arguments. This can easily add up to 50 or so arguments, and we hope this will not be a problem on any platform.

The arguments are:

**A line number**

This is the line number of the original line; used in error messages only.

**A string**

This is the SQL command that is to be issued. It is modified by the input variables, i.e., the variables that where not known at compile time but are to be entered in the command. Where the variables should go the string contains `?`.

**Input variables**

Every input variable causes ten arguments to be created. (See below.)

*ECPGt_EOIT*

An enum telling that there are no more input variables.

**Output variables**

Every output variable causes ten arguments to be created. (See below.) These variables are filled by the function.
ECPGt_EORT

An enum telling that there are no more variables.

For every variable that is part of the SQL command, the function gets ten arguments:

1. The type as a special symbol.
2. A pointer to the value or a pointer to the pointer.
3. The size of the variable if it is a char or varchar.
4. The number of elements in the array (for array fetches).
5. The offset to the next element in the array (for array fetches).
6. The type of the indicator variable as a special symbol.
7. A pointer to the indicator variable.
8. 0
9. The number of elements in the indicator array (for array fetches).
10. The offset to the next element in the indicator array (for array fetches).

Note that not all SQL commands are treated in this way. For instance, an open cursor statement like:

EXEC SQL OPEN cursor;

is not copied to the output. Instead, the cursor’s DECLARE command is used at the position of the OPEN command because it indeed opens the cursor.

Here is a complete example describing the output of the preprocessor of a file foo.pgc (details might change with each particular version of the preprocessor):

EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;

is translated into:

/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

int index;
int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ? ", ECPGt_int,&(index),1L,1L,sizeof(int),
ECPGt_NO_INDICATOR, NULL, 0L, 0L, 0L, ECPGt_EDIT,
ECPGt_int, &(result), 1L, 1L, sizeof(int),
ECPGt_NO_INDICATOR, NULL, 0L, 0L, 0L, ECPGt_EORT);

#line 147 "foo.pgc"

(The indentation here is added for readability and not something the preprocessor does.)
Chapter 33. The Information Schema

The information schema consists of a set of views that contain information about the objects defined in the current database. The information schema is defined in the SQL standard and can therefore be expected to be portable and remain stable — unlike the system catalogs, which are specific to PostgreSQL and are modelled after implementation concerns. The information schema views do not, however, contain information about PostgreSQL-specific features; to inquire about those you need to query the system catalogs or other PostgreSQL-specific views.

33.1. The Schema

The information schema itself is a schema named information_schema. This schema automatically exists in all databases. The owner of this schema is the initial database user in the cluster, and that user naturally has all the privileges on this schema, including the ability to drop it (but the space savings achieved by that are minuscule).

By default, the information schema is not in the schema search path, so you need to access all objects in it through qualified names. Since the names of some of the objects in the information schema are generic names that might occur in user applications, you should be careful if you want to put the information schema in the path.

33.2. Data Types

The columns of the information schema views use special data types that are defined in the information schema. These are defined as simple domains over ordinary built-in types. You should not use these types for work outside the information schema, but your applications must be prepared for them if they select from the information schema.

These types are:

- cardinal_number
  A nonnegative integer.
- character_data
  A character string (without specific maximum length).
- sql_identifier
  A character string. This type is used for SQL identifiers, the type character_data is used for any other kind of text data.
- time_stamp
  A domain over the type timestamp with time zone

Every column in the information schema has one of these four types.

Boolean (true/false) data is represented in the information schema by a column of type character_data that contains either YES or NO. (The information schema was invented before the type boolean was added to the SQL standard, so this convention is necessary to keep the information schema backward compatible.)
33.3. information_schema_catalog_name

information_schema_catalog_name is a table that always contains one row and one column containing the name of the current database (current catalog, in SQL terminology).

Table 33-1. information_schema_catalog_name Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>catalog_name</td>
<td>sql_identifier</td>
<td>Name of the database that contains this information schema</td>
</tr>
</tbody>
</table>

33.4. administrable_role_authorizations

The view administrable_role_authorizations identifies all roles that the current user has the admin option for.

Table 33-2. administrable_role_authorizations Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>Name of the role to which this role membership was granted (can be the current user, or a different role in case of nested role memberships)</td>
</tr>
<tr>
<td>role_name</td>
<td>sql_identifier</td>
<td>Name of a role</td>
</tr>
<tr>
<td>is_grantable</td>
<td>character_data</td>
<td>Always YES</td>
</tr>
</tbody>
</table>

33.5. applicable_roles

The view applicable_roles identifies all roles whose privileges the current user can use. This means there is some chain of role grants from the current user to the role in question. The current user itself is also an applicable role. The set of applicable roles is generally used for permission checking.

Table 33-3. applicable_roles Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>Name of the role to which this role membership was granted (can be the current user, or a different role in case of nested role memberships)</td>
</tr>
<tr>
<td>role_name</td>
<td>sql_identifier</td>
<td>Name of a role</td>
</tr>
</tbody>
</table>
### 33.6. attributes

The view attributes contains information about the attributes of composite data types defined in the database. (Note that the view does not give information about table columns, which are sometimes called attributes in PostgreSQL contexts.)

**Table 33-4. attributes Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the data type (always the current database)</td>
</tr>
<tr>
<td>udt_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the data type</td>
</tr>
<tr>
<td>udt_name</td>
<td>sql_identifier</td>
<td>Name of the data type</td>
</tr>
<tr>
<td>attribute_name</td>
<td>sql_identifier</td>
<td>Name of the attribute</td>
</tr>
<tr>
<td>ordinal_position</td>
<td>cardinal_number</td>
<td>Ordinal position of the attribute within the data type (count starts at 1)</td>
</tr>
<tr>
<td>attribute_default</td>
<td>character_data</td>
<td>Default expression of the attribute</td>
</tr>
<tr>
<td>is_nullable</td>
<td>character_data</td>
<td>YES if the attribute is possibly nullable, NO if it is known not nullable</td>
</tr>
<tr>
<td>data_type</td>
<td>character_data</td>
<td>Data type of the attribute, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in attribute_udt_name and associated columns).</td>
</tr>
<tr>
<td>character_maximum_length</td>
<td>cardinal_number</td>
<td>If data_type identifies a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>character_octet_length</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies a character type, the maximum possible length in octets (bytes) of a datum (this should not be of concern to PostgreSQL users); null for all other data types.</td>
</tr>
<tr>
<td>numeric_precision</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies a numeric type, this column contains the (declared or implicit) precision of the type for this attribute. The precision indicates the number of significant digits. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column <code>numeric_precision_radix</code>. For all other data types, this column is null.</td>
</tr>
<tr>
<td>numeric_precision_radix</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies a numeric type, this column indicates in which base the values in the columns <code>numeric_precision</code> and <code>numeric_scale</code> are expressed. The value is either 2 or 10. For all other data types, this column is null.</td>
</tr>
<tr>
<td>numeric_scale</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies an exact numeric type, this column contains the (declared or implicit) scale of the type for this attribute. The scale indicates the number of significant digits to the right of the decimal point. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column <code>numeric_precision_radix</code>. For all other data types, this column is null.</td>
</tr>
<tr>
<td>datetime_precision</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies a date, time, or interval type, the declared precision; null for all other data types or if no precision was declared.</td>
</tr>
<tr>
<td>interval_type</td>
<td>character_data</td>
<td>Not yet implemented</td>
</tr>
</tbody>
</table>
### Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval_precision</td>
<td>character_data</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>attribute_udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that the attribute data type is defined in (always the current database)</td>
</tr>
<tr>
<td>attribute_udt_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that the attribute data type is defined in</td>
</tr>
<tr>
<td>attribute_udt_name</td>
<td>sql_identifier</td>
<td>Name of the attribute data type</td>
</tr>
<tr>
<td>scope_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>maximum_cardinality</td>
<td>cardinal_number</td>
<td>Always null, because arrays always have unlimited maximum cardinality in PostgreSQL</td>
</tr>
<tr>
<td>dtd_identifier</td>
<td>sql_identifier</td>
<td>An identifier of the data type descriptor of the column, unique among the data type descriptors pertaining to the table. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)</td>
</tr>
<tr>
<td>is_derived_reference_attribute</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
</tbody>
</table>

See also under Section 33.12, a similarly structured view, for further information on some of the columns.

### 33.7. check_constraint_routine_usage

The view `check_constraint_routine_usage` identifies routines (functions and procedures) that are used by a check constraint. Only those routines are shown that are owned by a currently enabled role.

**Table 33-5. check_constraint_routine_usage Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the constraint (always the current database)</td>
</tr>
</tbody>
</table>
### 33.8. check CONSTRAINTS

The view `check_constraints` contains all check constraints, either defined on a table or on a domain, that are owned by a currently enabled role. (The owner of the table or domain is the owner of the constraint.)

#### Table 33-6. check_constraints Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the constraint (always the current database)</td>
</tr>
<tr>
<td>constraint_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the constraint</td>
</tr>
<tr>
<td>constraint_name</td>
<td>sql_identifier</td>
<td>Name of the constraint</td>
</tr>
<tr>
<td>check_clause</td>
<td>character_data</td>
<td>The check expression of the check constraint</td>
</tr>
</tbody>
</table>

### 33.9. column_DOMAIN_USAGE

The view `column_domain_usage` identifies all columns (of a table or a view) that make use of some domain defined in the current database and owned by a currently enabled role.

#### Table 33-7. column_domain_usage Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the domain (always the current database)</td>
</tr>
<tr>
<td>domain_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the domain</td>
</tr>
<tr>
<td>domain_name</td>
<td>sql_identifier</td>
<td>Name of the domain</td>
</tr>
</tbody>
</table>

586
### 33.10. column_privileges

The view `column_privileges` identifies all privileges granted on columns to a currently enabled role or by a currently enabled role. There is one row for each combination of column, grantor, and grantee.

In PostgreSQL, you can only grant privileges on entire tables, not individual columns. Therefore, this view contains the same information as `table_privileges`, just represented through one row for each column in each appropriate table, but it only covers privilege types where column granularity is possible: `SELECT`, `INSERT`, `UPDATE`, `REFERENCES`. If you want to make your applications fit for possible future developments, it is generally the right choice to use this view instead of `table_privileges` if one of those privilege types is concerned.

#### Table 33-8. column_privileges Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>sql_identifier</td>
<td>Name of the role that granted the privilege</td>
</tr>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>Name of the role that the privilege was granted to</td>
</tr>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table that contains the column (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table that contains the column</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table that contains the column</td>
</tr>
<tr>
<td>column_name</td>
<td>sql_identifier</td>
<td>Name of the column</td>
</tr>
<tr>
<td>privilege_type</td>
<td>character_data</td>
<td>Type of the privilege: <code>SELECT</code>, <code>INSERT</code>, <code>UPDATE</code>, <code>REFERENCES</code></td>
</tr>
<tr>
<td>is_grantable</td>
<td>character_data</td>
<td>YES if the privilege is grantable, NO if not</td>
</tr>
</tbody>
</table>
### 33.11. `column_udt_usage`

The view `column_udt_usage` identifies all columns that use data types owned by a currently enabled role. Note that in PostgreSQL, built-in data types behave like user-defined types, so they are included here as well. See also Section 33.12 for details.

**Table 33-9. `column_udt_usage` Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that the column data type (the underlying type of the domain, if applicable) is defined in (always the current database)</td>
</tr>
<tr>
<td>udt_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that the column data type (the underlying type of the domain, if applicable) is defined in</td>
</tr>
<tr>
<td>udt_name</td>
<td>sql_identifier</td>
<td>Name of the column data type (the underlying type of the domain, if applicable)</td>
</tr>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the table (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the table</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table</td>
</tr>
<tr>
<td>column_name</td>
<td>sql_identifier</td>
<td>Name of the column</td>
</tr>
</tbody>
</table>

### 33.12. `columns`

The view `columns` contains information about all table columns (or view columns) in the database. System columns (`oid`, etc.) are not included. Only those columns are shown that the current user has access to (by way of being the owner or having some privilege).

**Table 33-10. `columns` Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the table (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the table</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table</td>
</tr>
<tr>
<td>column_name</td>
<td>sql_identifier</td>
<td>Name of the column</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ordinal_position</td>
<td>cardinal_number</td>
<td>Ordinal position of the column within the table (count starts at 1)</td>
</tr>
<tr>
<td>column_default</td>
<td>character_data</td>
<td>Default expression of the column</td>
</tr>
<tr>
<td>is_nullable</td>
<td>character_data</td>
<td>YES if the column is possibly nullable, NO if it is known not nullable. A not-null constraint is one way a column can be known not nullable, but there can be others.</td>
</tr>
<tr>
<td>data_type</td>
<td>character_data</td>
<td>Data type of the column, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in udt_name and associated columns). If the column is based on a domain, this column refers to the type underlying the domain (and the domain is identified in domain_name and associated columns).</td>
</tr>
<tr>
<td>character_maximum_length</td>
<td>cardinal_number</td>
<td>If data_type identifies a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.</td>
</tr>
<tr>
<td>character_octet_length</td>
<td>cardinal_number</td>
<td>If data_type identifies a character type, the maximum possible length in octets (bytes) of a datum (this should not be of concern to PostgreSQL users); null for all other data types.</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>numeric_precision</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies a numeric type, this column contains the (declared or implicit) precision of the type for this column. The precision indicates the number of significant digits. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column <code>numeric_precision_radix</code>. For all other data types, this column is null.</td>
</tr>
<tr>
<td>numeric_precision_radix</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies a numeric type, this column indicates in which base the values in the columns <code>numeric_precision</code> and <code>numeric_scale</code> are expressed. The value is either 2 or 10. For all other data types, this column is null.</td>
</tr>
<tr>
<td>numeric_scale</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies an exact numeric type, this column contains the (declared or implicit) scale of the type for this column. The scale indicates the number of significant digits to the right of the decimal point. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column <code>numeric_precision_radix</code>. For all other data types, this column is null.</td>
</tr>
<tr>
<td>datetime_precision</td>
<td>cardinal_number</td>
<td>If <code>data_type</code> identifies a date, time, or interval type, the declared precision; null for all other data types or if no precision was declared.</td>
</tr>
<tr>
<td>interval_type</td>
<td>character_data</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>interval_precision</td>
<td>character_data</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>character_set_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>character_set_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>character_set_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>domain_catalog</td>
<td>sql_identifier</td>
<td>If the column has a domain type, the name of the database that the domain is defined in (always the current database), else null.</td>
</tr>
<tr>
<td>domain_schema</td>
<td>sql_identifier</td>
<td>If the column has a domain type, the name of the schema that the domain is defined in, else null.</td>
</tr>
<tr>
<td>domain_name</td>
<td>sql_identifier</td>
<td>If the column has a domain type, the name of the domain, else null.</td>
</tr>
<tr>
<td>udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that the column data type (the underlying type of the domain, if applicable) is defined in (always the current database)</td>
</tr>
<tr>
<td>udt_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that the column data type (the underlying type of the domain, if applicable) is defined in</td>
</tr>
<tr>
<td>udt_name</td>
<td>sql_identifier</td>
<td>Name of the column data type (the underlying type of the domain, if applicable)</td>
</tr>
<tr>
<td>scope_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>maximum_cardinality</td>
<td>cardinal_number</td>
<td>Always null, because arrays always have unlimited maximum cardinality in PostgreSQL</td>
</tr>
</tbody>
</table>
## Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtd_identifier</td>
<td>sql_identifier</td>
<td>An identifier of the data type descriptor of the column, unique among the data type descriptors pertaining to the table. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)</td>
</tr>
<tr>
<td>is_self_referencing</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>is_identity</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>identity_generation</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>identity_start</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>identity_increment</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>identity_maximum</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>identity_minimum</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>identity_cycle</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>is_generated</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>generation_expression</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>is_updatable</td>
<td>character_data</td>
<td>YES if the column is updatable, NO if not (Columns in base tables are always updatable, columns in views not necessarily)</td>
</tr>
</tbody>
</table>

Since data types can be defined in a variety of ways in SQL, and PostgreSQL contains additional ways to define data types, their representation in the information schema can be somewhat difficult. The column `data_type` is supposed to identify the underlying built-in type of the column. In PostgreSQL, this means that the type is defined in the system catalog schema `pg_catalog`. This column might be useful if the application can handle the well-known built-in types specially (for example, format the numeric types differently or use the data in the precision columns). The columns `udt_name`, `udt_schema`, and `udt_catalog` always identify the underlying data type of the column, even if the column is based on a domain. (Since PostgreSQL treats built-in types like user-defined types, built-in types appear here as well. This is an extension of the SQL standard.) These columns should be used if an application wants to process data differently according to the type, because in that case it wouldn’t matter if the column is really based on a domain. If the column is based on a domain, the identity
of the domain is stored in the columns domain_name, domain_schema, and domain_catalog. If you want to pair up columns with their associated data types and treat domains as separate types, you could write \texttt{coalesce(domain_name, udt_name)}, etc.

### 33.13. \texttt{constraint_column_usage}

The view \texttt{constraint_column_usage} identifies all columns in the current database that are used by some constraint. Only those columns are shown that are contained in a table owned by a currently enabled role. For a check constraint, this view identifies the columns that are used in the check expression. For a foreign key constraint, this view identifies the columns that the foreign key references. For a unique or primary key constraint, this view identifies the constrained columns.

**Table 33-11. \texttt{constraint_column_usage} Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table that contains the column that is used by some constraint (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table that contains the column that is used by some constraint</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table that contains the column that is used by some constraint</td>
</tr>
<tr>
<td>column_name</td>
<td>sql_identifier</td>
<td>Name of the column that is used by some constraint</td>
</tr>
<tr>
<td>constraint_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the constraint (always the current database)</td>
</tr>
<tr>
<td>constraint_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the constraint</td>
</tr>
<tr>
<td>constraint_name</td>
<td>sql_identifier</td>
<td>Name of the constraint</td>
</tr>
</tbody>
</table>

### 33.14. \texttt{constraint_table_usage}

The view \texttt{constraint_table_usage} identifies all tables in the current database that are used by some constraint and are owned by a currently enabled role. (This is different from the view \texttt{table_constraints}, which identifies all table constraints along with the table they are defined on.) For a foreign key constraint, this view identifies the table that the foreign key references. For a unique or primary key constraint, this view simply identifies the table the constraint belongs to. Check constraints and not-null constraints are not included in this view.

**Table 33-12. \texttt{constraint_table_usage} Columns**
### 33.15. `data_type_privileges`

The view `data_type_privileges` identifies all data type descriptors that the current user has access to, by way of being the owner of the described object or having some privilege for it. A data type descriptor is generated whenever a data type is used in the definition of a table column, a domain, or a function (as parameter or return type) and stores some information about how the data type is used in that instance (for example, the declared maximum length, if applicable). Each data type descriptor is assigned an arbitrary identifier that is unique among the data type descriptor identifiers assigned for one object (table, domain, function). This view is probably not useful for applications, but it is used to define some other views in the information schema.

#### Table 33-13. `data_type_privileges` Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the described object (always the current database)</td>
</tr>
<tr>
<td>object_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the described object</td>
</tr>
<tr>
<td>object_name</td>
<td>sql_identifier</td>
<td>Name of the described object</td>
</tr>
<tr>
<td>object_type</td>
<td>character_data</td>
<td>The type of the described object: one of TABLE (the data type descriptor pertains to a column of that table), DOMAIN (the data type descriptors pertains to that domain), ROUTINE (the data type descriptor pertains to a parameter or the return data type of that function).</td>
</tr>
</tbody>
</table>
### 33.16. domain_constraints

The view `domain_constraints` contains all constraints belonging to domains defined in the current database.

**Table 33-14. domain_constraints Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the constraint (always the current database)</td>
</tr>
<tr>
<td>constraint_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the constraint</td>
</tr>
<tr>
<td>constraint_name</td>
<td>sql_identifier</td>
<td>Name of the constraint</td>
</tr>
<tr>
<td>domain_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the domain (always the current database)</td>
</tr>
<tr>
<td>domain_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the domain</td>
</tr>
<tr>
<td>domain_name</td>
<td>sql_identifier</td>
<td>Name of the domain</td>
</tr>
<tr>
<td>is_deferrable</td>
<td>character_data</td>
<td>YES if the constraint is deferrable, NO if not</td>
</tr>
<tr>
<td>initially_deferred</td>
<td>character_data</td>
<td>YES if the constraint is deferrable and initially deferred, NO if not</td>
</tr>
</tbody>
</table>

### 33.17. domain_udt_usage

The view `domain_udt_usage` identifies all domains that are based on data types owned by a currently enabled role. Note that in PostgreSQL, built-in data types behave like user-defined types, so they are included here as well.

**Table 33-15. domain_udt_usage Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that the domain data type is defined in (always the current database)</td>
</tr>
</tbody>
</table>
Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>udt_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that the domain data type is defined in</td>
</tr>
<tr>
<td>udt_name</td>
<td>sql_identifier</td>
<td>Name of the domain data type</td>
</tr>
<tr>
<td>domain_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the domain (always the current database)</td>
</tr>
<tr>
<td>domain_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the domain</td>
</tr>
<tr>
<td>domain_name</td>
<td>sql_identifier</td>
<td>Name of the domain</td>
</tr>
</tbody>
</table>

33.18. domains

The view `domains` contains all domains defined in the current database.

Table 33-16. domains Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the domain (always the current database)</td>
</tr>
<tr>
<td>domain_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the domain</td>
</tr>
<tr>
<td>domain_name</td>
<td>sql_identifier</td>
<td>Name of the domain</td>
</tr>
<tr>
<td>data_type</td>
<td>character_data</td>
<td>Data type of the domain, if it is a built-in type, or ARRAY if it is some array (in that case, see the view <code>element_types</code>), else USER-DEFINED (in that case, the type is identified in <code>udt_name</code> and associated columns).</td>
</tr>
<tr>
<td>character_maximum_length</td>
<td>cardinal_number</td>
<td>If the domain has a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.</td>
</tr>
<tr>
<td>character_octet_length</td>
<td>cardinal_number</td>
<td>If the domain has a character type, the maximum possible length in octets (bytes) of a datum (this should not be of concern to PostgreSQL users); null for all other data types.</td>
</tr>
<tr>
<td>character_set_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
</tbody>
</table>
## Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>character_set_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>character_set_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>numeric_precision</td>
<td>cardinal_number</td>
<td>If the domain has a numeric type, this column contains the (declared or implicit) precision of the type for this column. The precision indicates the number of significant digits. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.</td>
</tr>
<tr>
<td>numeric_precision_radix</td>
<td>cardinal_number</td>
<td>If the domain has a numeric type, this column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10. For all other data types, this column is null.</td>
</tr>
<tr>
<td>numeric_scale</td>
<td>cardinal_number</td>
<td>If the domain has an exact numeric type, this column contains the (declared or implicit) scale of the type for this column. The scale indicates the number of significant digits to the right of the decimal point. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>datetime_precision</td>
<td>cardinal_number</td>
<td>If the domain has a date, time, or interval type, the declared precision; null for all other data types or if no precision was declared.</td>
</tr>
<tr>
<td>interval_type</td>
<td>character_data</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>interval_precision</td>
<td>character_data</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>domain_default</td>
<td>character_data</td>
<td>Default expression of the domain</td>
</tr>
<tr>
<td>udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that the domain data type is defined in (always the current database)</td>
</tr>
<tr>
<td>udt_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that the domain data type is defined in</td>
</tr>
<tr>
<td>udt_name</td>
<td>sql_identifier</td>
<td>Name of the domain data type</td>
</tr>
<tr>
<td>scope_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>maximum_cardinality</td>
<td>cardinal_number</td>
<td>Always null, because arrays always have unlimited maximum cardinality in PostgreSQL</td>
</tr>
<tr>
<td>dtd_identifier</td>
<td>sql_identifier</td>
<td>An identifier of the data type descriptor of the domain, unique among the data type descriptors pertaining to the domain (which is trivial, because a domain only contains one data type descriptor). This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)</td>
</tr>
</tbody>
</table>

### 33.19. element_types

The view element_types contains the data type descriptors of the elements of arrays. When a table column, domain, function parameter, or function return value is defined to be of an array type, the respective information schema view only contains ARRAY in the column data_type. To obtain information on the element type of the array, you can join the respective view with this view. For example, to show the columns of a table with data types and array element types, if applicable, you
could do:

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
  ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
    = (e.object_catalog, e.object_schema, e.object_name, e.object_type, e.collection_)
WHERE c.table_schema = '...' AND c.table_name = '...
ORDER BY c.ordinal_position;
```

This view only includes objects that the current user has access to, by way of being the owner or having some privilege.

### Table 33-17. `element_types` Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the object that uses the array being described (always the current database)</td>
</tr>
<tr>
<td>object_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the object that uses the array being described</td>
</tr>
<tr>
<td>object_name</td>
<td>sql_identifier</td>
<td>Name of the object that uses the array being described</td>
</tr>
<tr>
<td>object_type</td>
<td>character_data</td>
<td>The type of the object that uses the array being described: one of TABLE (the array is used by a column of that table), DOMAIN (the array is used by that domain), ROUTINE (the array is used by a parameter or the return data type of that function).</td>
</tr>
<tr>
<td>collection_type_identifier</td>
<td>sql_identifier</td>
<td>The identifier of the data type descriptor of the array being described. Use this to join with the <code>dtd_identifier</code> columns of other information schema views.</td>
</tr>
<tr>
<td>data_type</td>
<td>character_data</td>
<td>Data type of the array elements, if it is a built-in type, else USER-DEFINED (in that case, the type is identified in <code>udt_name</code> and associated columns).</td>
</tr>
<tr>
<td>character_maximum_length</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to array element data types in PostgreSQL.</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>character_octet_length</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to array element data types in PostgreSQL</td>
</tr>
<tr>
<td>character_set_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>character_set_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>character_set_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>numeric_precision</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to array element data types in PostgreSQL</td>
</tr>
<tr>
<td>numeric_precision_radix</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to array element data types in PostgreSQL</td>
</tr>
<tr>
<td>numeric_scale</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to array element data types in PostgreSQL</td>
</tr>
<tr>
<td>datetime_precision</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to array element data types in PostgreSQL</td>
</tr>
<tr>
<td>interval_type</td>
<td>character_data</td>
<td>Always null, since this information is not applied to array element data types in PostgreSQL</td>
</tr>
<tr>
<td>interval_precision</td>
<td>character_data</td>
<td>Always null, since this information is not applied to array element data types in PostgreSQL</td>
</tr>
<tr>
<td>domain_default</td>
<td>character_data</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that the data type of the elements is defined in (always the current database)</td>
</tr>
</tbody>
</table>
33.20. enabled_roles

The view enabled_roles identifies the currently “enabled roles”. The enabled roles are recursively defined as the current user together with all roles that have been granted to the enabled roles with automatic inheritance. In other words, these are all roles that the current user has direct or indirect, automatically inheriting membership in.

For permission checking, the set of “applicable roles” is applied, which can be broader than the set of enabled roles. So generally, it is better to use the view applicable_roles instead of this one; see also there.

Table 33-18. enabled_roles Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role_name</td>
<td>sql_identifier</td>
<td>Name of a role</td>
</tr>
</tbody>
</table>

33.21. key_column_usage

The view key_column_usage identifies all columns in the current database that are restricted by some unique, primary key, or foreign key constraint. Check constraints are not included in this view. Only those columns are shown that the current user has access to, by way of being the owner or having some privilege.

Table 33-19. key_column_usage Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the constraint (always the current database)</td>
</tr>
<tr>
<td>constraint_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the constraint</td>
</tr>
<tr>
<td>constraint_name</td>
<td>sql_identifier</td>
<td>Name of the constraint</td>
</tr>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table that contains the column that is restricted by this constraint (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table that contains the column that is restricted by this constraint</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table that contains the column that is restricted by this constraint</td>
</tr>
<tr>
<td>column_name</td>
<td>sql_identifier</td>
<td>Name of the column that is restricted by this constraint</td>
</tr>
<tr>
<td>ordinal_position</td>
<td>cardinal_number</td>
<td>Ordinal position of the column within the constraint key (count starts at 1)</td>
</tr>
<tr>
<td>position_in_unique_constraint</td>
<td>cardinal_number</td>
<td>For a foreign-key constraint, ordinal position of the referenced column within its unique constraint (count starts at 1); otherwise null</td>
</tr>
</tbody>
</table>

#### 33.22. parameters

The view `parameters` contains information about the parameters (arguments) of all functions in the current database. Only those functions are shown that the current user has access to (by way of being the owner or having some privilege).

### Table 33-20. parameters Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>specific_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the function (always the current database)</td>
</tr>
<tr>
<td>specific_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the function</td>
</tr>
<tr>
<td>specific_name</td>
<td>sql_identifier</td>
<td>The “specific name” of the function. See Section 33.29 for more information.</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ordinal_position</td>
<td>cardinal_number</td>
<td>Ordinal position of the parameter in the argument list of the function (count starts at 1)</td>
</tr>
<tr>
<td>parameter_mode</td>
<td>character_data</td>
<td>IN for input parameter, OUT for output parameter, and INOUT for input/output parameter.</td>
</tr>
<tr>
<td>is_result</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>as_locator</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>parameter_name</td>
<td>sql_identifier</td>
<td>Name of the parameter, or null if the parameter has no name.</td>
</tr>
<tr>
<td>data_type</td>
<td>character_data</td>
<td>Data type of the parameter, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in udt_name and associated columns).</td>
</tr>
<tr>
<td>character_maximum_length</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to parameter data types in PostgreSQL.</td>
</tr>
<tr>
<td>character_octet_length</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to parameter data types in PostgreSQL.</td>
</tr>
<tr>
<td>character_set_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>character_set_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>character_set_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>collation_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>collation_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>collation_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>numeric_precision</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to parameter data types in PostgreSQL.</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>numeric_precision_radix</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to parameter data types in PostgreSQL</td>
</tr>
<tr>
<td>numeric_scale</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to parameter data types in PostgreSQL</td>
</tr>
<tr>
<td>datetime_precision</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to parameter data types in PostgreSQL</td>
</tr>
<tr>
<td>interval_type</td>
<td>character_data</td>
<td>Always null, since this information is not applied to parameter data types in PostgreSQL</td>
</tr>
<tr>
<td>interval_precision</td>
<td>character_data</td>
<td>Always null, since this information is not applied to parameter data types in PostgreSQL</td>
</tr>
<tr>
<td>udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that the data type of the parameter is defined in (always the current database)</td>
</tr>
<tr>
<td>udt_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that the data type of the parameter is defined in</td>
</tr>
<tr>
<td>udt_name</td>
<td>sql_identifier</td>
<td>Name of the data type of the parameter</td>
</tr>
<tr>
<td>scope_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>maximum_cardinality</td>
<td>cardinal_number</td>
<td>Always null, because arrays always have unlimited maximum cardinality in PostgreSQL</td>
</tr>
</tbody>
</table>
Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtd_identifier</td>
<td>sql_identifier</td>
<td>An identifier of the data type descriptor of the parameter, unique among the data type descriptors pertaining to the function. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)</td>
</tr>
</tbody>
</table>

### 33.23. referential_constraints

The view referential_constraints contains all referential (foreign key) constraints in the current database that belong to a table owned by a currently enabled role.

#### Table 33-21. referential_constraints Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the constraint (always the current database)</td>
</tr>
<tr>
<td>constraint_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the constraint</td>
</tr>
<tr>
<td>constraint_name</td>
<td>sql_identifier</td>
<td>Name of the constraint</td>
</tr>
<tr>
<td>unique_constraint_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the unique or primary key constraint that the foreign key constraint references (always the current database)</td>
</tr>
<tr>
<td>unique_constraint_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the unique or primary key constraint that the foreign key constraint references</td>
</tr>
<tr>
<td>unique_constraint_name</td>
<td>sql_identifier</td>
<td>Name of the unique or primary key constraint that the foreign key constraint references</td>
</tr>
<tr>
<td>match_option</td>
<td>character_data</td>
<td>Match option of the foreign key constraint: FULL, PARTIAL, or NONE.</td>
</tr>
<tr>
<td>update_rule</td>
<td>character_data</td>
<td>Update rule of the foreign key constraint: CASCADE, SET NULL, SET DEFAULT, RESTRICT, or NO ACTION.</td>
</tr>
</tbody>
</table>
### 33.24. role_column_grants

The view `role_column_grants` identifies all privileges granted on columns where the grantor or grantee is a currently enabled role. Further information can be found under `column_privileges`.

**Table 33-22. role_column_grants Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>sql_identifier</td>
<td>Name of the role that granted the privilege</td>
</tr>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>Name of the role that the privilege was granted to</td>
</tr>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table that contains the column (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table that contains the column</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table that contains the column</td>
</tr>
<tr>
<td>column_name</td>
<td>sql_identifier</td>
<td>Name of the column</td>
</tr>
<tr>
<td>privilege_type</td>
<td>character_data</td>
<td>Type of the privilege: SELECT, INSERT, UPDATE, or REFERENCES</td>
</tr>
<tr>
<td>is_grantable</td>
<td>character_data</td>
<td>YES if the privilege is grantable, NO if not</td>
</tr>
</tbody>
</table>

### 33.25. role_routine_grants

The view `role_routine_grants` identifies all privileges granted on functions where the grantor or grantee is a currently enabled role. Further information can be found under `routine_privileges`.

**Table 33-23. role_routine_grants Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>sql_identifier</td>
<td>Name of the role that granted the privilege</td>
</tr>
</tbody>
</table>
Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>Name of the role that the privilege was granted to</td>
</tr>
<tr>
<td>specific_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the function (always the current database)</td>
</tr>
<tr>
<td>specific_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the function</td>
</tr>
<tr>
<td>specific_name</td>
<td>sql_identifier</td>
<td>The &quot;specific name&quot; of the function. See Section 33.29 for more information.</td>
</tr>
<tr>
<td>routine_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the function (always the current database)</td>
</tr>
<tr>
<td>routine_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the function</td>
</tr>
<tr>
<td>routine_name</td>
<td>sql_identifier</td>
<td>Name of the function (might be duplicated in case of overloading)</td>
</tr>
<tr>
<td>privilege_type</td>
<td>character_data</td>
<td>Always EXECUTE (the only privilege type for functions)</td>
</tr>
<tr>
<td>is_grantable</td>
<td>character_data</td>
<td>YES if the privilege is grantable, NO if not</td>
</tr>
</tbody>
</table>

33.26. role_table_grants

The view role_table_grants identifies all privileges granted on tables or views where the grantor or grantee is a currently enabled role. Further information can be found under table_privileges.

Table 33-24. role_table_grants Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>sql_identifier</td>
<td>Name of the role that granted the privilege</td>
</tr>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>Name of the role that the privilege was granted to</td>
</tr>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table</td>
</tr>
<tr>
<td>privilege_type</td>
<td>character_data</td>
<td>Type of the privilege: SELECT, DELETE, INSERT, UPDATE, REFERENCES, or TRIGGER</td>
</tr>
</tbody>
</table>
### 33.27. role_usage_grants

The view `role_usage_grants` is meant to identify `USAGE` privileges granted on various kinds of objects to a currently enabled role or by a currently enabled role. In PostgreSQL, this currently only applies to domains, and since domains do not have real privileges in PostgreSQL, this view is empty. Further information can be found under `usage_privileges`. In the future, this view might contain more useful information.

**Table 33-25. role_usage_grants Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>sql_identifier</td>
<td>In the future, the name of the role that granted the privilege</td>
</tr>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>In the future, the name of the role that the privilege was granted to</td>
</tr>
<tr>
<td>object_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the object (always the current database)</td>
</tr>
<tr>
<td>object_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the object</td>
</tr>
<tr>
<td>object_name</td>
<td>sql_identifier</td>
<td>Name of the object</td>
</tr>
<tr>
<td>object_type</td>
<td>character_data</td>
<td>In the future, the type of the object</td>
</tr>
<tr>
<td>privilege_type</td>
<td>character_data</td>
<td>Always <code>USAGE</code></td>
</tr>
<tr>
<td>is_grantable</td>
<td>character_data</td>
<td>YES if the privilege is grantable, NO if not</td>
</tr>
</tbody>
</table>

### 33.28. routine_privileges

The view `routine_privileges` identifies all privileges granted to a currently enabled role or by a currently enabled role. There is one row for each combination of function, grantor, and grantee.

**Table 33-26. routine_privileges Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>sql_identifier</td>
<td>Name of the role that granted the privilege</td>
</tr>
</tbody>
</table>
### 33.29. routines

The view `routines` contains all functions in the current database. Only those functions are shown that the current user has access to (by way of being the owner or having some privilege).

#### Table 33-27. routines Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>specific_catalog</td>
<td><code>sql_identifier</code></td>
<td>Name of the database containing the function (always the current database)</td>
</tr>
<tr>
<td>specific_schema</td>
<td><code>sql_identifier</code></td>
<td>Name of the schema containing the function</td>
</tr>
<tr>
<td>specific_name</td>
<td><code>sql_identifier</code></td>
<td>The &quot;specific name&quot; of the function. See Section 33.29 for more information.</td>
</tr>
<tr>
<td>routine_catalog</td>
<td><code>sql_identifier</code></td>
<td>Name of the database containing the function (always the current database)</td>
</tr>
<tr>
<td>routine_schema</td>
<td><code>sql_identifier</code></td>
<td>Name of the schema containing the function</td>
</tr>
<tr>
<td>routine_name</td>
<td><code>sql_identifier</code></td>
<td>Name of the function (might be duplicated in case of overloading)</td>
</tr>
<tr>
<td>privilege_type</td>
<td><code>character_data</code></td>
<td>Always <code>EXECUTE</code> (the only privilege type for functions)</td>
</tr>
<tr>
<td>is_grantable</td>
<td><code>character_data</code></td>
<td>YES if the privilege is grantable, NO if not</td>
</tr>
</tbody>
</table>
## Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>routine_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the function (always the current database)</td>
</tr>
<tr>
<td>routine_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the function</td>
</tr>
<tr>
<td>routine_name</td>
<td>sql_identifier</td>
<td>Name of the function (might be duplicated in case of overloading)</td>
</tr>
<tr>
<td>routine_type</td>
<td>character_data</td>
<td>Always <strong>FUNCTION</strong> (In the future there might be other types of routines.)</td>
</tr>
<tr>
<td>module_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>module_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>module_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>udt_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>udt_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>udt_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>data_type</td>
<td>character_data</td>
<td>Return data type of the function, if it is a built-in type, or <strong>ARRAY</strong> if it is some array (in that case, see the view <code>element_types</code>), else <strong>USER-DEFINED</strong> (in that case, the type is identified in <code>type_udt_name</code> and associated columns).</td>
</tr>
<tr>
<td>character_maximum_length</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to return data types in PostgreSQL</td>
</tr>
<tr>
<td>character_octet_length</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to return data types in PostgreSQL</td>
</tr>
<tr>
<td>character_set_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>character_set_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>character_set_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
</tbody>
</table>
## Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collation_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>collation_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>numeric_precision</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to return data types in PostgreSQL</td>
</tr>
<tr>
<td>numeric_precision_radix</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to return data types in PostgreSQL</td>
</tr>
<tr>
<td>numeric_scale</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to return data types in PostgreSQL</td>
</tr>
<tr>
<td>datetime_precision</td>
<td>cardinal_number</td>
<td>Always null, since this information is not applied to return data types in PostgreSQL</td>
</tr>
<tr>
<td>interval_type</td>
<td>character_data</td>
<td>Always null, since this information is not applied to return data types in PostgreSQL</td>
</tr>
<tr>
<td>interval_precision</td>
<td>character_data</td>
<td>Always null, since this information is not applied to return data types in PostgreSQL</td>
</tr>
<tr>
<td>type_udt_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that the return data type of the function is defined in (always the current database)</td>
</tr>
<tr>
<td>type_udt_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that the return data type of the function is defined in</td>
</tr>
<tr>
<td>type_udt_name</td>
<td>sql_identifier</td>
<td>Name of the return data type of the function</td>
</tr>
<tr>
<td>scope_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>scope_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>maximum_cardinality</td>
<td>cardinal_number</td>
<td>Always null, because arrays always have unlimited maximum cardinality in PostgreSQL.</td>
</tr>
<tr>
<td>dtd_identifier</td>
<td>sql_identifier</td>
<td>An identifier of the data type descriptor of the return data type of this function, unique among the data type descriptors pertaining to the function. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)</td>
</tr>
<tr>
<td>routine_body</td>
<td>character_data</td>
<td>If the function is an SQL function, then SQL, else EXTERNAL.</td>
</tr>
<tr>
<td>routine_definition</td>
<td>character_data</td>
<td>The source text of the function (null if the function is not owned by a currently enabled role). (According to the SQL standard, this column is only applicable if routine_body is SQL, but in PostgreSQL it will contain whatever source text was specified when the function was created.)</td>
</tr>
<tr>
<td>external_name</td>
<td>character_data</td>
<td>If this function is a C function, then the external name (link symbol) of the function; else null. (This works out to be the same value that is shown in routine_definition.)</td>
</tr>
<tr>
<td>external_language</td>
<td>character_data</td>
<td>The language the function is written in.</td>
</tr>
<tr>
<td>parameter_style</td>
<td>character_data</td>
<td>Always GENERAL (The SQL standard defines other parameter styles, which are not available in PostgreSQL.)</td>
</tr>
<tr>
<td>is_deterministic</td>
<td>character_data</td>
<td>If the function is declared immutable (called deterministic in the SQL standard), then YES, else NO. (You cannot query the other volatility levels available in PostgreSQL through the information schema.)</td>
</tr>
</tbody>
</table>
### Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql_data_access</td>
<td>character_data</td>
<td>Always <strong>MODIFIES</strong>, meaning that the function possibly modifies SQL data. This information is not useful for PostgreSQL.</td>
</tr>
<tr>
<td>is_null_call</td>
<td>character_data</td>
<td>If the function automatically returns null if any of its arguments are null, then <strong>YES</strong>, else <strong>NO</strong>.</td>
</tr>
<tr>
<td>sql_path</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>schema_level_routine</td>
<td>character_data</td>
<td>Always <strong>YES</strong> (The opposite would be a method of a user-defined type, which is a feature not available in PostgreSQL.)</td>
</tr>
<tr>
<td>max_dynamic_result_sets</td>
<td>cardinal_number</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>is_user_defined_cast</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>is_implicitly_invocable</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>security_type</td>
<td>character_data</td>
<td>If the function runs with the privileges of the current user, then <strong>INVOKER</strong>, if the function runs with the privileges of the user who defined it, then <strong>DEFINER</strong>.</td>
</tr>
<tr>
<td>to_sql_specific_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>to_sql_specific_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>to_sql_specific_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>as_locator</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>created</td>
<td>time_stamp</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>last_altered</td>
<td>time_stamp</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>new_savepoint_level</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>is_udt_dependent</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>result_cast_from_data_type</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL.</td>
</tr>
<tr>
<td>Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>result_cast_as_locator</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_char_max_length</td>
<td>cardinal_number</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_char_octet_length</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_char_set_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_char_set_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_char_set_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_collation_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_collation_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_collation_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_numeric_precision</td>
<td>cardinal_number</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_numeric_precision_radix</td>
<td>cardinal_number</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_numeric_scale</td>
<td>cardinal_number</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_datetime_precision</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_interval_type</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_interval_precision</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_type_udt_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_type_udt_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_type_udt_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_scope_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_scope_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_scope_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_maximum_cardinality</td>
<td>cardinal_number</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>result_cast_dtd_identifier</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
</tbody>
</table>
33.30. schemata

The view `schemata` contains all schemas in the current database that are owned by a currently enabled role.

Table 33-28. schemata Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>catalog_name</td>
<td>sql_identifier</td>
<td>Name of the database that the schema is contained in (always the current database)</td>
</tr>
<tr>
<td>schema_name</td>
<td>sql_identifier</td>
<td>Name of the schema</td>
</tr>
<tr>
<td>schema_owner</td>
<td>sql_identifier</td>
<td>Name of the owner of the schema</td>
</tr>
<tr>
<td>default_character_set_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>default_character_set_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>default_character_set_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>sql_path</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
</tbody>
</table>

33.31. sequences

The view `sequences` contains all sequences defined in the current database. Only those sequences are shown that the current user has access to (by way of being the owner or having some privilege).

Table 33-29. sequences Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the sequence (always the current database)</td>
</tr>
<tr>
<td>sequence_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the sequence</td>
</tr>
<tr>
<td>sequence_name</td>
<td>sql_identifier</td>
<td>Name of the sequence</td>
</tr>
<tr>
<td>data_type</td>
<td>character_data</td>
<td>The data type of the sequence. In PostgreSQL, this is currently always bigint.</td>
</tr>
</tbody>
</table>
Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric_precision</td>
<td>cardinal_number</td>
<td>This column contains the (declared or implicit) precision of the sequence data type (see above). The precision indicates the number of significant digits. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix.</td>
</tr>
<tr>
<td>numeric_precision_radix</td>
<td>cardinal_number</td>
<td>This column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10.</td>
</tr>
<tr>
<td>numeric_scale</td>
<td>cardinal_number</td>
<td>This column contains the (declared or implicit) scale of the sequence data type (see above). The scale indicates the number of significant digits to the right of the decimal point. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix.</td>
</tr>
<tr>
<td>maximum_value</td>
<td>cardinal_number</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>minimum_value</td>
<td>cardinal_number</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>increment</td>
<td>cardinal_number</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>cycle_option</td>
<td>character_data</td>
<td>Not yet implemented</td>
</tr>
</tbody>
</table>

33.32. sql_features

The table sql_features contains information about which formal features defined in the SQL standard are supported by PostgreSQL. This is the same information that is presented in Appendix D. There you can also find some additional background information.

Table 33-30. sql_features Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_id</td>
<td>character_data</td>
<td>Identifier string of the feature</td>
</tr>
<tr>
<td>feature_name</td>
<td>character_data</td>
<td>Descriptive name of the feature</td>
</tr>
</tbody>
</table>
33.33. sql_implementation_info

The table `sql_implementation_info` contains information about various aspects that are left implementation-defined by the SQL standard. This information is primarily intended for use in the context of the ODBC interface; users of other interfaces will probably find this information to be of little use. For this reason, the individual implementation information items are not described here; you will find them in the description of the ODBC interface.

Table 33-31. sql_implementation_info Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>implementation_info_id</td>
<td>character_data</td>
<td>Identifier string of the implementation information item</td>
</tr>
<tr>
<td>implementation_info_name</td>
<td>character_data</td>
<td>Descriptive name of the implementation information item</td>
</tr>
<tr>
<td>integer_value</td>
<td>cardinal_number</td>
<td>Value of the implementation information item, or null if the value is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>contained in the column <code>character_value</code></td>
</tr>
<tr>
<td>character_value</td>
<td>character_data</td>
<td>Value of the implementation information item, or null if the value is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>contained in the column <code>integer_value</code></td>
</tr>
<tr>
<td>comments</td>
<td>character_data</td>
<td>Possibly a comment pertaining to the implementation information item</td>
</tr>
</tbody>
</table>
33.34. sql_languages

The table sql_languages contains one row for each SQL language binding that is supported by PostgreSQL. PostgreSQL supports direct SQL and embedded SQL in C; that is all you will learn from this table.

Table 33-32. sql_languages Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql_language_source</td>
<td>character_data</td>
<td>The name of the source of the language definition; always ISO 9075, that is, the SQL standard</td>
</tr>
<tr>
<td>sql_language_year</td>
<td>character_data</td>
<td>The year the standard referenced in sql_language_source was approved; currently 2003</td>
</tr>
<tr>
<td>sql_language_conformance</td>
<td>character_data</td>
<td>The standard conformance level for the language binding. For ISO 9075:2003 this is always CORE.</td>
</tr>
<tr>
<td>sql_language_integrity</td>
<td>character_data</td>
<td>Always null (This value is relevant to an earlier version of the SQL standard.)</td>
</tr>
<tr>
<td>sql_language_implementation</td>
<td>character_data</td>
<td>Always null</td>
</tr>
<tr>
<td>sql_language_binding_style</td>
<td>character_data</td>
<td>The language binding style, either DIRECT or EMBEDDED</td>
</tr>
<tr>
<td>sql_language_programming</td>
<td>character_data</td>
<td>The programming language, if the binding style is EMBEDDED, else null. PostgreSQL only supports the language C.</td>
</tr>
</tbody>
</table>

33.35. sql_packages

The table sql_packages contains information about which feature packages defined in the SQL standard are supported by PostgreSQL. Refer to Appendix D for background information on feature packages.

Table 33-33. sql_packages Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_id</td>
<td>character_data</td>
<td>Identifier string of the package</td>
</tr>
<tr>
<td>feature_name</td>
<td>character_data</td>
<td>Descriptive name of the package</td>
</tr>
<tr>
<td>is_supported</td>
<td>character_data</td>
<td>YES if the package is fully supported by the current version of PostgreSQL, NO if not</td>
</tr>
</tbody>
</table>
Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_verified_by</td>
<td>character_data</td>
<td>Always null, since the PostgreSQL development group does not perform formal testing of feature conformance</td>
</tr>
<tr>
<td>comments</td>
<td>character_data</td>
<td>Possibly a comment about the supported status of the package</td>
</tr>
</tbody>
</table>

### 33.36. sql_parts

The table `sql_parts` contains information about which of the several parts of the SQL standard are supported by PostgreSQL.

**Table 33-34. sql_parts Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature_id</td>
<td>character_data</td>
<td>An identifier string containing the number of the part</td>
</tr>
<tr>
<td>feature_name</td>
<td>character_data</td>
<td>Descriptive name of the part</td>
</tr>
<tr>
<td>is_supported</td>
<td>character_data</td>
<td>YES if the part is fully supported by the current version of PostgreSQL, NO if not</td>
</tr>
<tr>
<td>is_verified_by</td>
<td>character_data</td>
<td>Always null, since the PostgreSQL development group does not perform formal testing of feature conformance</td>
</tr>
<tr>
<td>comments</td>
<td>character_data</td>
<td>Possibly a comment about the supported status of the part</td>
</tr>
</tbody>
</table>

### 33.37. sql_sizing

The table `sql_sizing` contains information about various size limits and maximum values in PostgreSQL. This information is primarily intended for use in the context of the ODBC interface; users of other interfaces will probably find this information to be of little use. For this reason, the individual sizing items are not described here; you will find them in the description of the ODBC interface.

**Table 33-35. sql_sizing Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizing_id</td>
<td>cardinal_number</td>
<td>Identifier of the sizing item</td>
</tr>
<tr>
<td>sizing_name</td>
<td>character_data</td>
<td>Descriptive name of the sizing item</td>
</tr>
</tbody>
</table>
Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>supported_value</td>
<td>cardinal_number</td>
<td>Value of the sizing item, or 0 if the size is unlimited or cannot be determined, or null if the features for which the sizing item is applicable are not supported</td>
</tr>
<tr>
<td>comments</td>
<td>character_data</td>
<td>Possibly a comment pertaining to the sizing item</td>
</tr>
</tbody>
</table>

### 33.38. sql_sizing_profiles

The table `sql_sizing_profiles` contains information about the `sql_sizing` values that are required by various profiles of the SQL standard. PostgreSQL does not track any SQL profiles, so this table is empty.

**Table 33-36. sql_sizing_profiles Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizing_id</td>
<td>cardinal_number</td>
<td>Identifier of the sizing item</td>
</tr>
<tr>
<td>sizing_name</td>
<td>character_data</td>
<td>Descriptive name of the sizing item</td>
</tr>
<tr>
<td>profile_id</td>
<td>character_data</td>
<td>Identifier string of a profile</td>
</tr>
<tr>
<td>required_value</td>
<td>cardinal_number</td>
<td>The value required by the SQL profile for the sizing item, or 0 if the profile places no limit on the sizing item, or null if the profile does not require any of the features for which the sizing item is applicable</td>
</tr>
<tr>
<td>comments</td>
<td>character_data</td>
<td>Possibly a comment pertaining to the sizing item within the profile</td>
</tr>
</tbody>
</table>

### 33.39. table_constraints

The view `table_constraints` contains all constraints belonging to tables that the current user owns or has some privilege on.

**Table 33-37. table_constraints Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the constraint (always the current database)</td>
</tr>
</tbody>
</table>
### Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the constraint</td>
</tr>
<tr>
<td>constraint_name</td>
<td>sql_identifier</td>
<td>Name of the constraint</td>
</tr>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table</td>
</tr>
<tr>
<td>constraint_type</td>
<td>character_data</td>
<td>Type of the constraint: CHECK, FOREIGN KEY, PRIMARY KEY, or UNIQUE</td>
</tr>
<tr>
<td>is_deferrable</td>
<td>character_data</td>
<td>YES if the constraint is deferrable, NO if not</td>
</tr>
<tr>
<td>initially_deferred</td>
<td>character_data</td>
<td>YES if the constraint is deferrable and initially deferred, NO if not</td>
</tr>
</tbody>
</table>

### 33.40. table_privileges

The view *table_privileges* identifies all privileges granted on tables or views to a currently enabled role or by a currently enabled role. There is one row for each combination of table, grantor, and grantee.

#### Table 33-38. table_privileges Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>sql_identifier</td>
<td>Name of the role that granted the privilege</td>
</tr>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>Name of the role that the privilege was granted to</td>
</tr>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table</td>
</tr>
<tr>
<td>privilege_type</td>
<td>character_data</td>
<td>Type of the privilege: SELECT, DELETE, INSERT, UPDATE, REFERENCES, or TRIGGER</td>
</tr>
<tr>
<td>is_grantable</td>
<td>character_data</td>
<td>YES if the privilege is grantable, NO if not</td>
</tr>
<tr>
<td>with_hierarchy</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
</tbody>
</table>
### 33.41. tables

The view tables contains all tables and views defined in the current database. Only those tables and views are shown that the current user has access to (by way of being the owner or having some privilege).

**Table 33-39. tables Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table</td>
</tr>
<tr>
<td>table_type</td>
<td>character_data</td>
<td>Type of the table: BASE TABLE for a persistent base table (the normal table type), VIEW for a view, or LOCAL TEMPORARY for a temporary table</td>
</tr>
<tr>
<td>self_referencing_column_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>reference_generation</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>user_defined_type_catalog</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>user_defined_type_schema</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>user_defined_type_name</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>is_insertable_into</td>
<td>character_data</td>
<td>YES if the table is insertable into, NO if not (Base tables are always insertable into, views not necessarily.)</td>
</tr>
<tr>
<td>is_typed</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>commit_action</td>
<td>character_data</td>
<td>If the table is a temporary table, then PRESERVE, else null. (The SQL standard defines other commit actions for temporary tables, which are not supported by PostgreSQL.)</td>
</tr>
</tbody>
</table>

### 33.42. triggers

The view triggers contains all triggers defined in the current database on tables that the current user owns or has some privilege on.
<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigger_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the trigger (always the current database)</td>
</tr>
<tr>
<td>trigger_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the trigger</td>
</tr>
<tr>
<td>trigger_name</td>
<td>sql_identifier</td>
<td>Name of the trigger</td>
</tr>
<tr>
<td>event_manipulation</td>
<td>character_data</td>
<td>Event that fires the trigger (INSERT, UPDATE, or DELETE)</td>
</tr>
<tr>
<td>event_object_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table that the trigger is defined on (always the current database)</td>
</tr>
<tr>
<td>event_object_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table that the trigger is defined on</td>
</tr>
<tr>
<td>event_object_table</td>
<td>sql_identifier</td>
<td>Name of the table that the trigger is defined on</td>
</tr>
<tr>
<td>action_order</td>
<td>cardinal_number</td>
<td>Not yet implemented</td>
</tr>
<tr>
<td>action_condition</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>action_statement</td>
<td>character_data</td>
<td>Statement that is executed by the trigger (currently always EXECUTE PROCEDURE function(...))</td>
</tr>
<tr>
<td>action_orientation</td>
<td>character_data</td>
<td>Identifies whether the trigger fires once for each processed row or once for each statement (ROW or STATEMENT)</td>
</tr>
<tr>
<td>condition_timing</td>
<td>character_data</td>
<td>Time at which the trigger fires (BEFORE or AFTER)</td>
</tr>
<tr>
<td>condition_reference_old</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>condition_reference_new</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>condition_reference_old</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>condition_reference_new</td>
<td>sql_identifier</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>created</td>
<td>time_stamp</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
</tbody>
</table>

Triggers in PostgreSQL have two incompatibilities with the SQL standard that affect the representation in the information schema. First, trigger names are local to the table in PostgreSQL, rather than being independent schema objects. Therefore there can be duplicate trigger names defined in one schema, as long as they belong to different tables. (trigger_catalog and...
trigger_schema are really the values pertaining to the table that the trigger is defined on.) Second, triggers can be defined to fire on multiple events in PostgreSQL (e.g., ON INSERT OR UPDATE), whereas the SQL standard only allows one. If a trigger is defined to fire on multiple events, it is represented as multiple rows in the information schema, one for each type of event. As a consequence of these two issues, the primary key of the view triggers is really (trigger_catalog, trigger_schema, trigger_name, event_object_table, event_manipulation) instead of (trigger_catalog, trigger_schema, trigger_name), which is what the SQL standard specifies. Nonetheless, if you define your triggers in a manner that conforms with the SQL standard (trigger names unique in the schema and only one event type per trigger), this will not affect you.

### 33.43. usage_privileges

The view usage_privileges is meant to identify USAGE privileges granted on various kinds of objects to a currently enabled role or by a currently enabled role. In PostgreSQL, this currently only applies to domains, and since domains do not have real privileges in PostgreSQL, this view shows implicit USAGE privileges granted to PUBLIC for all domains. In the future, this view might contain more useful information.

**Table 33-41. usage_privileges Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>sql_identifier</td>
<td>Currently set to the name of the owner of the object</td>
</tr>
<tr>
<td>grantee</td>
<td>sql_identifier</td>
<td>Currently always PUBLIC</td>
</tr>
<tr>
<td>object_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the object (always the current database)</td>
</tr>
<tr>
<td>object_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the object</td>
</tr>
<tr>
<td>object_name</td>
<td>sql_identifier</td>
<td>Name of the object</td>
</tr>
<tr>
<td>object_type</td>
<td>character_data</td>
<td>Currently always DOMAIN</td>
</tr>
<tr>
<td>privilege_type</td>
<td>character_data</td>
<td>Always USAGE</td>
</tr>
<tr>
<td>is_grantable</td>
<td>character_data</td>
<td>Currently always NO</td>
</tr>
</tbody>
</table>

### 33.44. view_column_usage

The view view_column_usage identifies all columns that are used in the query expression of a view (the SELECT statement that defines the view). A column is only included if the table that contains the column is owned by a currently enabled role.

**Note:** Columns of system tables are not included. This should be fixed sometime.

**Table 33-42. view_column_usage Columns**
### 33.45. \texttt{view\_routine\_usage}

The \texttt{view\_routine\_usage} identifies all routines (functions and procedures) that are used in the query expression of a view (the \texttt{SELECT} statement that defines the view). A routine is only included if that routine is owned by a currently enabled role.

#### Table 33-43. \texttt{view\_routine\_usage} Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the view (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the view</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the view</td>
</tr>
<tr>
<td>specific_catalog</td>
<td>sql_identifier</td>
<td>Name of the database containing the function (always the current database)</td>
</tr>
<tr>
<td>specific_schema</td>
<td>sql_identifier</td>
<td>Name of the schema containing the function</td>
</tr>
<tr>
<td>specific_name</td>
<td>sql_identifier</td>
<td>The “specific name” of the function. See Section 33.29 for more information.</td>
</tr>
</tbody>
</table>
33.46. **view_table_usage**

The view **view_table_usage** identifies all tables that are used in the query expression of a view (the `SELECT` statement that defines the view). A table is only included if that table is owned by a currently enabled role.

**Note:** System tables are not included. This should be fixed sometime.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>view_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the view (always the current database)</td>
</tr>
<tr>
<td>view_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the view</td>
</tr>
<tr>
<td>view_name</td>
<td>sql_identifier</td>
<td>Name of the view</td>
</tr>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the table that is used by the view (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the table that is used by the view</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the table that is used by the view</td>
</tr>
</tbody>
</table>

33.47. **views**

The view **views** contains all views defined in the current database. Only those views are shown that the current user has access to (by way of being the owner or having some privilege).

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_catalog</td>
<td>sql_identifier</td>
<td>Name of the database that contains the view (always the current database)</td>
</tr>
<tr>
<td>table_schema</td>
<td>sql_identifier</td>
<td>Name of the schema that contains the view</td>
</tr>
<tr>
<td>table_name</td>
<td>sql_identifier</td>
<td>Name of the view</td>
</tr>
<tr>
<td>view_definition</td>
<td>character_data</td>
<td>Query expression defining the view (null if the view is not owned by a currently enabled role)</td>
</tr>
</tbody>
</table>
### Chapter 33. The Information Schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_option</td>
<td>character_data</td>
<td>Applies to a feature not available in PostgreSQL</td>
</tr>
<tr>
<td>is_updatable</td>
<td>character_data</td>
<td>YES if the view is updatable (allows UPDATE and DELETE), NO if not</td>
</tr>
<tr>
<td>is_insertable_into</td>
<td>character_data</td>
<td>YES if the view is insertable into (allows INSERT), NO if not</td>
</tr>
</tbody>
</table>
V. Server Programming

This part is about extending the server functionality with user-defined functions, data types, triggers, etc. These are advanced topics which should probably be approached only after all the other user documentation about PostgreSQL has been understood. Later chapters in this part describe the server-side programming languages available in the PostgreSQL distribution as well as general issues concerning server-side programming languages. It is essential to read at least the earlier sections of Chapter 34 (covering functions) before diving into the material about server-side programming languages.
Chapter 34. Extending SQL

In the sections that follow, we will discuss how you can extend the PostgreSQL SQL query language by adding:

- functions (starting in Section 34.3)
- aggregates (starting in Section 34.10)
- data types (starting in Section 34.11)
- operators (starting in Section 34.12)
- operator classes for indexes (starting in Section 34.14)

34.1. How Extensibility Works

PostgreSQL is extensible because its operation is catalog-driven. If you are familiar with standard relational database systems, you know that they store information about databases, tables, columns, etc., in what are commonly known as system catalogs. (Some systems call this the data dictionary.) The catalogs appear to the user as tables like any other, but the DBMS stores its internal bookkeeping in them. One key difference between PostgreSQL and standard relational database systems is that PostgreSQL stores much more information in its catalogs: not only information about tables and columns, but also information about data types, functions, access methods, and so on. These tables can be modified by the user, and since PostgreSQL bases its operation on these tables, this means that PostgreSQL can be extended by users. By comparison, conventional database systems can only be extended by changing hardcoded procedures in the source code or by loading modules specially written by the DBMS vendor.

The PostgreSQL server can moreover incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a shared library) that implements a new type or function, and PostgreSQL will load it as required. Code written in SQL is even more trivial to add to the server. This ability to modify its operation “on the fly” makes PostgreSQL uniquely suited for rapid prototyping of new applications and storage structures.

34.2. The PostgreSQL Type System

PostgreSQL data types are divided into base types, composite types, domains, and pseudo-types.

34.2.1. Base Types

Base types are those, like int4, that are implemented below the level of the SQL language (typically in a low-level language such as C). They generally correspond to what are often known as abstract data types. PostgreSQL can only operate on such types through functions provided by the user and only understands the behavior of such types to the extent that the user describes them. Base types are further subdivided into scalar and array types. For each scalar type, a corresponding array type is automatically created that can hold variable-size arrays of that scalar type.
34.2.2. Composite Types

Composite types, or row types, are created whenever the user creates a table. It is also possible to use CREATE TYPE to define a “stand-alone” composite type with no associated table. A composite type is simply a list of types with associated field names. A value of a composite type is a row or record of field values. The user can access the component fields from SQL queries. Refer to Section 8.15 for more information on composite types.

34.2.3. Domains

A domain is based on a particular base type and for many purposes is interchangeable with its base type. However, a domain can have constraints that restrict its valid values to a subset of what the underlying base type would allow.

Domains can be created using the SQL command CREATE DOMAIN. Their creation and use is not discussed in this chapter.

34.2.4. Pseudo-Types

There are a few “pseudo-types” for special purposes. Pseudo-types cannot appear as columns of tables or attributes of composite types, but they can be used to declare the argument and result types of functions. This provides a mechanism within the type system to identify special classes of functions. Table 8-20 lists the existing pseudo-types.

34.2.5. Polymorphic Types

Four pseudo-types of special interest are anyelement, anyarray, anynonarray, and anyenum, which are collectively called polymorphic types. Any function declared using these types is said to be a polymorphic function. A polymorphic function can operate on many different data types, with the specific data type(s) being determined by the data types actually passed to it in a particular call.

Polymorphic arguments and results are tied to each other and are resolved to a specific data type when a query calling a polymorphic function is parsed. Each position (either argument or return value) declared as anyelement is allowed to have any specific actual data type, but in any given call they must all be the same actual type. Each position declared as anyarray can have any array data type, but similarly they must all be the same type. If there are positions declared anyelement and others declared anyarray, the actual array type in the anyarray positions must be an array whose elements are the same type appearing in the anyelement positions. anynonarray is treated exactly the same as anyelement, but adds the additional constraint that the actual type must not be an array type. anyenum is treated exactly the same as anyelement, but adds the additional constraint that the actual type must be an enum type.

Thus, when more than one argument position is declared with a polymorphic type, the net effect is that only certain combinations of actual argument types are allowed. For example, a function declared as equal(anyelement, anyelement) will take any two input values, so long as they are of the same data type.

When the return value of a function is declared as a polymorphic type, there must be at least one argument position that is also polymorphic, and the actual data type supplied as the argument determines the actual result type for that call. For example, if there were not already an array subscripting mechanism, one could define a function that implements subscripting as subscript(anyarray, integer) returns anyelement. This declaration constrains the actual first argument to be an
array type, and allows the parser to infer the correct result type from the actual first argument’s type. Another example is that a function declared as \( f(\text{anyarray}) \) returns anyenum will only accept arrays of enum types.

Note that anynonarray and anyenum do not represent separate type variables; they are the same type as anyelement, just with an additional constraint. For example, declaring a function as \( f(\text{anyelement}, \text{anyenum}) \) is equivalent to declaring it as \( f(\text{anyenum}, \text{anyenum}) \): both actual arguments have to be the same enum type.

### 34.3. User-Defined Functions

PostgreSQL provides four kinds of functions:

- query language functions (functions written in SQL) (Section 34.4)
- procedural language functions (functions written in, for example, PL/pgSQL or PL/Tcl) (Section 34.7)
- internal functions (Section 34.8)
- C-language functions (Section 34.9)

Every kind of function can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.

Many kinds of functions can take or return certain pseudo-types (such as polymorphic types), but the available facilities vary. Consult the description of each kind of function for more details.

It’s easiest to define SQL functions, so we’ll start by discussing those. Most of the concepts presented for SQL functions will carry over to the other types of functions.

Throughout this chapter, it can be useful to look at the reference page of the `CREATE FUNCTION` command to understand the examples better. Some examples from this chapter can be found in `funcs.sql` and `funcs.c` in the `src/tutorial` directory in the PostgreSQL source distribution.

### 34.4. Query Language (SQL) Functions

SQL functions execute an arbitrary list of SQL statements, returning the result of the last query in the list. In the simple (non-set) case, the first row of the last query’s result will be returned. (Bear in mind that “the first row” of a multirow result is not well-defined unless you use `ORDER BY`.) If the last query happens to return no rows at all, the null value will be returned.

Alternatively, an SQL function can be declared to return a set, by specifying the function’s return type as `SETOF sometype`. In this case all rows of the last query’s result are returned. Further details appear below.

The body of an SQL function must be a list of SQL statements separated by semicolons. A semicolon after the last statement is optional. Unless the function is declared to return `void`, the last statement must be a `SELECT`. 
Any collection of commands in the SQL language can be packaged together and defined as a function. Besides SELECT queries, the commands can include data modification queries (INSERT, UPDATE, and DELETE), as well as other SQL commands. (The only exception is that you cannot put BEGIN, COMMIT, ROLLBACK, or SAVEPOINT commands into a SQL function.) However, the final command must be a SELECT that returns whatever is specified as the function’s return type. Alternatively, if you want to define a SQL function that performs actions but has no useful value to return, you can define it as returning void. In that case, the function body must not end with a SELECT. For example, this function removes rows with negative salaries from the emp table:

```
CREATE FUNCTION clean_emp() RETURNS void AS 'DELETE FROM emp
   WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();
```

```
clean_emp
-----------
(1 row)
```

The syntax of the CREATE FUNCTION command requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see Section 4.1.2.2) for the string constant. If you choose to use regular single-quoted string constant syntax, you must double single quote marks (‘) and backslashes (\) (assuming escape string syntax) in the body of the function (see Section 4.1.2.1).

Arguments to the SQL function are referenced in the function body using the syntax $n$: $1 refers to the first argument, $2 to the second, and so on. If an argument is of a composite type, then the dot notation, e.g., $1.name, can be used to access attributes of the argument. The arguments can only be used as data values, not as identifiers. Thus for example this is reasonable:

```
INSERT INTO mytable VALUES ($1);
```

but this will not work:

```
INSERT INTO $1 VALUES (42);
```

### 34.4.1. SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as integer:

```
CREATE FUNCTION one() RETURNS integer AS $$
   SELECT 1 AS result;
$$ LANGUAGE SQL;
```

```
-- Alternative syntax for string literal:
CREATE FUNCTION one() RETURNS integer AS 'SELECT 1 AS result;
' LANGUAGE SQL;
```

633
SELECT one();

one
-----
  1

Notice that we defined a column alias within the function body for the result of the function (with the name `result`), but this column alias is not visible outside the function. Hence, the result is labeled `one` instead of `result`.

It is almost as easy to define SQL functions that take base types as arguments. In the example below, notice how we refer to the arguments within the function as `$1` and `$2`.

```sql
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
SELECT $1 + $2;
$$ LANGUAGE SQL;
SELECT add_em(1, 2) AS answer;

answer
--------
   3
```

Here is a more useful function, which might be used to debit a bank account:

```sql
CREATE FUNCTION tf1 (integer, numeric) RETURNS integer AS $$
UPDATE bank
    SET balance = balance - $2
    WHERE accountno = $1;
SELECT 1;
$$ LANGUAGE SQL;

A user could execute this function to debit account 17 by $100.00 as follows:

SELECT tf1(17, 100.0);

In practice one would probably like a more useful result from the function than a constant 1, so a more likely definition is:

```sql
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
UPDATE bank
    SET balance = balance - $2
    WHERE accountno = $1;
SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;
```

which adjusts the balance and returns the new balance.
Chapter 34. Extending SQL

34.4.2. SQL Functions on Composite Types

When writing functions with arguments of composite types, we must not only specify which argument we want (as we did above with $1$ and $2$) but also the desired attribute (field) of that argument. For example, suppose that emp is a table containing employee data, and therefore also the name of the composite type of each row of the table. Here is a function double_salary that computes what someone’s salary would be if it were doubled:

```sql
CREATE TABLE emp (  
    name text,  
    salary numeric,  
    age integer,  
    cubicle point  
) ;

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream  
FROM emp  
WHERE emp.cubicle ~= point '(2,1)';

name | dream
----- | -------
Bill  |  8400
```

Notice the use of the syntax $1.salary to select one field of the argument row value. Also notice how the calling SELECT command uses * to select the entire current row of a table as a composite value. The table row can alternatively be referenced using just the table name, like this:

```sql
SELECT name, double_salary(emp) AS dream  
FROM emp  
WHERE emp.cubicle ~= point '(2,1)';
```

but this usage is deprecated since it’s easy to get confused.

Sometimes it is handy to construct a composite argument value on-the-fly. This can be done with the ROW construct. For example, we could adjust the data being passed to the function:

```sql
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream  
FROM emp;
```

It is also possible to build a function that returns a composite type. This is an example of a function that returns a single emp row:

```sql
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,  
           1000.0 AS salary,  
           25 AS age,  
           point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

In this example we have specified each of the attributes with a constant value, but any computation could have been substituted for these constants.
Note two important things about defining the function:

- The select list order in the query must be exactly the same as that in which the columns appear in
  the table associated with the composite type. (Naming the columns, as we did above, is irrelevant
  to the system.)
- You must typecast the expressions to match the definition of the composite type, or you will get
  errors like this:

```
ERROR: function declared to return emp returns varchar instead of text at column 1
```

A different way to define the same function is:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Here we wrote a `SELECT` that returns just a single column of the correct composite type. This isn’t
really better in this situation, but it is a handy alternative in some cases — for example, if we need to
compute the result by calling another function that returns the desired composite value.

We could call this function directly in either of two ways:

```
SELECT new_emp();
```

```
new_emp
---------
(None,1000.0,25,"(2,2)")
```

```
SELECT * FROM new_emp();
```

```
name | salary | age | cubicle
------|--------|-----|---------
None  | 1000.0 | 25 | (2,2)
```

The second way is described more fully in Section 34.4.4.

When you use a function that returns a composite type, you might want only one field (attribute) from
its result. You can do that with syntax like this:

```
SELECT (new_emp()).name;
```

```
name
----
None
```

The extra parentheses are needed to keep the parser from getting confused. If you try to do it without
them, you get something like this:

```
SELECT new_emp().name;
```

```
ERROR: syntax error at or near "." at character 17
LINE 1: SELECT new_emp().name;
```

Another option is to use functional notation for extracting an attribute. The simple way to explain this
is that we can use the notations `attribute(table)` and `table.attribute` interchangeably.
SELECT name(new_emp());

name
------
None

-- This is the same as:
-- SELECT emp.name AS youngster FROM emp WHERE emp.age < 30;

SELECT name(emp) AS youngster FROM emp WHERE age(emp) < 30;

youngster
----------
Sam
Andy

Tip: The equivalence between functional notation and attribute notation makes it possible to use functions on composite types to emulate “computed fields”. For example, using the previous definition for double_salary(emp), we can write

SELECT emp.name, emp.double_salary FROM emp;

An application using this wouldn’t need to be directly aware that double_salary isn’t a real column of the table. (You can also emulate computed fields with views.)

Another way to use a function returning a composite type is to pass the result to another function that accepts the correct row type as input:

CREATE FUNCTION getname(emp) RETURNS text AS $$
SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
getname
---------
None
(1 row)

Still another way to use a function that returns a composite type is to call it as a table function, as described in Section 34.4.4.

34.4.3. Functions with Output Parameters

An alternative way of describing a function’s results is to define it with output parameters, as in this example:

CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT $1 + $2'
LANGUAGE SQL;

SELECT add_em(3,7);
This is not essentially different from the version of add_em shown in Section 34.4.1. The real value of output parameters is that they provide a convenient way of defining functions that return several columns. For example,

```sql
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

```sql
SELECT * FROM sum_n_product(11,42);
```

<table>
<thead>
<tr>
<th>sum</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>462</td>
</tr>
</tbody>
</table>

What has essentially happened here is that we have created an anonymous composite type for the result of the function. The above example has the same end result as

```sql
CREATE TYPE sum_prod AS (sum int, product int);
CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

but not having to bother with the separate composite type definition is often handy.

Notice that output parameters are not included in the calling argument list when invoking such a function from SQL. This is because PostgreSQL considers only the input parameters to define the function’s calling signature. That means also that only the input parameters matter when referencing the function for purposes such as dropping it. We could drop the above function with either of

```sql
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

Parameters can be marked as **IN** (the default), **OUT**, or **INOUT**. An **INOUT** parameter serves as both an input parameter (part of the calling argument list) and an output parameter (part of the result record type).

### 34.4.4. SQL Functions as Table Sources

All SQL functions can be used in the **FROM** clause of a query, but it is particularly useful for functions returning composite types. If the function is defined to return a base type, the table function produces a one-column table. If the function is defined to return a composite type, the table function produces a column for each attribute of the composite type.

Here is an example:

```sql
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');
```
### 34.4.5. SQL Functions Returning Sets

When an SQL function is declared as returning `SETOF sometype`, the function’s final `SELECT` query is executed to completion, and each row it outputs is returned as an element of the result set.

This feature is normally used when calling the function in the `FROM` clause. In this case each row returned by the function becomes a row of the table seen by the query. For example, assume that table `foo` has the same contents as above, and we say:

```sql
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

```sql
SELECT * FROM getfoo(1) AS t1;
```

Then we would get:

<table>
<thead>
<tr>
<th>fooid</th>
<th>foosubid</th>
<th>fooname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Ed</td>
</tr>
</tbody>
</table>

(2 rows)

It is also possible to return multiple rows with the columns defined by output parameters, like this:

```sql
CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int) RETURNS SETOF record AS $$
    SELECT x + tab.y, x * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Then we would get:

<table>
<thead>
<tr>
<th>fooid</th>
<th>foosubid</th>
<th>fooname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Ed</td>
</tr>
</tbody>
</table>

(2 rows)

The key point here is that you must write `RETURNS SETOF record` to indicate that the function returns multiple rows instead of just one. If there is only one output parameter, write that parameter’s type instead of `record`.

Currently, functions returning sets can also be called in the select list of a query. For each row that the query generates by itself, the function returning set is invoked, and an output row is generated for each element of the function’s result set. Note, however, that this capability is deprecated and might...
be removed in future releases. The following is an example function returning a set from the select list:

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL;
```

```
SELECT * FROM nodes;
  name | parent
-------+--------
  Top  |       
  Child1 | Top
  Child2 | Top
  Child3 | Top
  SubChild1 | Child1
  SubChild2 | Child1
(6 rows)
```

```
SELECT listchildren('Top');

  listchildren
  ------------
   Child1
   Child2
   Child3
(3 rows)
```

```
SELECT name, listchildren(name) FROM nodes;

  name | listchildren
  --------+--------------
   Top | Child1
   Top | Child2
   Top | Child3
  Child1 | SubChild1
  Child1 | SubChild2
(5 rows)
```

In the last `SELECT`, notice that no output row appears for `Child2`, `Child3`, etc. This happens because `listchildren` returns an empty set for those arguments, so no result rows are generated.

### 34.4.6. Polymorphic SQL Functions

SQL functions can be declared to accept and return the polymorphic types `anyelement`, `anyarray`, `anynonarray`, and `anyenum`. See Section 34.2.5 for a more detailed explanation of polymorphic functions. Here is a polymorphic function `make_array` that builds up an array from two arbitrary data type elements:

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
```

```
SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;

  intarray | textarray
----------+-----------
     {1,2} |     {a,b}
(1 row)
```
Notice the use of the typecast 'a'::text to specify that the argument is of type text. This is required if the argument is just a string literal, since otherwise it would be treated as type unknown, and array of unknown is not a valid type. Without the typecast, you will get errors like this:

ERROR: could not determine polymorphic type because input has type "unknown"

It is permitted to have polymorphic arguments with a fixed return type, but the converse is not. For example:

```sql
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;
```

```sql
SELECT is_greater(1, 2);
```

```sql
is_greater
------------
  f
(1 row)
```

```sql
CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
```

```sql
ERROR: cannot determine result data type
DETAIL: A function returning a polymorphic type must have at least one polymorphic argument.
```

Polymorphism can be used with functions that have output arguments. For example:

```sql
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray) AS 'select $1, array[$1,$1]' LANGUAGE sql;
```

```sql
SELECT * FROM dup(22);
```

```sql
  f2 | f3
----+---------
   22 | {22,22}
(1 row)
```

### 34.5. Function Overloading

More than one function can be defined with the same SQL name, so long as the arguments they take are different. In other words, function names can be overloaded. When a query is executed, the server will determine which function to call from the data types and the number of the provided arguments. Overloading can also be used to simulate functions with a variable number of arguments, up to a finite maximum number.

When creating a family of overloaded functions, one should be careful not to create ambiguities. For instance, given the functions:

```sql
CREATE FUNCTION test(int, real) RETURNS ...
```
CREATE FUNCTION test(smallint, double precision) RETURNS ... 

it is not immediately clear which function would be called with some trivial input like \texttt{test(1, 1.5)}. The currently implemented resolution rules are described in Chapter 10, but it is unwise to design a system that subtly relies on this behavior.

A function that takes a single argument of a composite type should generally not have the same name as any attribute (field) of that type. Recall that \texttt{attribute(table)} is considered equivalent to \texttt{table.attribute}. In the case that there is an ambiguity between a function on a composite type and an attribute of the composite type, the attribute will always be used. It is possible to override that choice by schema-qualifying the function name (that is, \texttt{schema.func(table)}) but it's better to avoid the problem by not choosing conflicting names.

When overloading C-language functions, there is an additional constraint: The C name of each function in the family of overloaded functions must be different from the C names of all other functions, either internal or dynamically loaded. If this rule is violated, the behavior is not portable. You might get a run-time linker error, or one of the functions will get called (usually the internal one). The alternative form of the \texttt{AS} clause for the SQL \texttt{CREATE FUNCTION} command decouples the SQL function name from the function name in the C source code. For instance:

\begin{verbatim}
CREATE FUNCTION test(int) RETURNS int
  AS 'filename', 'test_1arg'
  LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
  AS 'filename', 'test_2arg'
  LANGUAGE C;
\end{verbatim}

The names of the C functions here reflect one of many possible conventions.

### 34.6. Function Volatility Categories

Every function has a \textit{volatility} classification, with the possibilities being \texttt{VOLATILE}, \texttt{STABLE}, or \texttt{IMMUTABLE}. \texttt{VOLATILE} is the default if the \texttt{CREATE FUNCTION} command does not specify a category. The volatility category is a promise to the optimizer about the behavior of the function:

- A \texttt{VOLATILE} function can do anything, including modifying the database. It can return different results on successive calls with the same arguments. The optimizer makes no assumptions about the behavior of such functions. A query using a volatile function will re-evaluate the function at every row where its value is needed.

- A \texttt{STABLE} function cannot modify the database and is guaranteed to return the same results given the same arguments for all rows within a single statement. This category allows the optimizer to optimize multiple calls of the function to a single call. In particular, it is safe to use an expression containing such a function in an index scan condition. (Since an index scan will evaluate the comparison value only once, not once at each row, it is not valid to use a \texttt{VOLATILE} function in an index scan condition.)

- An \texttt{IMMUTABLE} function cannot modify the database and is guaranteed to return the same results given the same arguments forever. This category allows the optimizer to pre-evaluate the function when a query calls it with constant arguments. For example, a query like \texttt{SELECT ... WHERE x = 2 + 2} can be simplified on sight to \texttt{SELECT ... WHERE x = 4}, because the function underlying the integer addition operator is marked \texttt{IMMUTABLE}. 

642
For best optimization results, you should label your functions with the strictest volatility category that is valid for them.

Any function with side-effects must be labeled \texttt{VOLATILE}, so that calls to it cannot be optimized away. Even a function with no side-effects needs to be labeled \texttt{VOLATILE} if its value can change within a single query; some examples are \texttt{random()}, \texttt{currval()}, \texttt{timeofday()}.

There is relatively little difference between \texttt{STABLE} and \texttt{IMMUTABLE} categories when considering simple interactive queries that are planned and immediately executed: it doesn’t matter a lot whether a function is executed once during planning or once during query execution startup. But there is a big difference if the plan is saved and reused later. Labeling a function \texttt{IMMUTABLE} when it really isn’t might allow it to be prematurely folded to a constant during planning, resulting in a stale value being re-used during subsequent uses of the plan. This is a hazard when using prepared statements or when using function languages that cache plans (such as PL/pgSQL).

Because of the snapshotting behavior of MVCC (see Chapter 13) a function containing only \texttt{SELECT} commands can safely be marked \texttt{STABLE}, even if it selects from tables that might be undergoing modifications by concurrent queries. PostgreSQL will execute a \texttt{STABLE} function using the snapshot established for the calling query, and so it will see a fixed view of the database throughout that query.

Also note that the \texttt{current_timestamp} family of functions qualify as stable, since their values do not change within a transaction.

The same snapshotting behavior is used for \texttt{SELECT} commands within \texttt{IMMUTABLE} functions. It is generally unwise to select from database tables within an \texttt{IMMUTABLE} function at all, since the immutability will be broken if the table contents ever change. However, PostgreSQL does not enforce that you do not do that.

A common error is to label a function \texttt{IMMUTABLE} when its results depend on a configuration parameter. For example, a function that manipulates timestamps might well have results that depend on the timezone setting. For safety, such functions should be labeled \texttt{STABLE} instead.

\textbf{Note:} Before PostgreSQL release 8.0, the requirement that \texttt{STABLE} and \texttt{IMMUTABLE} functions cannot modify the database was not enforced by the system. Release 8.0 enforces it by requiring SQL functions and procedural language functions of these categories to contain no SQL commands other than \texttt{SELECT}. (This is not a completely bulletproof test, since such functions could still call \texttt{VOLATILE} functions that modify the database. If you do that, you will find that the \texttt{STABLE} or \texttt{IMMUTABLE} function does not notice the database changes applied by the called function.)

### 34.7. Procedural Language Functions

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called \textit{procedural languages} (PLs). Procedural languages aren’t built into the PostgreSQL server; they are offered by loadable modules. See Chapter 37 and following chapters for more information.

### 34.8. Internal Functions

Internal functions are functions written in C that have been statically linked into the PostgreSQL server. The “body” of the function definition specifies the C-language name of the function, which
need not be the same as the name being declared for SQL use. (For reasons of backwards compatibility, an empty body is accepted as meaning that the C-language function name is the same as the SQL name.)

Normally, all internal functions present in the server are declared during the initialization of the database cluster (initdb), but a user could use CREATE FUNCTION to create additional alias names for an internal function. Internal functions are declared in CREATE FUNCTION with language name internal. For instance, to create an alias for the sqrt function:

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
  AS 'dsqrt'
  LANGUAGE internal
  STRICT;
```

(Most internal functions expect to be declared “strict”.)

Note: Not all “predefined” functions are “internal” in the above sense. Some predefined functions are written in SQL.

### 34.9. C-Language Functions

User-defined functions can be written in C (or a language that can be made compatible with C, such as C++). Such functions are compiled into dynamically loadable objects (also called shared libraries) and are loaded by the server on demand. The dynamic loading feature is what distinguishes “C language” functions from “internal” functions — the actual coding conventions are essentially the same for both. (Hence, the standard internal function library is a rich source of coding examples for user-defined C functions.)

Two different calling conventions are currently used for C functions. The newer “version 1” calling convention is indicated by writing a `PG_FUNCTION_INFO_V1()` macro call for the function, as illustrated below. Lack of such a macro indicates an old-style (“version 0”) function. The language name specified in CREATE FUNCTION is C in either case. Old-style functions are now deprecated because of portability problems and lack of functionality, but they are still supported for compatibility reasons.

#### 34.9.1. Dynamic Loading

The first time a user-defined function in a particular loadable object file is called in a session, the dynamic loader loads that object file into memory so that the function can be called. The CREATE FUNCTION for a user-defined C function must therefore specify two pieces of information for the function: the name of the loadable object file, and the C name (link symbol) of the specific function to call within that object file. If the C name is not explicitly specified then it is assumed to be the same as the SQL function name.

The following algorithm is used to locate the shared object file based on the name given in the CREATE FUNCTION command:

1. If the name is an absolute path, the given file is loaded.
2. If the name starts with the string `$libdir`, that part is replaced by the PostgreSQL package library directory name, which is determined at build time.
3. If the name does not contain a directory part, the file is searched for in the path specified by the configuration variable dynamic_library_path.

4. Otherwise (the file was not found in the path, or it contains a non-absolute directory part), the dynamic loader will try to take the name as given, which will most likely fail. (It is unreliable to depend on the current working directory.)

If this sequence does not work, the platform-specific shared library file name extension (often .so) is appended to the given name and this sequence is tried again. If that fails as well, the load will fail.

It is recommended to locate shared libraries either relative to $libdir or through the dynamic library path. This simplifies version upgrades if the new installation is at a different location. The actual directory that $libdir stands for can be found out with the command `pg_config --pkglibdir`.

The user ID the PostgreSQL server runs as must be able to traverse the path to the file you intend to load. Making the file or a higher-level directory not readable and/or not executable by the postgres user is a common mistake.

In any case, the file name that is given in the `CREATE FUNCTION` command is recorded literally in the system catalogs, so if the file needs to be loaded again the same procedure is applied.

**Note:** PostgreSQL will not compile a C function automatically. The object file must be compiled before it is referenced in a `CREATE FUNCTION` command. See Section 34.9.6 for additional information.

To ensure that a dynamically loaded object file is not loaded into an incompatible server, PostgreSQL checks that the file contains a “magic block” with the appropriate contents. This allows the server to detect obvious incompatibilities, such as code compiled for a different major version of PostgreSQL.

A magic block is required as of PostgreSQL 8.2. To include a magic block, write this in one (and only one) of the module source files, after having included the header `fmgr.h`:

```c
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

The `#ifdef` test can be omitted if the code doesn’t need to compile against pre-8.2 PostgreSQL releases.

After it is used for the first time, a dynamically loaded object file is retained in memory. Future calls in the same session to the function(s) in that file will only incur the small overhead of a symbol table lookup. If you need to force a reload of an object file, for example after recompiling it, begin a fresh session.

Optionally, a dynamically loaded file can contain initialization and finalization functions. If the file includes a function named `_PG_init`, that function will be called immediately after loading the file. The function receives no parameters and should return void. If the file includes a function named `_PG_fini`, that function will be called immediately before unloading the file. Likewise, the function receives no parameters and should return void. Note that `_PG_fini` will only be called during an unload of the file, not during process termination. (Presently, unloads are disabled and will never occur, but this may change in the future.)

### 34.9.2. Base Types in C-Language Functions

To know how to write C-language functions, you need to know how PostgreSQL internally represents base data types and how they can be passed to and from functions. Internally, PostgreSQL regards a
base type as a “blob of memory”. The user-defined functions that you define over a type in turn define the way that PostgreSQL can operate on it. That is, PostgreSQL will only store and retrieve the data from disk and use your user-defined functions to input, process, and output the data.

Base types can have one of three internal formats:

- pass by value, fixed-length
- pass by reference, fixed-length
- pass by reference, variable-length

By-value types can only be 1, 2, or 4 bytes in length (also 8 bytes, if sizeof(Datum) is 8 on your machine). You should be careful to define your types such that they will be the same size (in bytes) on all architectures. For example, the long type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas int type is 4 bytes on most Unix machines. A reasonable implementation of the int4 type on Unix machines might be:

```c
/* 4-byte integer, passed by value */
typedef int int4;
```

On the other hand, fixed-length types of any size can be passed by-reference. For example, here is a sample implementation of a PostgreSQL type:

```c
/* 16-byte structure, passed by reference */
typedef struct
{
    double x, y;
} Point;
```

Only pointers to such types can be used when passing them in and out of PostgreSQL functions. To return a value of such a type, allocate the right amount of memory with `palloc`, fill in the allocated memory, and return a pointer to it. (Also, if you just want to return the same value as one of your input arguments that’s of the same data type, you can skip the extra `palloc` and just return the pointer to the input value.)

Finally, all variable-length types must also be passed by reference. All variable-length types must begin with a length field of exactly 4 bytes, and all data to be stored within that type must be located in the memory immediately following that length field. The length field contains the total length of the structure, that is, it includes the size of the length field itself.

**Warning**

Never modify the contents of a pass-by-reference input value. If you do so you are likely to corrupt on-disk data, since the pointer you are given might point directly into a disk buffer. The sole exception to this rule is explained in Section 34.10.

As an example, we can define the type text as follows:

```c
typedef struct {
    int4 length;
    char data[1];
} text;
```
Chapter 34. Extending SQL

Obviously, the data field declared here is not long enough to hold all possible strings. Since it’s impossible to declare a variable-size structure in C, we rely on the knowledge that the C compiler won’t range-check array subscripts. We just allocate the necessary amount of space and then access the array as if it were declared the right length. (This is a common trick, which you can read about in many textbooks about C.)

When manipulating variable-length types, we must be careful to allocate the correct amount of memory and set the length field correctly. For example, if we wanted to store 40 bytes in a text structure, we might use a code fragment like this:

```c
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memcpy(destination->data, buffer, 40);
...
```

`VARHDRSZ` is the same as `sizeof(int4)`, but it’s considered good style to use the macro `VARHDRSZ` to refer to the size of the overhead for a variable-length type.

Table 34-1 specifies which C type corresponds to which SQL type when writing a C-language function that uses a built-in type of PostgreSQL. The “Defined In” column gives the header file that needs to be included to get the type definition. (The actual definition might be in a different file that is included by the listed file. It is recommended that users stick to the defined interface.) Note that you should always include `postgres.h` first in any source file, because it declares a number of things that you will need anyway.

**Table 34-1. Equivalent C Types for Built-In SQL Types**

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>C Type</th>
<th>Defined In</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstime</td>
<td>AbsoluteTime</td>
<td>utils/nabstime.h</td>
</tr>
<tr>
<td>boolean</td>
<td>bool</td>
<td>postgres.h (maybe compiler built-in)</td>
</tr>
<tr>
<td>box</td>
<td>BOX*</td>
<td>utils/geo_decls.h</td>
</tr>
<tr>
<td>bytea</td>
<td>bytea*</td>
<td>postgres.h</td>
</tr>
<tr>
<td>&quot;char&quot;</td>
<td>char</td>
<td>(compiler built-in)</td>
</tr>
<tr>
<td>character</td>
<td>BpChar*</td>
<td>postgres.h</td>
</tr>
<tr>
<td>cid</td>
<td>CommandId</td>
<td>postgres.h</td>
</tr>
<tr>
<td>date</td>
<td>DateADT</td>
<td>utils/date.h</td>
</tr>
<tr>
<td>smallint (int2)</td>
<td>int2 or int16</td>
<td>postgres.h</td>
</tr>
<tr>
<td>int2vector</td>
<td>int2vector*</td>
<td>postgres.h</td>
</tr>
<tr>
<td>integer (int4)</td>
<td>int4 or int32</td>
<td>postgres.h</td>
</tr>
<tr>
<td>real (float4)</td>
<td>float4*</td>
<td>postgres.h</td>
</tr>
<tr>
<td>double precision (float8)</td>
<td>float8*</td>
<td>postgres.h</td>
</tr>
<tr>
<td>interval</td>
<td>Interval*</td>
<td>utils/timestamp.h</td>
</tr>
<tr>
<td>lseg</td>
<td>LSEG*</td>
<td>utils/geo_decls.h</td>
</tr>
<tr>
<td>name</td>
<td>Name</td>
<td>postgres.h</td>
</tr>
<tr>
<td>SQL Type</td>
<td>C Type</td>
<td>Defined In</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>oid</td>
<td>Oid</td>
<td>postgres.h</td>
</tr>
<tr>
<td>oidvector</td>
<td>oidvector*</td>
<td>postgres.h</td>
</tr>
<tr>
<td>path</td>
<td>PATH*</td>
<td>utils/geo_decls.h</td>
</tr>
<tr>
<td>point</td>
<td>POINT*</td>
<td>utils/geo_decls.h</td>
</tr>
<tr>
<td>regproc</td>
<td>regproc</td>
<td>postgres.h</td>
</tr>
<tr>
<td>reltime</td>
<td>RelativeTime</td>
<td>utils/nabstime.h</td>
</tr>
<tr>
<td>text</td>
<td>text*</td>
<td>postgres.h</td>
</tr>
<tr>
<td>tid</td>
<td>ItemPointer</td>
<td>storage/itemptr.h</td>
</tr>
<tr>
<td>time</td>
<td>TimeADT</td>
<td>utils/date.h</td>
</tr>
<tr>
<td>time with time zone</td>
<td>TimeTzADT</td>
<td>utils/date.h</td>
</tr>
<tr>
<td>timestamp</td>
<td>Timestamp*</td>
<td>utils/timestamp.h</td>
</tr>
<tr>
<td>tinterval</td>
<td>TimeInterval</td>
<td>utils/nabstime.h</td>
</tr>
<tr>
<td>varchar</td>
<td>VarChar*</td>
<td>postgres.h</td>
</tr>
<tr>
<td>xid</td>
<td>TransactionId</td>
<td>postgres.h</td>
</tr>
</tbody>
</table>

Now that we’ve gone over all of the possible structures for base types, we can show some examples of real functions.

### 34.9.3. Version 0 Calling Conventions

We present the “old style” calling convention first — although this approach is now deprecated, it’s easier to get a handle on initially. In the version-0 method, the arguments and result of the C function are just declared in normal C style, but being careful to use the C representation of each SQL data type as shown above.

Here are some examples:

```c
#include "postgres.h"
#include <string.h>

/* by value */

int
add_one(int arg)
{
    return arg + 1;
}

/* by reference, fixed length */

float8 *
add_one_float8(float8 *arg)
{
    float8  *result = (float8 *) palloc(sizeof(float8));
    *result = *arg + 1.0;
    return result;
}
```
Point *
makepoint(Point *pointx, Point *pointy)
{
    Point *new_point = (Point *) palloc(sizeof(Point));
    new_point->x = pointx->x;
    new_point->y = pointy->y;
    return new_point;
}

/* by reference, variable length */
text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
            (void *) VARDATA(t), /* source */
            VARSIZE(t) - VARHDRSZ); /* how many bytes */
    return new_t;
}

/*
 * add_one(integer) returns integer
*/
text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);
    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
            VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    return new_text;
}

Supposing that the above code has been prepared in file func.c and compiled into a shared object, we could define the functions to PostgreSQL with commands like this:

CREATE FUNCTION add_one(integer) RETURNS integer
    AS 'DIRECTORY/func', 'add_one'
    LANGUAGE C STRICT;

-- note overloading of SQL function name "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
    AS 'DIRECTORY/func', 'add_one_float8'
    LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
Chapter 34. Extending SQL

AS 'DIRECTORY/funcs', 'makepoint'
LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
AS 'DIRECTORY/funcs', 'copytext'
LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
AS 'DIRECTORY/funcs', 'concat_text'
LANGUAGE C STRICT;

Here, DIRECTORY stands for the directory of the shared library file (for instance the PostgreSQL tutorial directory, which contains the code for the examples used in this section). (Better style would be to use just 'funcs' in the AS clause, after having added DIRECTORY to the search path. In any case, we can omit the system-specific extension for a shared library, commonly .so or .sl.)

Notice that we have specified the functions as “strict”, meaning that the system should automatically assume a null result if any input value is null. By doing this, we avoid having to check for null inputs in the function code. Without this, we’d have to check for null values explicitly, by checking for a null pointer for each pass-by-reference argument. (For pass-by-value arguments, we don’t even have a way to check!)

Although this calling convention is simple to use, it is not very portable; on some architectures there are problems with passing data types that are smaller than int this way. Also, there is no simple way to return a null result, nor to cope with null arguments in any way other than making the function strict. The version-1 convention, presented next, overcomes these objections.

34.9.4. Version 1 Calling Conventions

The version-1 calling convention relies on macros to suppress most of the complexity of passing arguments and results. The C declaration of a version-1 function is always:

Datum funcname(PG_FUNCTION_ARGS)

In addition, the macro call:

PG_FUNCTION_INFO_V1(funcname);

must appear in the same source file. (Conventionally, it’s written just before the function itself.) This macro call is not needed for internal-language functions, since PostgreSQL assumes that all internal functions use the version-1 convention. It is, however, required for dynamically-loaded functions.

In a version-1 function, each actual argument is fetched using a PG_GETARG_xxx() macro that corresponds to the argument’s data type, and the result is returned using a PG_RETURN_xxx() macro for the return type. PG_GETARG_xxx() takes as its argument the number of the function argument to fetch, where the count starts at 0. PG_RETURN_xxx() takes as its argument the actual value to return.

Here we show the same functions as above, coded in version-1 style:

#include "postgres.h"
#include <string.h>
#include "fmgr.h"

/* by value */
Chapter 34. Extending SQL

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32 arg = PG_GETARG_INT32(0);
    PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */
PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature. */
    float8 arg = PG_GETARG_FLOAT8(0);
    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden. */
    Point *pointx = PG_GETARG_POINT_P(0);
    Point *pointy = PG_GETARG_POINT_P(1);
    Point *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */
PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = {text *} palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */ (void *) VARDATA(t), /* source */
VARSIZE(t) - VARHDRSZ); /* how many bytes */
PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
  text  *arg1 = PG_GETARG_TEXT_P(0);
  text  *arg2 = PG_GETARG_TEXT_P(1);
  int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
  text *new_text = (text *) palloc(new_text_size);
  SET_VARSIZE(new_text, new_text_size);
  memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
  memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
          VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
  PG_RETURN_TEXT_P(new_text);
}

The CREATE FUNCTION commands are the same as for the version-0 equivalents.

At first glance, the version-1 coding conventions might appear to be just pointless obscurantism. They
do, however, offer a number of improvements, because the macros can hide unnecessary detail. An
example is that in coding add_one_float8, we no longer need to be aware that float8 is a pass-
by-reference type. Another example is that the GETARG macros for variable-length types allow for
more efficient fetching of “toasted” (compressed or out-of-line) values.

One big improvement in version-1 functions is better handling of null inputs and results. The macro
PG_ARGISNULL(n) allows a function to test whether each input is null. (Of course, doing this is
only necessary in functions not declared “strict”.) As with the PG_GETARG_xxx() macros, the
input arguments are counted beginning at zero. Note that one should refrain from executing
PG_GETARG_xxx() until one has verified that the argument isn’t null. To return a null result, execute
PG_RETURN_NULL(); this works in both strict and nonstrict functions.

Other options provided in the new-style interface are two variants of the PG_GETARG_xxx() macros.
The first of these, PG_GETARG_xxx_COPY(), guarantees to return a copy of the specified argument
that is safe for writing into. (The normal macros will sometimes return a pointer to a value that is
physically stored in a table, which must not be written to. Using the PG_GETARG_xxx_COPY() macros guarantees a writable result.) The second variant consists of the PG_GETARG_xxx_SLICE() macros which take three arguments. The first is the number of the function argument (as above). The
second and third are the offset and length of the segment to be returned. Offsets are counted from
zero, and a negative length requests that the remainder of the value be returned. These macros provide
more efficient access to parts of large values in the case where they have storage type “external”.
(The storage type of a column can be specified using ALTER TABLE tablename ALTER COLUMN
colname SET STORAGE storagetype. storagetype is one of plain, external, extended, or
main.)

Finally, the version-1 function call conventions make it possible to return set results (Section 34.9.10)
and implement trigger functions (Chapter 35) and procedural-language call handlers (Chapter 48).
Version-1 code is also more portable than version-0, because it does not break restrictions on function
call protocol in the C standard. For more details see src/backend/utils/fmgr/README in the
source distribution.
34.9.5. Writing Code

Before we turn to the more advanced topics, we should discuss some coding rules for PostgreSQL C-language functions. While it might be possible to load functions written in languages other than C into PostgreSQL, this is usually difficult (when it is possible at all) because other languages, such as C++, FORTRAN, or Pascal often do not follow the same calling convention as C. That is, other languages do not pass argument and return values between functions in the same way. For this reason, we will assume that your C-language functions are actually written in C.

The basic rules for writing and building C functions are as follows:

- Use `pg_config --includedir-server` to find out where the PostgreSQL server header files are installed on your system (or the system that your users will be running on).
- Compiling and linking your code so that it can be dynamically loaded into PostgreSQL always requires special flags. See Section 34.9.6 for a detailed explanation of how to do it for your particular operating system.
- Remember to define a “magic block” for your shared library, as described in Section 34.9.1.
- When allocating memory, use the PostgreSQL functions `palloc` and `pfree` instead of the corresponding C library functions `malloc` and `free`. The memory allocated by `palloc` will be freed automatically at the end of each transaction, preventing memory leaks.
- Always zero the bytes of your structures using `memset`. Without this, it’s difficult to support hash indexes or hash joins, as you must pick out only the significant bits of your data structure to compute a hash. Even if you initialize all fields of your structure, there might be alignment padding (holes in the structure) that contain garbage values.
- Most of the internal PostgreSQL types are declared in `postgres.h`, while the function manager interfaces (e.g., `PG_FUNCTION_ARGS`, etc.) are in `fmgr.h`, so you will need to include at least these two files. For portability reasons it’s best to include `postgres.h` first, before any other system or user header files. Including `postgres.h` will also include `elog.h` and `palloc.h` for you.
- Symbol names defined within object files must not conflict with each other or with symbols defined in the PostgreSQL server executable. You will have to rename your functions or variables if you get error messages to this effect.

34.9.6. Compiling and Linking Dynamically-Loaded Functions

Before you are able to use your PostgreSQL extension functions written in C, they must be compiled and linked in a special way to produce a file that can be dynamically loaded by the server. To be precise, a shared library needs to be created.

For information beyond what is contained in this section you should read the documentation of your operating system, in particular the manual pages for the C compiler, `cc`, and the link editor, `ld`. In addition, the PostgreSQL source code contains several working examples in the `contrib` directory. If you rely on these examples you will make your modules dependent on the availability of the PostgreSQL source code, however.

Creating shared libraries is generally analogous to linking executables: first the source files are compiled into object files, then the object files are linked together. The object files need to be created as position-independent code (PIC), which conceptually means that they can be placed at an arbitrary location in memory when they are loaded by the executable. (Object files intended for executables
are usually not compiled that way.) The command to link a shared library contains special flags to
distinguish it from linking an executable (at least in theory — on some systems the practice is much
uglier).

In the following examples we assume that your source code is in a file foo.c and we will create a
shared library foo.so. The intermediate object file will be called foo.o unless otherwise noted. A
shared library can contain more than one object file, but we only use one here.

**BSD/OS**

The compiler flag to create PIC is `-fpic`. The linker flag to create shared libraries is `-shared`.

```
gcc -fpic -c foo.c
ld -shared -o foo.so foo.o
```

This is applicable as of version 4.0 of BSD/OS.

**FreeBSD**

The compiler flag to create PIC is `-fpic`. To create shared libraries the compiler flag is `-shared`.

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

This is applicable as of version 3.0 of FreeBSD.

**HP-UX**

The compiler flag of the system compiler to create PIC is `+z`. When using GCC it’s `-fpic`. The
linker flag for shared libraries is `-b`. So:

```
cc +z -c foo.c
```

or:

```
gcc -fpic -c foo.c
```

and then:

```
ld -b -o foo.sl foo.o
```

HP-UX uses the extension `.sl` for shared libraries, unlike most other systems.

**IRIX**

PIC is the default, no special compiler options are necessary. The linker option to produce shared
libraries is `-shared`.

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

**Linux**

The compiler flag to create PIC is `-fpic`. On some platforms in some situations `-fPIC` must be
used if `-fpic` does not work. Refer to the GCC manual for more information. The compiler flag
to create a shared library is `-shared`. A complete example looks like this:

```
cc -fpic -c foo.c
c -shared -o foo.so foo.o
```

**MacOS X**

Here is an example. It assumes the developer tools are installed.

```
cc -c foo.c
c -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```
Chapter 34. Extending SQL

NetBSD

The compiler flag to create PIC is `-fpic`. For ELF systems, the compiler with the flag `-shared` is used to link shared libraries. On the older non-ELF systems, `ld -Bshareable` is used.

```
gcc -fpic -c foo.c
```

```
gcc -shared -o foo.so foo.o
```

OpenBSD

The compiler flag to create PIC is `-fpic`. `ld -Bshareable` is used to link shared libraries.

```
gcc -fpic -c foo.c
```

```
ld -Bshareable -o foo.so foo.o
```

Solaris

The compiler flag to create PIC is `-KPIC` with the Sun compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with either compiler or alternatively `-shared` with GCC.

```
cc -KPIC -c foo.c
```

```
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
```

```
gcc -G -o foo.so foo.o
```

Tru64 UNIX

PIC is the default, so the compilation command is the usual one. `ld` with special options is used to do the linking.

```
cc -c foo.c
```

```
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

The same procedure is used with GCC instead of the system compiler; no special options are required.

UnixWare

The compiler flag to create PIC is `-K PIC` with the SCO compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with the SCO compiler and `-shared` with GCC.

```
cc -K PIC -c foo.c
```

```
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
```

```
gcc -shared -o foo.so foo.o
```

**Tip:** If this is too complicated for you, you should consider using GNU Libtool\(^1\), which hides the platform differences behind a uniform interface.

The resulting shared library file can then be loaded into PostgreSQL. When specifying the file name to the `CREATE FUNCTION` command, one must give it the name of the shared library file, not the intermediate object file. Note that the system’s standard shared-library extension (usually `.so` or `.sl`) can be omitted from the `CREATE FUNCTION` command, and normally should be omitted for best portability.

Refer back to Section 34.9.1 about where the server expects to find the shared library files.

---

\(^1\) [http://www.gnu.org/software/libtool/](http://www.gnu.org/software/libtool/)
34.9.7. Extension Building Infrastructure

If you are thinking about distributing your PostgreSQL extension modules, setting up a portable build system for them can be fairly difficult. Therefore the PostgreSQL installation provides a build infrastructure for extensions, called PGXS, so that simple extension modules can be built simply against an already installed server. Note that this infrastructure is not intended to be a universal build system framework that can be used to build all software interfacing to PostgreSQL; it simply automates common build rules for simple server extension modules. For more complicated packages, you need to write your own build system.

To use the infrastructure for your extension, you must write a simple makefile. In that makefile, you need to set some variables and finally include the global PGXS makefile. Here is an example that builds an extension module named `isbn_issn` consisting of a shared library, an SQL script, and a documentation text file:

```makefile
MODULES = isbn_issn
DATA_built = isbn_issn.sql
DOCS = README.isbn_issn

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

The last three lines should always be the same. Earlier in the file, you assign variables or add custom make rules.

The following variables can be set:

- **MODULES**
  - list of shared objects to be built from source file with same stem (do not include suffix in this list)

- **DATA**
  - random files to install into `prefix/share/contrib`

- **DATA_built**
  - random files to install into `prefix/share/contrib`, which need to be built first

- **DOCS**
  - random files to install under `prefix/doc/contrib`

- **SCRIPTS**
  - script files (not binaries) to install into `prefix/bin`

- **SCRIPTS_built**
  - script files (not binaries) to install into `prefix/bin`, which need to be built first

- **REGRESS**
  - list of regression test cases (without suffix), see below
  - or at most one of these two:

- **PROGRAM**
  - a binary program to build (list objects files in `OBJJS`)
MODULE_big

a shared object to build (list object files in OBJJS)

The following can also be set:

EXTRA_CLEAN

extra files to remove in make clean

PG_CPPFLAGS

will be added to CPPFLAGS

PG_LIBS

will be added to PROGRAM link line

SHLIB_LINK

will be added to MODULE_big link line

PG_CONFIG

path to pg_config program for the PostgreSQL installation to build against (typically just pg_config to use the first one in your PATH)

Put this makefile as Makefile in the directory which holds your extension. Then you can do make to compile, and later make install to install your module. By default, the extension is compiled and installed for the PostgreSQL installation that corresponds to the first pg_config program found in your path. You can use a different installation by setting PG_CONFIG to point to its pg_config program, either within the makefile or on the make command line.

Caution
Changing PG_CONFIG only works when building against PostgreSQL 8.3 or later. With older releases it does not work to set it to anything except pg_config; you must alter your PATH to select the installation to build against.

The scripts listed in the REGRESS variable are used for regression testing of your module, just like make installcheck is used for the main PostgreSQL server. For this to work you need to have a subdirectory named sql/ in your extension’s directory, within which you put one file for each group of tests you want to run. The files should have extension .sql, which should not be included in the REGRESS list in the makefile. For each test there should be a file containing the expected result in a subdirectory named expected/, with extension .out. The tests are run by executing make installcheck, and the resulting output will be compared to the expected files. The differences will be written to the file regression.diffs in diff -c format. Note that trying to run a test which is missing the expected file will be reported as “trouble”, so make sure you have all expected files.

Tip: The easiest way of creating the expected files is creating empty files, then carefully inspecting the result files after a test run (to be found in the results/ directory), and copying them to expected/ if they match what you want from the test.
34.9.8. Composite-Type Arguments

Composite types do not have a fixed layout like C structures. Instances of a composite type can contain null fields. In addition, composite types that are part of an inheritance hierarchy can have different fields than other members of the same inheritance hierarchy. Therefore, PostgreSQL provides a function interface for accessing fields of composite types from C.

Suppose we want to write a function to answer the query:

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
FROM emp
WHERE name = 'Bill' OR name = 'Sam';
```

Using call conventions version 0, we can define `c_overpaid` as:

```c
#include "postgres.h"
#include "executor/executor.h" /* for GetAttributeByName() */

bool
c_overpaid(HeapTupleHeader t, /* the current row of emp */
int32 limit)
{
  bool isnull;
  int32 salary;

  salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
  if (isnull)
    return false;
  return salary > limit;
}
```

In version-1 coding, the above would look like this:

```c
#include "postgres.h"
#include "executor/executor.h" /* for GetAttributeByName() */

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
  HeapTupleHeader t = PG_GETARG_HEAPTUPLEHEADER(0);
  int32 limit = PG_GETARG_INT32(1);
  bool isnull;
  Datum salary;

  salary = GetAttributeByName(t, "salary", &isnull);
  if (isnull)
    PG_RETURN_BOOL(false);
  /* Alternatively, we might prefer to do PG_RETURN_NULL() for null salary. */
  PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```
GetAttributeByName is the PostgreSQL system function that returns attributes out of the specified row. It has three arguments: the argument of type HeapTupleHeader passed into the function, the name of the desired attribute, and a return parameter that tells whether the attribute is null. GetAttributeByName returns a Datum value that you can convert to the proper data type by using the appropriate DatumGetXXX() macro. Note that the return value is meaningless if the null flag is set; always check the null flag before trying to do anything with the result.

There is also GetAttributeByNum, which selects the target attribute by column number instead of name.

The following command declares the function c_overpaid in SQL:

```sql
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
AS 'DIRECTORY/funcs', 'c_overpaid'
LANGUAGE C STRICT;
```

Notice we have used STRICT so that we did not have to check whether the input arguments were NULL.

### 34.9.9. Returning Rows (Composite Types)

To return a row or composite-type value from a C-language function, you can use a special API that provides macros and functions to hide most of the complexity of building composite data types. To use this API, the source file must include:

```c
#include "funcapi.h"
```

There are two ways you can build a composite data value (henceforth a “tuple”): you can build it from an array of Datum values, or from an array of C strings that can be passed to the input conversion functions of the tuple’s column data types. In either case, you first need to obtain or construct a TupleDesc descriptor for the tuple structure. When working with Datums, you pass the TupleDesc to BlessTupleDesc, and then call heap_form_tuple for each row. When working with C strings, you pass the TupleDesc to TupleDescGetAttInMetadata, and then call BuildTupleFromCStrings for each row. In the case of a function returning a set of tuples, the setup steps can all be done once during the first call of the function.

Several helper functions are available for setting up the needed TupleDesc. The recommended way to do this in most functions returning composite values is to call:

```c
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
    Oid *resultTypeId,
    TupleDesc *resultTupleDesc)
```

passing the same fcinfo struct passed to the calling function itself. (This of course requires that you use the version-1 calling conventions.) resultTypeId can be specified as NULL or as the address of a local variable to receive the function’s result type OID. resultTupleDesc should be the address of a local TupleDesc variable. Check that the result is TYPEFUNC_COMPOSITE; if so, resultTupleDesc has been filled with the needed TupleDesc. (If it is not, you can report an error along the lines of “function returning record called in context that cannot accept type record”.)

**Tip:** get_call_result_type can resolve the actual type of a polymorphic function result; so it is useful in functions that return scalar polymorphic results, not only functions that return composites. The resultTypeId output is primarily useful for functions returning polymorphic scalars.
Note: get_call_result_type has a sibling get_expr_result_type, which can be used to resolve the expected output type for a function call represented by an expression tree. This can be used when trying to determine the result type from outside the function itself. There is also get_func_result_type, which can be used when only the function's OID is available. However these functions are not able to deal with functions declared to return record, and get_func_result_type cannot resolve polymorphic types, so you should preferentially use get_call_result_type.

Older, now-deprecated functions for obtaining TupleDescs are:

TupleDesc RelationNameGetTupleDesc(const char *relname)

to get a TupleDesc for the row type of a named relation, and:

TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)

to get a TupleDesc based on a type OID. This can be used to get a TupleDesc for a base or composite type. It will not work for a function that returns record, however, and it cannot resolve polymorphic types.

Once you have a TupleDesc, call:

TupleDesc BlessTupleDesc(TupleDesc tupdesc)

if you plan to work with Datums, or:

AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)

if you plan to work with C strings. If you are writing a function returning set, you can save the results of these functions in the FuncCallContext structure — use the tuple_desc or attinmeta field respectively.

When working with Datums, use:

HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)

to build a HeapTuple given user data in Datum form.

When working with C strings, use:

HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)

to build a HeapTuple given user data in C string form. values is an array of C strings, one for each attribute of the return row. Each C string should be in the form expected by the input function of the attribute data type. In order to return a null value for one of the attributes, the corresponding pointer in the values array should be set to NULL. This function will need to be called again for each row you return.

Once you have built a tuple to return from your function, it must be converted into a Datum. Use:

HeapTupleGetDatum(HeapTuple tuple)

to convert a HeapTuple into a valid Datum. This Datum can be returned directly if you intend to return just a single row, or it can be used as the current return value in a set-returning function.

An example appears in the next section.
34.9.10. Returning Sets

There is also a special API that provides support for returning sets (multiple rows) from a C-language function. A set-returning function must follow the version-1 calling conventions. Also, source files must include `funcapi.h`, as above.

A set-returning function (SRF) is called once for each item it returns. The SRF must therefore save enough state to remember what it was doing and return the next item on each call. The structure `FuncCallContext` is provided to help control this process. Within a function, `fcinfo->flinfo->fn_extra` is used to hold a pointer to `FuncCallContext` across calls.

typedef struct
{
    /*
     * Number of times we’ve been called before
     * call_cntr is initialized to 0 for you by SRF_FIRSTCALL_INIT(), and
     * incremented for you every time SRF_RETURN_NEXT() is called.
     */
    uint32 call_cntr;

    /*
     * OPTIONAL maximum number of calls
     * max_calls is here for convenience only and setting it is optional.
     * If not set, you must provide alternative means to know when the
     * function is done.
     */
    uint32 max_calls;

    /*
     * OPTIONAL pointer to result slot
     * This is obsolete and only present for backwards compatibility, viz,
     * user-defined SRFs that use the deprecated TupleDescGetSlot().
     */
    TupleTableSlot *slot;

    /*
     * OPTIONAL pointer to miscellaneous user-provided context information
     * user_fctx is for use as a pointer to your own data to retain
     * arbitrary context information between calls of your function.
     */
    void *user_fctx;

    /*
     * OPTIONAL pointer to struct containing attribute type input metadata
     * attinmeta is for use when returning tuples (i.e., composite data types)
     * and is not used when returning base data types. It is only needed
     * if you intend to use BuildTupleFromCStrings() to create the return
     * tuple.
     */
    AttInMetadata *attinmeta;

    /*
     * memory context used for structures that must live for multiple calls
     */
} FuncCallContext;
Chapter 34. Extending SQL

* multi_call_memory_ctx is set by SRF_FIRSTCALL_INIT() for you, and used
  by SRF_RETURN_DONE() for cleanup. It is the most appropriate memory
  context for any memory that is to be reused across multiple calls
  of the SRF.
*/
MemoryContext multi_call_memory_ctx;

/*********************************
* OPTIONAL pointer to struct containing tuple description
* tuple_desc is for use when returning tuples (i.e. composite data types)
* and is only needed if you are going to build the tuples with
* heap_form_tuple() rather than with BuildTupleFromCStrings(). Note that
* the TupleDesc pointer stored here should usually have been run through
* BlessTupleDesc() first.
*/
TupleDesc tuple_desc;
}

An SRF uses several functions and macros that automatically manipulate the FuncCallContext structure (and expect to find it via fn_extra). Use:

SRF_IS_FIRSTCALL()

to determine if your function is being called for the first or a subsequent time. On the first call (only) use:

SRF_FIRSTCALL_INIT()

to initialize the FuncCallContext. On every function call, including the first, use:

SRF_PERCALL_SETUP()

to properly set up for using the FuncCallContext and clearing any previously returned data left over from the previous pass.

If your function has data to return, use:

SRF_RETURN_NEXT(funcctx, result)

to return it to the caller. (result must be of type Datum, either a single value or a tuple prepared as described above.) Finally, when your function is finished returning data, use:

SRF_RETURN_DONE(funcctx)

to clean up and end the SRF.

The memory context that is current when the SRF is called is a transient context that will be cleared between calls. This means that you do not need to call pfree on everything you allocated using palloc; it will go away anyway. However, if you want to allocate any data structures to live across calls, you need to put them somewhere else. The memory context referenced by multi_call_memory_ctx is a suitable location for any data that needs to survive until the SRF is finished running. In most cases, this means that you should switch into multi_call_memory_ctx while doing the first-call setup.

A complete pseudo-code example looks like the following:
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    Datum result;
    MemoryContext oldcontext;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* One-time setup code appears here: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }
    /* Each-time setup code appears here: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* this is just one way we might test whether we are done: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Here we want to return another item: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* Here we are done returning items and just need to clean up: */
        user code
        SRF_RETURN_DONE(funcctx);
    }
}

A complete example of a simple SRF returning a composite type looks like:

PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    int call_cntr;
    int max_calls;
    TupleDesc tupdesc;
    AttInMetadata *attinmeta;

    /* stuff done only on the first call of the function */
if (SRF_IS_FIRSTCALL())
{
    MemoryContext oldcontext;

    /* create a function context for cross-call persistence */
    funcctx = SRF_FIRSTCALL_INIT();

    /* switch to memory context appropriate for multiple function calls */
    oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

    /* total number of tuples to be returned */
    funcctx->max_calls = PG_GETARG_UINT32(0);

    /* Build a tuple descriptor for our result type */
    if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
        ereport(ERROR,
                {errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                 errmsg("function returning record called in context "
                             "that cannot accept type record");});

    /* generate attribute metadata needed later to produce tuples from raw
     * C strings
     */
    attinmeta = TupleDescGetAttInMetadata(tupdesc);
    funcctx->attinmeta = attinmeta;

    MemoryContextSwitchTo(oldcontext);
}

/* stuff done on every call of the function */
funcctx = SRF_PERCALL_SETUP();

call_cntr = funcctx->call_cntr;
max_calls = funcctx->max_calls;
attinmeta = funcctx->attinmeta;

if (call_cntr < max_calls)    /* do when there is more left to send */
{
    char **values;
    HeapTuple tuple;
    Datum result;

    /* Prepare a values array for building the returned tuple.
     * This should be an array of C strings which will
     * be processed later by the type input functions.
     */
    values = (char **) palloc(3 * sizeof(char *));
    values[0] = (char *) palloc(16 * sizeof(char));
    values[1] = (char *) palloc(16 * sizeof(char));
    values[2] = (char *) palloc(16 * sizeof(char));

    snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
    snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
    snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));
/* build a tuple */
tuple = BuildTupleFromCStrings(attinmeta, values);

/* make the tuple into a datum */
result = HeapTupleGetDatum(tuple);

/* clean up (this is not really necessary) */
pfree(values[0]);
pfree(values[1]);
pfree(values[2]);
pfree(values);

SRF_RETURN_NEXT(funcctx, result);
}
else /* do when there is no more left */
{
    SRF_RETURN_DONE(funcctx);
}

One way to declare this function in SQL is:

CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;

A different way is to use OUT parameters:

CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;

Notice that in this method the output type of the function is formally an anonymous record type.
The directory contrib/tablefunc in the source distribution contains more examples of set-returning functions.

34.9.11. Polymorphic Arguments and Return Types

C-language functions can be declared to accept and return the polymorphic types anyelement, anyarray, anynonarray, and anyenum. See Section 34.2.5 for a more detailed explanation of polymorphic functions. When function arguments or return types are defined as polymorphic types, the function author cannot know in advance what data type it will be called with, or need to return. There are two routines provided in fmgr.h to allow a version-1 C function to discover the actual data types of its arguments and the type it is expected to return. The routines are called get_fn_expr_rettype(FmgrInfo *flinfo) and get_fn_expr_argtype(FmgrInfo *flinfo, int argnum). They return the result or argument type OID, or InvalidOid if the information is not available. The structure flinfo is normally accessed as fcline->flinfo. The parameter argnum is zero based. get_call_result_type can also be used as an alternative to get_fn_expr_rettype.
For example, suppose we want to write a function to accept a single element of any type, and return a one-dimensional array of that type:

```c
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType *result;
    Oid    element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum  element;
    bool   isnull;
    int16  typlen;
    bool   typbyval;
    char   typalign;
    int    ndims;
    int    dims[MAXDIM];
    int    lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* get the provided element, being careful in case it’s NULL */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* we have one dimension */
    ndims = 1;
    /* and one element */
    dims[0] = 1;
    /* and lower bound is 1 */
    lbs[0] = 1;

    /* get required info about the element type */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* now build the array */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

The following command declares the function `make_array` in SQL:

```sql
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
AS 'DIRECTORY/funcs', 'make_array'
LANGUAGE C IMMUTABLE;
```
34.9.12. Shared Memory and LWLocks

Add-ins can reserve LWLocks and an allocation of shared memory on server startup. The add-in’s shared library must be preloaded by specifying it in shared_preload_libraries. Shared memory is reserved by calling:

```c
void RequestAddinShmemSpace(int size)
```

from your \_PG\_init function.

LWLocks are reserved by calling:

```c
void RequestAddinLWLocks(int n)
```

from \_PG\_init.

To avoid possible race-conditions, each backend should use the LWLock `AddinShmemInitLock` when connecting to and initializing its allocation of shared memory, as shown here:

```c
static mystruct *ptr = NULL;
if (!ptr)
{
  bool found;
  LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
  ptr = ShmemInitStruct("my struct name", size, &found);
  if (!ptr)
    elog(ERROR, "out of shared memory");
  if (!found)
    {
      initialize contents of shmem area;
      acquire any requested LWLocks using:
      ptr->mylockid = LWLockAssign();
    }
  LWLockRelease(AddinShmemInitLock);
}
```

34.10. User-Defined Aggregates

Aggregate functions in PostgreSQL are expressed in terms of state values and state transition functions. That is, an aggregate operates using a state value that is updated as each successive input row is processed. To define a new aggregate function, one selects a data type for the state value, an initial value for the state, and a state transition function. The state transition function is just an ordinary function that could also be used outside the context of the aggregate. A final function can also be specified, in case the desired result of the aggregate is different from the data that needs to be kept in the running state value.

Thus, in addition to the argument and result data types seen by a user of the aggregate, there is an internal state-value data type that might be different from both the argument and result types.

If we define an aggregate that does not use a final function, we have an aggregate that computes a running function of the column values from each row. `sum` is an example of this kind of aggregate. `sum` starts at zero and always adds the current row’s value to its running total. For example, if we
want to make a `sum` aggregate to work on a data type for complex numbers, we only need the addition function for that data type. The aggregate definition would be:

```sql
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

SELECT sum(a) FROM test_complex;

<table>
<thead>
<tr>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>(34, 53.9)</td>
</tr>
</tbody>
</table>

(Notice that we are relying on function overloading: there is more than one aggregate named `sum`, but PostgreSQL can figure out which kind of sum applies to a column of type `complex`.)

The above definition of `sum` will return zero (the initial state condition) if there are no nonnull input values. Perhaps we want to return null in that case instead — the SQL standard expects `sum` to behave that way. We can do this simply by omitting the `initcond` phrase, so that the initial state condition is null. Ordinarily this would mean that the `sfunc` would need to check for a null state-condition input. But for `sum` and some other simple aggregates like `max` and `min`, it is sufficient to insert the first nonnull input value into the state variable and then start applying the transition function at the second nonnull input value. PostgreSQL will do that automatically if the initial condition is null and the transition function is marked “strict” (i.e., not to be called for null inputs).

Another bit of default behavior for a “strict” transition function is that the previous state value is retained unchanged whenever a null input value is encountered. Thus, null values are ignored. If you need some other behavior for null inputs, do not declare your transition function as strict; instead code it to test for null inputs and do whatever is needed.

`avg` (average) is a more complex example of an aggregate. It requires two pieces of running state: the sum of the inputs and the count of the number of inputs. The final result is obtained by dividing these quantities. Average is typically implemented by using an array as the state value. For example, the built-in implementation of `avg(float8)` looks like:

```sql
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0}'
);
```

(`float8_accum` requires a three-element array, not just two elements, because it accumulates the sum of squares as well as the sum and count of the inputs. This is so that it can be used for some other aggregates besides `avg`.)

Aggregate functions can use polymorphic state transition functions or final functions, so that the same functions can be used to implement multiple aggregates. See Section 34.2.5 for an explanation of polymorphic functions. Going a step further, the aggregate function itself can be specified with polymorphic input type(s) and state type, allowing a single aggregate definition to serve for multiple input data types. Here is an example of a polymorphic aggregate:

```sql
CREATE AGGREGATE array_accum (anyelement)
```
Chapter 34. Extending SQL

```c
(sfunc = array_append,
 stype = anyarray,
 initcond = '{}');
```

Here, the actual state type for any aggregate call is the array type having the actual input type as elements.

Here’s the output using two different actual data types as arguments:

```sql
SELECT attrelid::regclass, array_accum(attname)
FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
GROUP BY attrelid;
```

```
+---------------------------------------+-----------------------------+
<table>
<thead>
<tr>
<th>attrelid</th>
<th>array_accum</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_tablespace</td>
<td>{spcname, spcowner, spclocation, spcacl}</td>
</tr>
</tbody>
</table>
+--------------------------+-----------------------------+
(1 row)
```

```sql
SELECT attrelid::regclass, array_accum(atttypid)
FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
GROUP BY attrelid;
```

```
+---------------------------------------+-----------------------------+
<table>
<thead>
<tr>
<th>attrelid</th>
<th>array_accum</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_tablespace</td>
<td>{19, 26, 25, 1034}</td>
</tr>
</tbody>
</table>
+--------------------------+-----------------------------+
(1 row)
```

A function written in C can detect that it is being called as an aggregate transition or final function by seeing if it was passed an `AggState` node as the function call “context”, for example by:

```c
if (fcinfo->context && IsA(fcinfo->context, AggState))
```

One reason for checking this is that when it is true, the first input must be a temporary transition value and can therefore safely be modified in-place rather than allocating a new copy. (This is the only case where it is safe for a function to modify a pass-by-reference input.) See `int8inc()` for an example.

For further details see the `CREATE AGGREGATE` command.

34.11. User-Defined Types

As described in Section 34.2, PostgreSQL can be extended to support new data types. This section describes how to define new base types, which are data types defined below the level of the SQL language. Creating a new base type requires implementing functions to operate on the type in a low-level language, usually C.

The examples in this section can be found in `complex.sql` and `complex.c` in the `src/tutorial` directory of the source distribution. See the `README` file in that directory for instructions about running the examples.

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in
memory. The input function takes a null-terminated character string as its argument and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type as argument and returns a null-terminated character string. If we want to do anything more with the type than merely store it, we must provide additional functions to implement whatever operations we’d like to have for the type.

Suppose we want to define a type complex that represents complex numbers. A natural way to represent a complex number in memory would be the following C structure:

```c
typedef struct Complex {
  double x;
  double y;
} Complex;
```

We will need to make this a pass-by-reference type, since it’s too large to fit into a single Datum value.

As the external string representation of the type, we choose a string of the form \((x, y)\).

The input and output functions are usually not hard to write, especially the output function. But when defining the external string representation of the type, remember that you must eventually write a complete and robust parser for that representation as your input function. For instance:

```c
PG_FUNCTION_INFO_V1(complex_in);

Datum complex_in(PG_FUNCTION_ARGS)
{
  char *str = PG_GETARG_CSTRING(0);
  double x,
          y;
  Complex *result;

  if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
    ereport(ERROR,
            (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
            errmsg("invalid input syntax for complex: \"%s\"",
                    str)));

  result = (Complex *) palloc(sizeof(Complex));
  result->x = x;
  result->y = y;
  PG_RETURN_POINTER(result);
}
```

The output function can simply be:

```c
PG_FUNCTION_INFO_V1(complex_out);

Datum complex_out(PG_FUNCTION_ARGS)
{
  Complex *complex = (Complex *) PG_GETARG_POINTER(0);
  char *result;

  result = (char *) palloc(100);
  snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
  PG_RETURN_CSTRING(result);
}
```
You should be careful to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in. This is a particularly common problem when floating-point numbers are involved.

Optionally, a user-defined type can provide binary input and output routines. Binary I/O is normally faster but less portable than textual I/O. As with textual I/O, it is up to you to define exactly what the external binary representation is. Most of the built-in data types try to provide a machine-independent binary representation. For complex, we will piggy-back on the binary I/O converters for type float8:

```c
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex      *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex      *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Once we have written the I/O functions and compiled them into a shared library, we can define the complex type in SQL. First we declare it as a shell type:

```sql
CREATE TYPE complex;
```

This serves as a placeholder that allows us to reference the type while defining its I/O functions. Now we can define the I/O functions:

```sql
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
```
Chapter 34. Extending SQL

```
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
RETURNS complex
AS 'filename'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
RETURNS bytea
AS 'filename'
LANGUAGE C IMMUTABLE STRICT;
```

Finally, we can provide the full definition of the data type:

```
CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out,
    receive = complex_recv,
    send = complex_send,
    alignment = double
);
```

When you define a new base type, PostgreSQL automatically provides support for arrays of that type. The array type typically has the same name as the base type with the underscore character (\_) prepended.

Once the data type exists, we can declare additional functions to provide useful operations on the data type. Operators can then be defined atop the functions, and if needed, operator classes can be created to support indexing of the data type. These additional layers are discussed in following sections.

If the values of your data type vary in size (in internal form), you should make the data type TOASTable (see Section 53.2). You should do this even if the data are always too small to be compressed or stored externally, because TOAST can save space on small data too, by reducing header overhead.

To do this, the internal representation must follow the standard layout for variable-length data: the first four bytes must be a char[4] field which is never accessed directly (customarily named vl_len_\_). You must use SET_VARSIZE() to store the size of the datum in this field and VARSIZE() to retrieve it. The C functions operating on the data type must always be careful to unpack any toasted values they are handed, by using PG\_DETOAST\_DATUM. (This detail is customarily hidden by defining type-specific GETARG\_DATATYPE\_P macros.) Then, when running the CREATE TYPE command, specify the internal length as variable and select the appropriate storage option.

If the alignment is unimportant (either just for a specific function or because the data type specifies byte alignment anyways) then it’s possible to avoid some of the overhead of PG\_DETOAST\_DATUM. You can use PG\_DETOAST\_DATUM\_PACKED instead (customarily hidden by defining a GETARG\_DATATYPE\_PP macro) and using the macros VARSIZE\_ANY\_EXHDR and VARDATA\_ANY to access a potentially-packed datum. Again, the data returned by these macros is not aligned even if the data type definition specifies an alignment. If the alignment is important you must go through the regular PG\_DETOAST\_DATUM interface.

**Note:** Older code frequently declares vl_len_ as an int32 field instead of char[4]. This is OK as long as the struct definition has other fields that have at least int32 alignment. But it is dangerous to use such a struct definition when working with a potentially unaligned datum; the compiler may
take it as license to assume the datum actually is aligned, leading to core dumps on architectures
that are strict about alignment.

For further details see the description of the CREATE TYPE command.

### 34.12. User-Defined Operators

Every operator is “syntactic sugar” for a call to an underlying function that does the real work; so
you must first create the underlying function before you can create the operator. However, an operator
is not merely syntactic sugar, because it carries additional information that helps the query planner
optimize queries that use the operator. The next section will be devoted to explaining that additional
information.

PostgreSQL supports left unary, right unary, and binary operators. Operators can be overloaded; that
is, the same operator name can be used for different operators that have different numbers and types
of operands. When a query is executed, the system determines the operator to call from the number
and types of the provided operands.

Here is an example of creating an operator for adding two complex numbers. We assume we’ve
already created the definition of type complex (see Section 34.11). First we need a function that does
the work, then we can define the operator:

```sql
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Now we could execute a query like this:

```sql
SELECT (a + b) AS c FROM test_complex;
```

```
c
(5.2,6.05)
(133.42,144.95)
```

We’ve shown how to create a binary operator here. To create unary operators, just omit one of
leftarg (for left unary) or rightarg (for right unary). The procedure clause and the argument
clauses are the only required items in CREATE OPERATOR. The commutator clause shown in the
example is an optional hint to the query optimizer. Further details about commutator and other opti-
mizer hints appear in the next section.
34.13. Operator Optimization Information

A PostgreSQL operator definition can include several optional clauses that tell the system useful things about how the operator behaves. These clauses should be provided whenever appropriate, because they can make for considerable speedups in execution of queries that use the operator. But if you provide them, you must be sure that they are right! Incorrect use of an optimization clause can result in slow queries, subtly wrong output, or other Bad Things. You can always leave out an optimization clause if you are not sure about it; the only consequence is that queries might run slower than they need to.

Additional optimization clauses might be added in future versions of PostgreSQL. The ones described here are all the ones that release 8.3.23 understands.

34.13.1. COMMUTATOR

The COMMUTATOR clause, if provided, names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x, y. Notice that B is also the commutator of A. For example, operators < and > for a particular data type are usually each others’ commutators, and operator + is usually commutative with itself. But operator – is usually not commutative with anything.

The left operand type of a commutable operator is the same as the right operand type of its commutator, and vice versa. So the name of the commutator operator is all that PostgreSQL needs to be given to look up the commutator, and that’s all that needs to be provided in the COMMUTATOR clause.

It’s critical to provide commutator information for operators that will be used in indexes and join clauses, because this allows the query optimizer to “flip around” such a clause to the forms needed for different plan types. For example, consider a query with a WHERE clause like tab1.x = tab2.y, where tab1.x and tab2.y are of a user-defined type, and suppose that tab2.y is indexed. The optimizer cannot generate an index scan unless it can determine how to flip the clause around to tab2.y = tab1.x, because the index-scan machinery expects to see the indexed column on the left of the operator it is given. PostgreSQL will not simply assume that this is a valid transformation — the creator of the = operator must specify that it is valid, by marking the operator with commutator information.

When you are defining a self-commutative operator, you just do it. When you are defining a pair of commutative operators, things are a little trickier: how can the first one to be defined refer to the other one, which you haven’t defined yet? There are two solutions to this problem:

• One way is to omit the COMMUTATOR clause in the first operator that you define, and then provide one in the second operator’s definition. Since PostgreSQL knows that commutative operators come in pairs, when it sees the second definition it will automatically go back and fill in the missing COMMUTATOR clause in the first definition.

• The other, more straightforward way is just to include COMMUTATOR clauses in both definitions. When PostgreSQL processes the first definition and realizes that COMMUTATOR refers to a nonexistent operator, the system will make a dummy entry for that operator in the system catalog. This dummy entry will have valid data only for the operator name, left and right operand types, and result type, since that’s all that PostgreSQL can deduce at this point. The first operator’s catalog entry will link to this dummy entry. Later, when you define the second operator, the system updates the dummy entry with the additional information from the second definition. If you try to use the dummy operator before it’s been filled in, you’ll just get an error message.
34.13.2. NEGATOR

The **NEGATOR** clause, if provided, names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return Boolean results and (x A y) equals NOT (x B y) for all possible inputs x, y. Notice that B is also the negator of A. For example, < and >= are a negator pair for most data types. An operator can never validly be its own negator.

Unlike commutators, a pair of unary operators could validly be marked as each others’ negators; that would mean (A x) equals NOT (B x) for all x, or the equivalent for right unary operators.

An operator’s negator must have the same left and/or right operand types as the operator to be defined, so just as with **COMMUTATOR**, only the operator name need be given in the **NEGATOR** clause.

Providing a negator is very helpful to the query optimizer since it allows expressions like NOT (x = y) to be simplified into x <> y. This comes up more often than you might think, because NOT operations can be inserted as a consequence of other rearrangements.

Pairs of negator operators can be defined using the same methods explained above for commutator pairs.

34.13.3. RESTRICT

The **RESTRICT** clause, if provided, names a restriction selectivity estimation function for the operator. (Note that this is a function name, not an operator name.) **RESTRICT** clauses only make sense for binary operators that return boolean. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a **WHERE**-clause condition of the form:

```
column OP constant
```

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by **WHERE** clauses that have this form. (What happens if the constant is on the left, you might be wondering? Well, that’s one of the things that **COMMUTATOR** is for...)

Writing new restriction selectivity estimation functions is far beyond the scope of this chapter, but fortunately you can usually just use one of the system’s standard estimators for many of your own operators. These are the standard restriction estimators:

```
eqsel for =
eqsel for <>
scalarltssel for < or <=
scalargtssel for > or >-
```

It might seem a little odd that these are the categories, but they make sense if you think about it. = will typically accept only a small fraction of the rows in a table; <> will typically reject only a small fraction. < will accept a fraction that depends on where the given constant falls in the range of values for that table column (which, it just so happens, is information collected by **ANALYZE** and made available to the selectivity estimator). <= will accept a slightly larger fraction than < for the same comparison constant, but they’re close enough to not be worth distinguishing, especially since we’re not likely to do better than a rough guess anyhow. Similar remarks apply to > and >-.

You can frequently get away with using either **eqsel** or **neqsel** for operators that have very high or very low selectivity, even if they aren’t really equality or inequality. For example, the approximate-equality geometric operators use **eqsel** on the assumption that they’ll usually only match a small fraction of the entries in a table.
You can use `scalarltsel` and `scalargtssel` for comparisons on data types that have some sensible means of being converted into numeric scalars for range comparisons. If possible, add the data type to those understood by the function `convert_to_scalar()` in `src/backend/utils/adt/selfuncs.c`. (Eventually, this function should be replaced by per-data-type functions identified through a column of the pg_type system catalog; but that hasn’t happened yet.) If you do not do this, things will still work, but the optimizer’s estimates won’t be as good as they could be.

There are additional selectivity estimation functions designed for geometric operators in `src/backend/utils/adt/geo_selfuncs.c`: `areasel`, `positionsel`, and `contsel`. At this writing these are just stubs, but you might want to use them (or even better, improve them) anyway.

### 34.13.4. JOIN

The `JOIN` clause, if provided, names a join selectivity estimation function for the operator. (Note that this is a function name, not an operator name.) `JOIN` clauses only make sense for binary operators that return boolean. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a `WHERE`-clause condition of the form:

```
  table1.column1 OP table2.column2
```

for the current operator. As with the `RESTRIC`T clause, this helps the optimizer very substantially by letting it figure out which of several possible join sequences is likely to take the least work.

As before, this chapter will make no attempt to explain how to write a join selectivity estimator function, but will just suggest that you use one of the standard estimators if one is applicable:

- `eqjoinsel` for `=`
- `neqjoinsel` for `<>
- `scalarltjoinsel` for `< or `<=`
- `scalargtjoinsel` for `>` or `>=`
- `areajoinsel` for 2D area-based comparisons
- `positionjoinsel` for 2D position-based comparisons
- `contjoinsel` for 2D containment-based comparisons

### 34.13.5. HASHES

The `HASHES` clause, if present, tells the system that it is permissible to use the hash join method for a join based on this operator. `HASHES` only makes sense for a binary operator that returns boolean, and in practice the operator must represent equality for some data type or pair of data types.

The assumption underlying hash join is that the join operator can only return true for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join will never compare them at all, implicitly assuming that the result of the join operator must be false. So it never makes sense to specify `HASHES` for operators that do not represent some form of equality. In most cases it is only practical to support hashing for operators that take the same data type on both sides. However, sometimes it is possible to design compatible hash functions for two or more data types; that is, functions that will generate the same hash codes for “equal” values, even though the values have different representations. For example, it’s fairly simple to arrange this property when hashing integers of different widths.
To be marked HASHES, the join operator must appear in a hash index operator family. This is not enforced when you create the operator, since of course the referencing operator family couldn’t exist yet. But attempts to use the operator in hash joins will fail at run time if no such operator family exists. The system needs the operator family to find the data-type-specific hash function(s) for the operator’s input data type(s). Of course, you must also create suitable hash functions before you can create the operator family.

Care should be exercised when preparing a hash function, because there are machine-dependent ways in which it might fail to do the right thing. For example, if your data type is a structure in which there might be uninteresting pad bits, you cannot simply pass the whole structure to hash_any. (Unless you write your other operators and functions to ensure that the unused bits are always zero, which is the recommended strategy.) Another example is that on machines that meet the IEEE floating-point standard, negative zero and positive zero are different values (different bit patterns) but they are defined to compare equal. If a float value might contain negative zero then extra steps are needed to ensure it generates the same hash value as positive zero.

A hash-joinable operator must have a commutator (itself if the two operand data types are the same, or a related equality operator if they are different) that appears in the same operator family. If this is not the case, planner errors might occur when the operator is used. Also, it is a good idea (but not strictly required) for a hash operator family that supports multiple data types to provide equality operators for every combination of the data types; this allows better optimization.

**Note:** The function underlying a hash-joinable operator must be marked immutable or stable. If it is volatile, the system will never attempt to use the operator for a hash join.

**Note:** If a hash-joinable operator has an underlying function that is marked strict, the function must also be complete: that is, it should return true or false, never null, for any two nonnull inputs. If this rule is not followed, hash-optimization of IN operations might generate wrong results. (Specifically, IN might return false where the correct answer according to the standard would be null, or it might yield an error complaining that it wasn’t prepared for a null result.)

### 34.13.6. MERGES

The MERGES clause, if present, tells the system that it is permissible to use the merge-join method for a join based on this operator. MERGES only makes sense for a binary operator that returns boolean, and in practice the operator must represent equality for some data type or pair of data types.

Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the “same place” in the sort order. In practice this means that the join operator must behave like equality. But it is possible to merge-join two distinct data types so long as they are logically compatible. For example, the smallint-versus-integer equality operator is merge-joinable. We only need sorting operators that will bring both data types into a logically compatible sequence.

To be marked MERGES, the join operator must appear as an equality member of a btree index operator family. This is not enforced when you create the operator, since of course the referencing operator family couldn’t exist yet. But the operator will not actually be used for merge joins unless a matching operator family can be found. The MERGES flag thus acts as a hint to the planner that it’s worth looking for a matching operator family.
A merge-joinable operator must have a commutator (itself if the two operand data types are the same, or a related equality operator if they are different) that appears in the same operator family. If this is not the case, planner errors might occur when the operator is used. Also, it is a good idea (but not strictly required) for a btree operator family that supports multiple data types to provide equality operators for every combination of the data types; this allows better optimization.

**Note:** The function underlying a merge-joinable operator must be marked immutable or stable. If it is volatile, the system will never attempt to use the operator for a merge join.

### 34.14. Interfacing Extensions To Indexes

The procedures described thus far let you define new types, new functions, and new operators. However, we cannot yet define an index on a column of a new data type. To do this, we must define an operator class for the new data type. Later in this section, we will illustrate this concept in an example: a new operator class for the B-tree index method that stores and sorts complex numbers in ascending absolute value order.

Operator classes can be grouped into operator families to show the relationships between semantically compatible classes. When only a single data type is involved, an operator class is sufficient, so we’ll focus on that case first and then return to operator families.

#### 34.14.1. Index Methods and Operator Classes

The `pg_am` table contains one row for every index method (internally known as access method). Support for regular access to tables is built into PostgreSQL, but all index methods are described in `pg_am`. It is possible to add a new index method by defining the required interface routines and then creating a row in `pg_am` — but that is beyond the scope of this chapter (see Chapter 50).

The routines for an index method do not directly know anything about the data types that the index method will operate on. Instead, an operator class identifies the set of operations that the index method needs to use to work with a particular data type. Operator classes are so called because one thing they specify is the set of WHERE-clause operators that can be used with an index (i.e., can be converted into an index-scan qualification). An operator class can also specify some support procedures that are needed by the internal operations of the index method, but do not directly correspond to any WHERE-clause operator that can be used with the index.

It is possible to define multiple operator classes for the same data type and index method. By doing this, multiple sets of indexing semantics can be defined for a single data type. For example, a B-tree index requires a sort ordering to be defined for each data type it works on. It might be useful for a complex-number data type to have one B-tree operator class that sorts the data by complex absolute value, another that sorts by real part, and so on. Typically, one of the operator classes will be deemed most commonly useful and will be marked as the default operator class for that data type and index method.

The same operator class name can be used for several different index methods (for example, both B-tree and hash index methods have operator classes named `int4_ops`), but each such class is an independent entity and must be defined separately.
34.14.2. Index Method Strategies

The operators associated with an operator class are identified by “strategy numbers”, which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like “less than” and “greater than or equal to” are interesting with respect to a B-tree. Because PostgreSQL allows the user to define operators, PostgreSQL cannot look at the name of an operator (e.g., < or >=) and tell what kind of comparison it is. Instead, the index method defines a set of “strategies”, which can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics.

The B-tree index method defines five strategies, shown in Table 34-2.

Table 34-2. B-tree Strategies

<table>
<thead>
<tr>
<th>Operation</th>
<th>Strategy Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td>1</td>
</tr>
<tr>
<td>less than or equal</td>
<td>2</td>
</tr>
<tr>
<td>equal</td>
<td>3</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>4</td>
</tr>
<tr>
<td>greater than</td>
<td>5</td>
</tr>
</tbody>
</table>

Hash indexes support only equality comparisons, and so they use only one strategy, shown in Table 34-3.

Table 34-3. Hash Strategies

<table>
<thead>
<tr>
<th>Operation</th>
<th>Strategy Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal</td>
<td>1</td>
</tr>
</tbody>
</table>

GiST indexes are more flexible: they do not have a fixed set of strategies at all. Instead, the “consistency” support routine of each particular GiST operator class interprets the strategy numbers however it likes. As an example, several of the built-in GiST index operator classes index two-dimensional geometric objects, providing the “R-tree” strategies shown in Table 34-4. Four of these are true two-dimensional tests (overlaps, same, contains, contained by); four of them consider only the X direction; and the other four provide the same tests in the Y direction.

Table 34-4. GiST Two-Dimensional “R-tree” Strategies

<table>
<thead>
<tr>
<th>Operation</th>
<th>Strategy Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>strictly left of</td>
<td>1</td>
</tr>
<tr>
<td>does not extend to right of</td>
<td>2</td>
</tr>
<tr>
<td>overlaps</td>
<td>3</td>
</tr>
<tr>
<td>does not extend to left of</td>
<td>4</td>
</tr>
<tr>
<td>strictly right of</td>
<td>5</td>
</tr>
<tr>
<td>same</td>
<td>6</td>
</tr>
<tr>
<td>contains</td>
<td>7</td>
</tr>
<tr>
<td>contained by</td>
<td>8</td>
</tr>
<tr>
<td>does not extend above</td>
<td>9</td>
</tr>
</tbody>
</table>
GIN indexes are similar to GiST indexes in flexibility: they don’t have a fixed set of strategies. Instead the support routines of each operator class interpret the strategy numbers according to the operator class’s definition. As an example, the strategy numbers used by the built-in operator classes for arrays are shown in Table 34-5.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Strategy Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>strictly below</td>
<td>10</td>
</tr>
<tr>
<td>strictly above</td>
<td>11</td>
</tr>
<tr>
<td>does not extend below</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 34-5. GIN Array Strategies

Notice that all strategy operators return Boolean values. In practice, all operators defined as index method strategies must return type `boolean`, since they must appear at the top level of a `WHERE` clause to be used with an index.

### 34.14.3. Index Method Support Routines

Strategies aren’t usually enough information for the system to figure out how to use an index. In practice, the index methods require additional support routines in order to work. For example, the B-tree index method must be able to compare two keys and determine whether one is greater than, equal to, or less than the other. Similarly, the hash index method must be able to compute hash codes for key values. These operations do not correspond to operators used in qualifications in SQL commands; they are administrative routines used by the index methods, internally.

Just as with strategies, the operator class identifies which specific functions should play each of these roles for a given data type and semantic interpretation. The index method defines the set of functions it needs, and the operator class identifies the correct functions to use by assigning them to the “support function numbers” specified by the index method.

B-trees require a single support function, shown in Table 34-6.

<table>
<thead>
<tr>
<th>Function</th>
<th>Support Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second</td>
<td>1</td>
</tr>
</tbody>
</table>

Hash indexes likewise require one support function, shown in Table 34-7.

Table 34-7. Hash Support Functions
GiST indexes require seven support functions, shown in Table 34-8.

### Table 34-8. GiST Support Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Support Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>consistent - determine whether key satisfies the query qualifier</td>
<td>1</td>
</tr>
<tr>
<td>union - compute union of a set of keys</td>
<td>2</td>
</tr>
<tr>
<td>compress - compute a compressed representation of a key or value to be indexed</td>
<td>3</td>
</tr>
<tr>
<td>decompress - compute a decompressed representation of a compressed key</td>
<td>4</td>
</tr>
<tr>
<td>penalty - compute penalty for inserting new key into subtree with given subtree’s key</td>
<td>5</td>
</tr>
<tr>
<td>picksplit - determine which entries of a page are to be moved to the new page and compute the union keys for resulting pages</td>
<td>6</td>
</tr>
<tr>
<td>equal - compare two keys and return true if they are equal</td>
<td>7</td>
</tr>
</tbody>
</table>

GIN indexes require four support functions, shown in Table 34-9.

### Table 34-9. GIN Support Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Support Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare - compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second</td>
<td>1</td>
</tr>
<tr>
<td>extractValue - extract keys from a value to be indexed</td>
<td>2</td>
</tr>
<tr>
<td>extractQuery - extract keys from a query condition</td>
<td>3</td>
</tr>
<tr>
<td>consistent - determine whether value matches query condition</td>
<td>4</td>
</tr>
</tbody>
</table>

Unlike strategy operators, support functions return whichever data type the particular index method expects; for example in the case of the comparison function for B-trees, a signed integer. The number and types of the arguments to each support function are likewise dependent on the index method. For B-tree and hash the support functions take the same input data types as do the operators included in the operator class, but this is not the case for most GIN and GiST support functions.

### 34.14.4. An Example

Now that we have seen the ideas, here is the promised example of creating a new operator
class. (You can find a working copy of this example in src/tutorial/complex.c and src/tutorial/complex.sql in the source distribution.) The operator class encapsulates operators that sort complex numbers in absolute value order, so we choose the name complex_abs_ops. First, we need a set of operators. The procedure for defining operators was discussed in Section 34.12. For an operator class on B-trees, the operators we require are:

- absolute-value less-than (strategy 1)
- absolute-value less-than-or-equal (strategy 2)
- absolute-value equal (strategy 3)
- absolute-value greater-than-or-equal (strategy 4)
- absolute-value greater-than (strategy 5)

The least error-prone way to define a related set of comparison operators is to write the B-tree comparison support function first, and then write the other functions as one-line wrappers around the support function. This reduces the odds of getting inconsistent results for corner cases. Following this approach, we first write:

```c
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)
static int complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double amag = Mag(a), bmag = Mag(b);
    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

Now the less-than function looks like:

```c
PG_FUNCTION_INFO_V1(complex_abs_lt);
Datum complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex *a = (Complex *) PG_GETARG_POINTER(0);
    Complex *b = (Complex *) PG_GETARG_POINTER(1);
    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

The other four functions differ only in how they compare the internal function’s result to zero.

Next we declare the functions and the operators based on the functions to SQL:

```sql
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
AS 'filename', 'complex_abs_lt'
LANGUAGE C IMMUTABLE STRICT;
CREATE OPERATOR < (
    leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
    ....
)
commutator = >, negator = >=,
restrict = scalarltssel, join = scalarltojinsel
);

It is important to specify the correct commutator and negator operators, as well as suitable restriction
and join selectivity functions, otherwise the optimizer will be unable to make effective use of the
index. Note that the less-than, equal, and greater-than cases should use different selectivity functions.

Other things worth noting are happening here:

• There can only be one operator named, say, = and taking type complex for both operands. In this
case we don’t have any other operator = for complex, but if we were building a practical data
type we’d probably want = to be the ordinary equality operation for complex numbers (and not
the equality of the absolute values). In that case, we’d need to use some other operator name for
complex_abs_eq.

• Although PostgreSQL can cope with functions having the same SQL name as long as they have
different argument data types, C can only cope with one global function having a given name. So
we shouldn’t name the C function something simple like abs_eq. Usually it’s a good practice to
include the data type name in the C function name, so as not to conflict with functions for other
data types.

• We could have made the SQL name of the function abs_eq, relying on PostgreSQL to distinguish
it by argument data types from any other SQL function of the same name. To keep the example
simple, we make the function have the same names at the C level and SQL level.

The next step is the registration of the support routine required by B-trees. The example C code that
implements this is in the same file that contains the operator functions. This is how we declare the
function:

CREATE FUNCTION complex_abs_cmp(complex, complex)
RETURNS integer
AS 'filename'
LANGUAGE C IMMUTABLE STRICT;

Now that we have the required operators and support routine, we can finally create the operator class:

CREATE OPERATOR CLASS complex_abs_ops
DEFAULT FOR TYPE complex USING btree AS
OPERATOR 1 <,
OPERATOR 2 <=,
OPERATOR 3 =,
OPERATOR 4 >=,
OPERATOR 5 >,
FUNCTION 1 complex_abs_cmp(complex, complex);

And we’re done! It should now be possible to create and use B-tree indexes on complex columns.
We could have written the operator entries more verbosely, as in:

OPERATOR 1 < (complex, complex),
but there is no need to do so when the operators take the same data type we are defining the operator class for.

The above example assumes that you want to make this new operator class the default B-tree operator class for the `complex` data type. If you don’t, just leave out the word `DEFAULT`.

### 34.14.5. Operator Classes and Operator Families

So far we have implicitly assumed that an operator class deals with only one data type. While there certainly can be only one data type in a particular index column, it is often useful to index operations that compare an indexed column to a value of a different data type. Also, if there is use for a cross-data-type operator in connection with an operator class, it is often the case that the other data type has a related operator class of its own. It is helpful to make the connections between related classes explicit, because this can aid the planner in optimizing SQL queries (particularly for B-tree operator classes, since the planner contains a great deal of knowledge about how to work with them).

To handle these needs, PostgreSQL uses the concept of an operator family. An operator family contains one or more operator classes, and can also contain indexable operators and corresponding support functions that belong to the family as a whole but not to any single class within the family. We say that such operators and functions are “loose” within the family, as opposed to being bound into a specific class. Typically each operator class contains single-data-type operators while cross-data-type operators are loose in the family.

All the operators and functions in an operator family must have compatible semantics, where the compatibility requirements are set by the index method. You might therefore wonder why bother to single out particular subsets of the family as operator classes; and indeed for many purposes the class divisions are irrelevant and the family is the only interesting grouping. The reason for defining operator classes is that they specify how much of the family is needed to support any particular index. If there is an index using an operator class, then that operator class cannot be dropped without dropping the index — but other parts of the operator family, namely other operator classes and loose operators, could be dropped. Thus, an operator class should be specified to contain the minimum set of operators and functions that are reasonably needed to work with an index on a specific data type, and then related but non-essential operators can be added as loose members of the operator family.

As an example, PostgreSQL has a built-in B-tree operator family `integer_ops`, which includes operator classes `int8_ops`, `int4_ops`, and `int2_ops` for indexes on `bigint`, `integer`, and `smallint` columns respectively. The family also contains cross-data-type comparison operators allowing any two of these types to be compared, so that an index on one of these types can be searched using a comparison value of another type. The family could be duplicated by these definitions:

```
CREATE OPERATOR FAMILY integer_ops USING btree;
CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- standard int8 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ;
CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
```
-- standard int4 comparisons
OPERATOR 1 <,
OPERATOR 2 <=,
OPERATOR 3 =,
OPERATOR 4 >=,
OPERATOR 5 >,
FUNCTION 1 btint4cmp(int4, int4);

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
-- standard int2 comparisons
OPERATOR 1 <,
OPERATOR 2 <=,
OPERATOR 3 =,
OPERATOR 4 >=,
OPERATOR 5 >,
FUNCTION 1 btint2cmp(int2, int2);

ALTER OPERATOR FAMILY integer_ops USING btree ADD
-- cross-type comparisons int8 vs int2
OPERATOR 1 < (int8, int2),
OPERATOR 2 <= (int8, int2),
OPERATOR 3 = (int8, int2),
OPERATOR 4 >= (int8, int2),
OPERATOR 5 > (int8, int2),
FUNCTION 1 btint82cmp(int8, int2),

-- cross-type comparisons int8 vs int4
OPERATOR 1 < (int8, int4),
OPERATOR 2 <= (int8, int4),
OPERATOR 3 = (int8, int4),
OPERATOR 4 >= (int8, int4),
OPERATOR 5 > (int8, int4),
FUNCTION 1 btint84cmp(int8, int4),

-- cross-type comparisons int4 vs int2
OPERATOR 1 < (int4, int2),
OPERATOR 2 <= (int4, int2),
OPERATOR 3 = (int4, int2),
OPERATOR 4 >= (int4, int2),
OPERATOR 5 > (int4, int2),
FUNCTION 1 btint42cmp(int4, int2),

-- cross-type comparisons int4 vs int8
OPERATOR 1 < (int4, int8),
OPERATOR 2 <= (int4, int8),
OPERATOR 3 = (int4, int8),
OPERATOR 4 >= (int4, int8),
OPERATOR 5 > (int4, int8),
FUNCTION 1 btint48cmp(int4, int8),

-- cross-type comparisons int2 vs int8
OPERATOR 1 < (int2, int8),
OPERATOR 2 <= (int2, int8),
OPERATOR 3 = (int2, int8),
OPERATOR 4 >= (int2, int8),
OPERATOR 5 > (int2, int8),
FUNCTION 1 btint28cmp(int2, int8) ,
-- cross-type comparisons int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;

Notice that this definition “overloads” the operator strategy and support function numbers: each number occurs multiple times within the family. This is allowed so long as each instance of a particular number has distinct input data types. The instances that have both input types equal to an operator class’s input type are the primary operators and support functions for that operator class, and in most cases should be declared as part of the operator class rather than as loose members of the family.

In a B-tree operator family, all the operators in the family must sort compatibly, meaning that the transitive laws hold across all the data types supported by the family: “if A = B and B = C, then A = C”, and “if A < B and B < C, then A < C”. For each operator in the family there must be a support function having the same two input data types as the operator. It is recommended that a family be complete, i.e., for each combination of data types, all operators are included. Each operator class should include just the non-cross-type operators and support function for its data type.

To build a multiple-data-type hash operator family, compatible hash support functions must be created for each data type supported by the family. Here compatibility means that the functions are guaranteed to return the same hash code for any two values that are considered equal by the family’s equality operators, even when the values are of different types. This is usually difficult to accomplish when the types have different physical representations, but it can be done in some cases. Notice that there is only one support function per data type, not one per equality operator. It is recommended that a family be complete, i.e., provide an equality operator for each combination of data types. Each operator class should include just the non-cross-type equality operator and the support function for its data type.

GIN and GiST indexes do not have any explicit notion of cross-data-type operations. The set of operators supported is just whatever the primary support functions for a given operator class can handle.

Note: Prior to PostgreSQL 8.3, there was no concept of operator families, and so any cross-data-type operators intended to be used with an index had to be bound directly into the index’s operator class. While this approach still works, it is deprecated because it makes an index’s dependencies too broad, and because the planner can handle cross-data-type comparisons more effectively when both data types have operators in the same operator family.

34.14.6. System Dependencies on Operator Classes

PostgreSQL uses operator classes to infer the properties of operators in more ways than just whether they can be used with indexes. Therefore, you might want to create operator classes even if you have no intention of indexing any columns of your data type.

In particular, there are SQL features such as ORDER BY and DISTINCT that require comparison and sorting of values. To implement these features on a user-defined data type, PostgreSQL looks for the default B-tree operator class for the data type. The “equals” member of this operator class defines the system’s notion of equality of values for GROUP BY and DISTINCT, and the sort ordering imposed by the operator class defines the default ORDER BY ordering.
Comparison of arrays of user-defined types also relies on the semantics defined by the default B-tree operator class.

If there is no default B-tree operator class for a data type, the system will look for a default hash operator class. But since that kind of operator class only provides equality, in practice it is only enough to support array equality.

When there is no default operator class for a data type, you will get errors like “could not identify an ordering operator” if you try to use these SQL features with the data type.

Note: In PostgreSQL versions before 7.4, sorting and grouping operations would implicitly use operators named =, <, and >. The new behavior of relying on default operator classes avoids having to make any assumption about the behavior of operators with particular names.

Another important point is that an operator that appears in a hash operator family is a candidate for hash joins, hash aggregation, and related optimizations. The hash operator family is essential here since it identifies the hash function(s) to use.

34.14.7. Special Features of Operator Classes

There are two special features of operator classes that we have not discussed yet, mainly because they are not useful with the most commonly used index methods.

Normally, declaring an operator as a member of an operator class (or family) means that the index method can retrieve exactly the set of rows that satisfy a WHERE condition using the operator. For example:

```
SELECT * FROM table WHERE integer_column < 4;
```

can be satisfied exactly by a B-tree index on the integer column. But there are cases where an index is useful as an inexact guide to the matching rows. For example, if a GiST index stores only bounding boxes for objects, then it cannot exactly satisfy a WHERE condition that tests overlap between non-rectangular objects such as polygons. Yet we could use the index to find objects whose bounding box overlaps the bounding box of the target object, and then do the exact overlap test only on the objects found by the index. If this scenario applies, the index is said to be “lossy” for the operator, and we add RECHECK to the OPERATOR clause in the CREATE OPERATOR CLASS command. RECHECK is valid if the index is guaranteed to return all the required rows, plus perhaps some additional rows, which can be eliminated by performing the original operator invocation.

Consider again the situation where we are storing in the index only the bounding box of a complex object such as a polygon. In this case there’s not much value in storing the whole polygon in the index entry — we might as well store just a simpler object of type box. This situation is expressed by the STORAGE option in CREATE OPERATOR CLASS: we’d write something like:

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
    ...
    STORAGE box;
```

At present, only the GiST and GIN index methods support a STORAGE type that’s different from the column data type. The GiST compress and decompress support routines must deal with data-type conversion when STORAGE is used. In GIN, the STORAGE type identifies the type of the “key” values, which normally is different from the type of the indexed column — for example, an operator
class for integer-array columns might have keys that are just integers. The GIN extractValue and extractQuery support routines are responsible for extracting keys from indexed values.
Chapter 35. Triggers

This chapter provides general information about writing trigger functions. Trigger functions can be written in most of the available procedural languages, including PL/pgSQL (Chapter 38), PL/Tcl (Chapter 39), PL/Perl (Chapter 40), and PL/Python (Chapter 41). After reading this chapter, you should consult the chapter for your favorite procedural language to find out the language-specific details of writing a trigger in it.

It is also possible to write a trigger function in C, although most people find it easier to use one of the procedural languages. It is not currently possible to write a trigger function in the plain SQL function language.

35.1. Overview of Trigger Behavior

A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed. Triggers can be defined to execute either before or after any INSERT, UPDATE, or DELETE operation, either once per modified row, or once per SQL statement. If a trigger event occurs, the trigger's function is called at the appropriate time to handle the event.

The trigger function must be defined before the trigger itself can be created. The trigger function must be declared as a function taking no arguments and returning type trigger. (The trigger function receives its input through a specially-passed TriggerData structure, not in the form of ordinary function arguments.)

Once a suitable trigger function has been created, the trigger is established with CREATE TRIGGER. The same trigger function can be used for multiple triggers.

PostgreSQL offers both per-row triggers and per-statement triggers. With a per-row trigger, the trigger function is invoked once for each row that is affected by the statement that fired the trigger. In contrast, a per-statement trigger is invoked only once when an appropriate statement is executed, regardless of the number of rows affected by that statement. In particular, a statement that affects zero rows will still result in the execution of any applicable per-statement triggers. These two types of triggers are sometimes called row-level triggers and statement-level triggers, respectively.

Triggers are also classified as before triggers and after triggers. Statement-level before triggers naturally fire before the statement starts to do anything, while statement-level after triggers fire at the very end of the statement. Row-level before triggers fire immediately before a particular row is operated on, while row-level after triggers fire at the end of the statement (but before any statement-level after triggers).

Trigger functions invoked by per-statement triggers should always return NULL. Trigger functions invoked by per-row triggers can return a table row (a value of type HeapTuple) to the calling executor, if they choose. A row-level trigger fired before an operation has the following choices:

- It can return NULL to skip the operation for the current row. This instructs the executor to not perform the row-level operation that invoked the trigger (the insertion or modification of a particular table row).
- For row-level INSERT and UPDATE triggers only, the returned row becomes the row that will be inserted or will replace the row being updated. This allows the trigger function to modify the row being inserted or updated.

A row-level before trigger that does not intend to cause either of these behaviors must be careful to return as its result the same row that was passed in (that is, the NEW row for INSERT and UPDATE
triggers, the OLD row for DELETE triggers).

The return value is ignored for row-level triggers fired after an operation, and so they can return NULL.

If more than one trigger is defined for the same event on the same relation, the triggers will be fired in alphabetical order by trigger name. In the case of before triggers, the possibly-modified row returned by each trigger becomes the input to the next trigger. If any before trigger returns NULL, the operation is abandoned for that row and subsequent triggers are not fired.

Typically, row before triggers are used for checking or modifying the data that will be inserted or updated. For example, a before trigger might be used to insert the current time into a timestamp column, or to check that two elements of the row are consistent. Row after triggers are most sensibly used to propagate the updates to other tables, or make consistency checks against other tables. The reason for this division of labor is that an after trigger can be certain it is seeing the final value of the row, while a before trigger cannot; there might be other before triggers firing after it. If you have no specific reason to make a trigger before or after, the before case is more efficient, since the information about the operation doesn’t have to be saved until end of statement.

If a trigger function executes SQL commands then these commands might fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger; for example, an INSERT trigger might execute a command that inserts an additional row into the same table, causing the INSERT trigger to be fired again. It is the trigger programmer’s responsibility to avoid infinite recursion in such scenarios.

When a trigger is being defined, arguments can be specified for it. The purpose of including arguments in the trigger definition is to allow different triggers with similar requirements to call the same function. As an example, there could be a generalized trigger function that takes as its arguments two column names and puts the current user in one and the current time stamp in the other. Properly written, this trigger function would be independent of the specific table it is triggering on. So the same function could be used for INSERT events on any table with suitable columns, to automatically track creation of records in a transaction table for example. It could also be used to track last-update events if defined as an UPDATE trigger.

Each programming language that supports triggers has its own method for making the trigger input data available to the trigger function. This input data includes the type of trigger event (e.g., INSERT or UPDATE) as well as any arguments that were listed in CREATE TRIGGER. For a row-level trigger, the input data also includes the NEW row for INSERT and UPDATE triggers, and/or the OLD row for UPDATE and DELETE triggers. Statement-level triggers do not currently have any way to examine the individual row(s) modified by the statement.

### 35.2. Visibility of Data Changes

If you execute SQL commands in your trigger function, and these commands access the table that the trigger is for, then you need to be aware of the data visibility rules, because they determine whether these SQL commands will see the data change that the trigger is fired for. Briefly:

- Statement-level triggers follow simple visibility rules: none of the changes made by a statement are visible to statement-level triggers that are invoked before the statement, whereas all modifications are visible to statement-level after triggers.
- The data change (insertion, update, or deletion) causing the trigger to fire is naturally not visible to SQL commands executed in a row-level before trigger, because it hasn’t happened yet.
Chapter 35. Triggers

- However, SQL commands executed in a row-level before trigger will see the effects of data changes for rows previously processed in the same outer command. This requires caution, since the ordering of these change events is not in general predictable; a SQL command that affects multiple rows can visit the rows in any order.

- When a row-level after trigger is fired, all data changes made by the outer command are already complete, and are visible to the invoked trigger function.

Further information about data visibility rules can be found in Section 42.4. The example in Section 35.4 contains a demonstration of these rules.

35.3. Writing Trigger Functions in C

This section describes the low-level details of the interface to a trigger function. This information is only needed when writing trigger functions in C. If you are using a higher-level language then these details are handled for you. In most cases you should consider using a procedural language before writing your triggers in C. The documentation of each procedural language explains how to write a trigger in that language.

Trigger functions must use the “version 1” function manager interface.

When a function is called by the trigger manager, it is not passed any normal arguments, but it is passed a “context” pointer pointing to a `TriggerData` structure. C functions can check whether they were called from the trigger manager or not by executing the macro:

```
CALLED_AS_TRIGGER(fcinfo)
```

which expands to:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

If this returns true, then it is safe to cast `fcinfo->context` to type `TriggerData *` and make use of the pointed-to `TriggerData` structure. The function must not alter the `TriggerData` structure or any of the data it points to.

`struct TriggerData` is defined in commands/trigger.h:

```
typedef struct TriggerData
{
    NodeTag     type;
    TriggerEvent tg_event;
    Relation     tg_relation;
    HeapTuple    tg_trigtuple;
    HeapTuple    tg_newtuple;
    Trigger     *tg_trigger;
    Buffer       tg_trigtuplebuf;
    Buffer       tg_newtuplebuf;
} TriggerData;
```

where the members are defined as follows:

```
type
    Always T_TriggerData.
```
Chapter 35. Triggers

tg_event
Describes the event for which the function is called. You can use the following macros to examine
tg_event:

TRIGGER_FIRED_BEFORE(tg_event)
Returns true if the trigger fired before the operation.

TRIGGER_FIRED_AFTER(tg_event)
Returns true if the trigger fired after the operation.

TRIGGER_FIRED_FOR_ROW(tg_event)
Returns true if the trigger fired for a row-level event.

TRIGGER_FIRED_FOR_STATEMENT(tg_event)
Returns true if the trigger fired for a statement-level event.

TRIGGER_FIRED_BY_INSERT(tg_event)
Returns true if the trigger was fired by an INSERT command.

TRIGGER_FIRED_BY_UPDATE(tg_event)
Returns true if the trigger was fired by an UPDATE command.

TRIGGER_FIRED_BY_DELETE(tg_event)
Returns true if the trigger was fired by a DELETE command.

tg_relation
A pointer to a structure describing the relation that the trigger fired for. Look at utils/rel.h for
details about this structure. The most interesting things are tg_relation->rd_att (descriptor
of the relation tuples) and tg_relation->rd_rel->relname (relation name; the type is not
char* but NameData; use SPI_getrelname(tg_relation) to get a char* if you need a
copy of the name).

tg_trigtuple
A pointer to the row for which the trigger was fired. This is the row being inserted, updated, or
deleted. If this trigger was fired for an INSERT or DELETE then this is what you should return
from the function if you don’t want to replace the row with a different one (in the case of INSERT)
or skip the operation.

tg_newtuple
A pointer to the new version of the row, if the trigger was fired for an UPDATE, and NULL if it is
for an INSERT or a DELETE. This is what you have to return from the function if the event is an
UPDATE and you don’t want to replace this row by a different one or skip the operation.

tg_trigger
A pointer to a structure of type Trigger, defined in utils/rel.h:
typedef struct Trigger
{
    Oid tgoeid;
    char *tgname;
    Oid tgfoid;
    int16 tgttype;
    char tgenabled;
};
Chapter 35. Triggers

bool tgisconstraint;
Oid tgconstrrelid;
Oid tgconstraint;
bool tgdeferrable;
bool tginitdeferred;
int16 tgnargs;
int16 tgnattr;
int16 *tgattr;
char **tgargs;
}

Trigger;

where tgname is the trigger’s name, tgnargs is number of arguments in tgargs, and tgargs is an array of pointers to the arguments specified in the CREATE TRIGGER statement. The other members are for internal use only.

tg_trigtuplebuf

The buffer containing tg_trigtuple, or InvalidBuffer if there is no such tuple or it is not stored in a disk buffer.

tg_newtuplebuf

The buffer containing tg_newtuple, or InvalidBuffer if there is no such tuple or it is not stored in a disk buffer.

A trigger function must return either a HeapTuple pointer or a NULL pointer (not an SQL null value, that is, do not set isNull true). Be careful to return either tg_trigtuple or tg_newtuple, as appropriate, if you don’t want to modify the row being operated on.

35.4. A Complete Example

Here is a very simple example of a trigger function written in C. (Examples of triggers written in procedural languages can be found in the documentation of the procedural languages.)

The function trigf reports the number of rows in the table ttest and skips the actual operation if the command attempts to insert a null value into the column x. (So the trigger acts as a not-null constraint but doesn’t abort the transaction.)

First, the table definition:

CREATE TABLE ttest (
    x integer
);

This is the source code of the trigger function:

#include "postgres.h"
#include "executor/spi.h" /* this is what you need to work with SPI */
#include "commands/trigger.h" /* ... and triggers */

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc tupdesc;
    HeapTuple rettuple;
    char *when;
    bool checknull = false;
    bool isnull;
    int ret, i;

    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
       elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
            && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(INFO, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);
    if (ret < 0)
        elog(NOTICE, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                   SPI_tuptable->tupdesc,
                                   1,
                                   &isnull));

    elog(INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }
After you have compiled the source code, declare the function and the triggers:

```sql
CREATE FUNCTION trigf() RETURNS trigger
AS 'filename'
LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
FOR EACH ROW EXECUTE PROCEDURE trigf();
```

Now you can test the operation of the trigger:

```sql
=> INSERT INTO ttest VALUES (NULL);
INFO: trigf (fired before): there are 0 rows in ttest
INSERT 0 0
-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO: trigf (fired before): there are 0 rows in ttest
INFO: trigf (fired after ): there are 1 rows in ttest

remember what we said about visibility.

INSERT 167793 1

vac= => SELECT * FROM ttest;
x
---
1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO: trigf (fired before): there are 1 rows in ttest
INFO: trigf (fired after ): there are 2 rows in ttest

remember what we said about visibility.

INSERT 167794 1

=> SELECT * FROM ttest;
x
---
1
2
(2 rows)
```
Chapter 35. Triggers

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO: trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO: trigf (fired before): there are 2 rows in ttest
INFO: trigf (fired after): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
  x
---
  1
  4
(2 rows)

=> DELETE FROM ttest;
INFO: trigf (fired before): there are 2 rows in ttest
INFO: trigf (fired before): there are 1 rows in ttest
INFO: trigf (fired after): there are 0 rows in ttest
INFO: trigf (fired after): there are 0 rows in ttest

   ^^^^^^^

remember what we said about visibility.

DELETE 2
=> SELECT * FROM ttest;
  x
---
(0 rows)

There are more complex examples in src/test/regress/regress.c and in contrib/spi.
Chapter 36. The Rule System

This chapter discusses the rule system in PostgreSQL. Production rule systems are conceptually simple, but there are many subtle points involved in actually using them.

Some other database systems define active database rules, which are usually stored procedures and triggers. In PostgreSQL, these can be implemented using functions and triggers as well.

The rule system (more precisely speaking, the query rewrite rule system) is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query planner for planning and execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The theoretical foundations and the power of this rule system are also discussed in *On Rules, Procedures, Caching and Views in Database Systems* and *A Unified Framework for Version Modeling Using Production Rules in a Database System*.

36.1. The Query Tree

To understand how the rule system works it is necessary to know when it is invoked and what its input and results are.

The rule system is located between the parser and the planner. It takes the output of the parser, one query tree, and the user-defined rewrite rules, which are also query trees with some extra information, and creates zero or more query trees as result. So its input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a query tree? It is an internal representation of an SQL statement where the single parts that it is built from are stored separately. These query trees can be shown in the server log if you set the configuration parameters `debug_print_parse`, `debug_print_rewritten`, or `debug_print_plan`. The rule actions are also stored as query trees, in the system catalog `pg_rewrite`. They are not formatted like the log output, but they contain exactly the same information.

Reading a raw query tree requires some experience. But since SQL representations of query trees are sufficient to understand the rule system, this chapter will not teach how to read them.

When reading the SQL representations of the query trees in this chapter it is necessary to be able to identify the parts the statement is broken into when it is in the query tree structure. The parts of a query tree are

the command type

This is a simple value telling which command (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) produced the query tree.

the range table

The range table is a list of relations that are used in the query. In a `SELECT` statement these are the relations given after the `FROM` key word.

Every range table entry identifies a table or view and tells by which name it is called in the other parts of the query. In the query tree, the range table entries are referenced by number rather than by name, so here it doesn’t matter if there are duplicate names as it would in an SQL statement. This can happen after the range tables of rules have been merged in. The examples in this chapter will not have this situation.
Chapter 36. The Rule System

the result relation

This is an index into the range table that identifies the relation where the results of the query go.

SELECT queries normally don’t have a result relation. The special case of a SELECT INTO is mostly identical to a CREATE TABLE followed by a INSERT ... SELECT and is not discussed separately here.

For INSERT, UPDATE, and DELETE commands, the result relation is the table (or view!) where the changes are to take effect.

the target list

The target list is a list of expressions that define the result of the query. In the case of a SELECT, these expressions are the ones that build the final output of the query. They correspond to the expressions between the keywords SELECT and FROM. (* is just an abbreviation for all the column names of a relation. It is expanded by the parser into the individual columns, so the rule system never sees it.)

DELETE commands don’t need a target list because they don’t produce any result. In fact, the planner will add a special CTID entry to the empty target list, but this is after the rule system and will be discussed later; for the rule system, the target list is empty.

For INSERT commands, the target list describes the new rows that should go into the result relation. It consists of the expressions in the VALUES clause or the ones from the SELECT clause in INSERT ... SELECT. The first step of the rewrite process adds target list entries for any columns that were not assigned to by the original command but have defaults. Any remaining columns (with neither a given value nor a default) will be filled in by the planner with a constant null expression.

For UPDATE commands, the target list describes the new rows that should replace the old ones. In the rule system, it contains just the expressions from the SET column = expression part of the command. The planner will handle missing columns by inserting expressions that copy the values from the old row into the new one. And it will add the special CTID entry just as for DELETE, too.

Every entry in the target list contains an expression that can be a constant value, a variable pointing to a column of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators, etc.

the qualification

The query’s qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean that tells whether the operation (INSERT, UPDATE, DELETE, or SELECT) for the final result row should be executed or not. It corresponds to the WHERE clause of an SQL statement.

the join tree

The query’s join tree shows the structure of the FROM clause. For a simple query like SELECT ... FROM a, b, c the join tree is just a list of the FROM items, because we are allowed to join them in any order. But when JOIN expressions, particularly outer joins, are used, we have to join in the order shown by the joins. In that case, the join tree shows the structure of the JOIN expressions. The restrictions associated with particular JOIN clauses (from ON or USING expressions) are stored as qualification expressions attached to those join-tree nodes. It turns out to be convenient to store the top-level WHERE expression as a qualification attached to the top-level join-tree item, too. So really the join tree represents both the FROM and WHERE clauses of a SELECT.
Chapter 36. The Rule System

36.2. Views and the Rule System

Views in PostgreSQL are implemented using the rule system. In fact, there is essentially no difference between:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

compared against the two commands:

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

because this is exactly what the `CREATE VIEW` command does internally. This has some side effects. One of them is that the information about a view in the PostgreSQL system catalogs is exactly the same as it is for a table. So for the parser, there is absolutely no difference between a table and a view. They are the same thing: relations.

36.2.1. How SELECT Rules Work

Rules `ON SELECT` are applied to all queries as the last step, even if the command given is an `INSERT`, `UPDATE` or `DELETE`. And they have different semantics from rules on the other command types in that they modify the query tree in place instead of creating a new one. So `SELECT` rules are described first.

Currently, there can be only one action in an `ON SELECT` rule, and it must be an unconditional `SELECT` action that is `INSTEAD`. This restriction was required to make rules safe enough to open them for ordinary users, and it restricts `ON SELECT` rules to act like views.

The examples for this chapter are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for `INSERT`, `UPDATE`, and `DELETE` operations so that the final result will be a view that behaves like a real table with some magic functionality. This is not such a simple example to start from and this makes things harder to get into. But it’s better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

For the example, we need a little `min` function that returns the lower of 2 integer values. We create that as:

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

The real tables we need in the first two rule system descriptions are these:

```
CREATE TABLE shoe_data (  
    shoename    text, -- primary key
```
Chapter 36. The Rule System

sh_avail integer, -- available number of pairs
slcolor text, -- preferred shoelace color
slminlen real, -- minimum shoelace length
slmaxlen real, -- maximum shoelace length
slunit text -- length unit
);

CREATE TABLE shoelace_data {
  sl_name text, -- primary key
  sl_avail integer, -- available number of pairs
  sl_color text, -- shoelace color
  sl_len real, -- shoelace length
  sl_unit text -- length unit
};

CREATE TABLE unit {
  un_name text, -- primary key
  un_fact real -- factor to transform to cm
};

As you can see, they represent shoe-store data.

The views are created as:

CREATE VIEW shoe AS
SELECT sh.shoename,
  sh.sh_avail,
  sh.slcolor,
  sh.slminlen,
  sh.slminlen * un.un_fact AS slminlen_cm,
  sh.slmaxlen,
  sh.slmaxlen * un.un_fact AS slmaxlen_cm,
  sh.slunit
FROM shoe_data sh, unit un
WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
SELECT s.sl_name,
  s.sl_avail,
  s.sl_color,
  s.sl_len,
  s.sl_unit,
  s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
SELECT rsh.shoename,
  rsh.sh_avail,
  rsl.sl_name,
  rsl.sl_avail,
  min(rsh.sh_avail, rsl.sl_avail) AS total_avail
FROM shoe rsh, shoelace rsl
WHERE rsl.sl_color = rsh.slcolor
AND rsl.sl_len_cm >= rsh.slminlen_cm
AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
Chapter 36. The Rule System

The CREATE VIEW command for the shoelace view (which is the simplest one we have) will create a relation shoelace and an entry in pg_rewrite that tells that there is a rewrite rule that must be applied whenever the relation shoelace is referenced in a query’s range table. The rule has no rule qualification (discussed later, with the non-SELECT rules, since SELECT rules currently cannot have them) and it is INSTEAD. Note that rule qualifications are not the same as query qualifications. The action of our rule has a query qualification. The action of the rule is one query tree that is a copy of the SELECT statement in the view creation command.

**Note:** The two extra range table entries for NEW and OLD (named \*NEW* and \*OLD* for historical reasons in the printed query tree) you can see in the pg_rewrite entry aren’t of interest for SELECT rules.

Now we populate unit, shoe_data and shoelace_data and run a simple query on a view:

```sql
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0, 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9, 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60, 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40, 'inch');

SELECT * FROM shoelace;
```

### shoelace Output

<table>
<thead>
<tr>
<th>sl_name</th>
<th>sl_avail</th>
<th>sl_color</th>
<th>sl_len</th>
<th>sl_unit</th>
<th>sl_len_cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl1</td>
<td>5</td>
<td>black</td>
<td>80</td>
<td>cm</td>
<td>80</td>
</tr>
<tr>
<td>sl2</td>
<td>6</td>
<td>black</td>
<td>100</td>
<td>cm</td>
<td>100</td>
</tr>
<tr>
<td>sl3</td>
<td>0</td>
<td>black</td>
<td>35</td>
<td>inch</td>
<td>88.9</td>
</tr>
<tr>
<td>sl4</td>
<td>8</td>
<td>black</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl5</td>
<td>1</td>
<td>brown</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl6</td>
<td>0</td>
<td>brown</td>
<td>0.9</td>
<td>m</td>
<td>90</td>
</tr>
</tbody>
</table>

(8 rows)

This is the simplest SELECT you can do on our views, so we take this opportunity to explain the basics of view rules. The SELECT * FROM shoelace was interpreted by the parser and produced the query tree:

```sql
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace;
```
FROM shoelace shoelace;

and this is given to the rule system. The rule system walks through the range table and checks if there are rules for any relation. When processing the range table entry for shoelace (the only one up to now) it finds the _RETURN_ rule with the query tree:

```
SELECT s.sl_name, s.sl_avail, s.sl_color, s.sl_len, s.sl_unit, s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace *OLD*, shoelace *NEW*, shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;
```

To expand the view, the rewriter simply creates a subquery range-table entry containing the rule’s action query tree, and substitutes this range table entry for the original one that referenced the view. The resulting rewritten query tree is almost the same as if you had typed:

```
SELECT shoelace.sl_name, shoelace.sl_avail, shoelace.sl_color, shoelace.sl_len, shoelace.sl_unit, shoelace.sl_len_cm
FROM (SELECT s.sl_name, s.sl_avail, s.sl_color, s.sl_len, s.sl_unit, s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace_data s, unit u
WHERE s.sl_unit = u.un_name) shoelace;
```

There is one difference however: the subquery’s range table has two extra entries shoelace *OLD* and shoelace *NEW*. These entries don’t participate directly in the query, since they aren’t referenced by the subquery’s join tree or target list. The rewriter uses them to store the access privilege check information that was originally present in the range-table entry that referenced the view. In this way, the executor will still check that the user has proper privileges to access the view, even though there’s no direct use of the view in the rewritten query.

That was the first rule applied. The rule system will continue checking the remaining range-table entries in the top query (in this example there are no more), and it will recursively check the range-table entries in the added subquery to see if any of them reference views. (But it won’t expand *OLD* or *NEW* — otherwise we’d have infinite recursion!) In this example, there are no rewrite rules for shoelace_data or unit, so rewriting is complete and the above is the final result given to the planner.

Now we want to write a query that finds out for which shoes currently in the store we have the matching shoelaces (color and length) and where the total number of exactly matching pairs is greater or equal to two.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;
```

<table>
<thead>
<tr>
<th>shoename</th>
<th>sh_avail</th>
<th>sl_name</th>
<th>sl_avail</th>
<th>total_avail</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh1</td>
<td>2</td>
<td>sl1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>sh3</td>
<td>4</td>
<td>sl7</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

(2 rows)
Chapter 36. The Rule System

The output of the parser this time is the query tree:

```sql
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM shoe_ready shoe_ready
WHERE shoe_ready.total_avail >= 2;
```

The first rule applied will be the one for the `shoe_ready` view and it results in the query tree:

```sql
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
    FROM shoe rsh, shoelace rsl
    WHERE rsl.sl_color = rsh.slcolor
    AND rsl.sl_len_cm >= rsh.slminlen_cm
    AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail >= 2;
```

Similarly, the rules for `shoe` and `shoelace` are substituted into the range table of the subquery, leading to a three-level final query tree:

```sql
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
    FROM (SELECT sh.shoename,
             sh.sh_avail,
             sh.slcolor,
             sh.slminlen,
             sh.slminlen * un.un_fact AS slminlen_cm,
             sh.slmaxlen,
             sh.slmaxlen * un.un_fact AS slmaxlen_cm,
             sh.slunit
        FROM shoe_data sh, unit un
        WHERE sh.slunit = un.un_name) rsh,
        (SELECT s.sl_name,
             s.sl_avail,
             s.sl_color,
             s.sl_len,
             s.sl_unit,
             s.sl_len * u.un_fact AS sl_len_cm
        FROM shoelace_data s, unit u
        WHERE s.sl_unit = u.un_name) rsl
    WHERE rsl.sl_color = rsh.slcolor
    AND rsl.sl_len_cm >= rsh.slminlen_cm
    AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
```
WHERE shoe_ready.total_avail > 2;

It turns out that the planner will collapse this tree into a two-level query tree: the bottommost \texttt{SELECT} commands will be “pulled up” into the middle \texttt{SELECT} since there’s no need to process them separately. But the middle \texttt{SELECT} will remain separate from the top, because it contains aggregate functions. If we pulled those up it would change the behavior of the topmost \texttt{SELECT}, which we don’t want. However, collapsing the query tree is an optimization that the rewrite system doesn’t have to concern itself with.

### 36.2.2. View Rules in Non-\texttt{SELECT} Statements

Two details of the query tree aren’t touched in the description of view rules above. These are the command type and the result relation. In fact, view rules don’t need this information.

There are only a few differences between a query tree for a \texttt{SELECT} and one for any other command. Obviously, they have a different command type and for a command other than a \texttt{SELECT}, the result relation points to the range-table entry where the result should go. Everything else is absolutely the same. So having two tables \texttt{t1} and \texttt{t2} with columns \texttt{a} and \texttt{b}, the query trees for the two statements:

\begin{verbatim}
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
\end{verbatim}

are nearly identical. In particular:

- The range tables contain entries for the tables \texttt{t1} and \texttt{t2}.
- The target lists contain one variable that points to column \texttt{b} of the range table entry for table \texttt{t2}.
- The qualification expressions compare the columns \texttt{a} of both range-table entries for equality.
- The join trees show a simple join between \texttt{t1} and \texttt{t2}.

The consequence is, that both query trees result in similar execution plans: They are both joins over the two tables. For the \texttt{UPDATE} the missing columns from \texttt{t1} are added to the target list by the planner and the final query tree will read as:

\begin{verbatim}
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
\end{verbatim}

and thus the executor run over the join will produce exactly the same result set as a:

\begin{verbatim}
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
\end{verbatim}

will do. But there is a little problem in \texttt{UPDATE}: The executor does not care what the results from the join it is doing are meant for. It just produces a result set of rows. The difference that one is a \texttt{SELECT} command and the other is an \texttt{UPDATE} is handled in the caller of the executor. The caller still knows (looking at the query tree) that this is an \texttt{UPDATE}, and it knows that this result should go into table \texttt{t1}. But which of the rows that are there has to be replaced by the new row?

To resolve this problem, another entry is added to the target list in \texttt{UPDATE} (and also in \texttt{DELETE}) statements: the current tuple ID (CTID). This is a system column containing the file block number...
and position in the block for the row. Knowing the table, the CTID can be used to retrieve the original row of \( t_1 \) to be updated. After adding the CTID to the target list, the query actually looks like:

\[
\text{SELECT } t1.a, t2.b, t1.ctid \text{ FROM } t1, t2 \text{ WHERE } t1.a = t2.a;
\]

Now another detail of PostgreSQL enters the stage. Old table rows aren’t overwritten, and this is why \texttt{ROLLBACK} is fast. In an \texttt{UPDATE}, the new result row is inserted into the table (after stripping the CTID) and in the row header of the old row, which the CTID pointed to, the \( \text{cmax} \) and \( \text{xmax} \) entries are set to the current command counter and current transaction ID. Thus the old row is hidden, and after the transaction commits the vacuum cleaner can really remove it.

Knowing all that, we can simply apply view rules in absolutely the same way to any command. There is no difference.

### 36.2.3. The Power of Views in PostgreSQL

The above demonstrates how the rule system incorporates view definitions into the original query tree. In the second example, a simple \texttt{SELECT} from one view created a final query tree that is a join of 4 tables (\texttt{unit} was used twice with different names).

The benefit of implementing views with the rule system is, that the planner has all the information about which tables have to be scanned plus the relationships between these tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single query tree. And this is still the situation when the original query is already a join over views. The planner has to decide which is the best path to execute the query, and the more information the planner has, the better this decision can be. And the rule system as implemented in PostgreSQL ensures, that this is all information available about the query up to that point.

### 36.2.4. Updating a View

What happens if a view is named as the target relation for an \texttt{INSERT}, \texttt{UPDATE}, or \texttt{DELETE}? After doing the substitutions described above, we will have a query tree in which the result relation points at a subquery range-table entry. This will not work, so the rewriter throws an error if it sees it has produced such a thing.

To change this, we can define rules that modify the behavior of these kinds of commands. This is the topic of the next section.

### 36.3. Rules on \texttt{INSERT}, \texttt{UPDATE}, and \texttt{DELETE}

Rules that are defined on \texttt{INSERT}, \texttt{UPDATE}, and \texttt{DELETE} are significantly different from the view rules described in the previous section. First, their \texttt{CREATE RULE} command allows more:

- They are allowed to have no action.
- They can have multiple actions.
- They can be \texttt{INSTEAD} or \texttt{ALSO} (the default).
- The pseudorelations \texttt{NEW} and \texttt{OLD} become useful.
• They can have rule qualifications.

Second, they don’t modify the query tree in place. Instead they create zero or more new query trees and can throw away the original one.

### 36.3.1. How Update Rules Work

Keep the syntax:

```sql
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | { command ; command ... } }
```

in mind. In the following, update rules means rules that are defined on `INSERT`, `UPDATE`, or `DELETE`. Update rules get applied by the rule system when the result relation and the command type of a query tree are equal to the object and event given in the `CREATE RULE` command. For update rules, the rule system creates a list of query trees. Initially the query-tree list is empty. There can be zero (`NOTHING` key word), one, or multiple actions. To simplify, we will look at a rule with one action. This rule can have a qualification or not and it can be `INSTEAD` or `ALSO` (the default).

What is a rule qualification? It is a restriction that tells when the actions of the rule should be done and when not. This qualification can only reference the pseudorelations `NEW` and/or `OLD`, which basically represent the relation that was given as object (but with a special meaning).

So we have three cases that produce the following query trees for a one-action rule.

**No qualification, with either `ALSO` or `INSTEAD`**

the query tree from the rule action with the original query tree’s qualification added

**Qualification given and `ALSO`**

the query tree from the rule action with the rule qualification and the original query tree’s qualification added

**Qualification given and `INSTEAD`**

the query tree from the rule action with the rule qualification and the original query tree’s qualification; and the original query tree with the negated rule qualification added

Finally, if the rule is `ALSO`, the unchanged original query tree is added to the list. Since only qualified `INSTEAD` rules already add the original query tree, we end up with either one or two output query trees for a rule with one action.

For `ON INSERT` rules, the original query (if not suppressed by `INSTEAD`) is done before any actions added by rules. This allows the actions to see the inserted row(s). But for `ON UPDATE` and `ON DELETE` rules, the original query is done after the actions added by rules. This ensures that the actions can see the to-be-updated or to-be-deleted rows; otherwise, the actions might do nothing because they find no rows matching their qualifications.

The query trees generated from rule actions are thrown into the rewrite system again, and maybe more rules get applied resulting in more or less query trees. So a rule’s actions must have either a different command type or a different result relation than the rule itself is on, otherwise this recursive process will end up in an infinite loop. (Recursive expansion of a rule will be detected and reported as an error.)

The query trees found in the actions of the `pg_rewrite` system catalog are only templates. Since they can reference the range-table entries for `NEW` and `OLD`, some substitutions have to be made before they can be used. For any reference to `NEW`, the target list of the original query is searched for a
corresponding entry. If found, that entry’s expression replaces the reference. Otherwise, NEW means the same as OLD (for an UPDATE) or is replaced by a null value (for an INSERT). Any reference to OLD is replaced by a reference to the range-table entry that is the result relation.

After the system is done applying update rules, it applies view rules to the produced query tree(s). Views cannot insert new update actions so there is no need to apply update rules to the output of view rewriting.

### 36.3.1.1. A First Rule Step by Step

Say we want to trace changes to the sl_avail column in the shoelace_data relation. So we set up a log table and a rule that conditionally writes a log entry when an UPDATE is performed on shoelace_data.

```sql
CREATE TABLE shoelace_log {
    sl_name text, -- shoelace changed
    sl_avail integer, -- new available value
    log_who text, -- who did it
    log_when timestamp -- when
};

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
WHERE NEW.sl_avail <> OLD.sl_avail
DO INSERT INTO shoelace_log VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    current_user,
    current_timestamp
);
```

Now someone does:

```sql
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

and we look at the log table:

```sql
SELECT * FROM shoelace_log;
```

<table>
<thead>
<tr>
<th>sl_name</th>
<th>sl_avail</th>
<th>log_who</th>
<th>log_when</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl7</td>
<td>6</td>
<td>A1</td>
<td>Tue Oct 20 16:14:45 1998 MET DST</td>
</tr>
</tbody>
</table>

(1 row)

That’s what we expected. What happened in the background is the following. The parser created the query tree:

```sql
UPDATE shoelace_data SET sl_avail = 6
FROM shoelace_data shoelace_data
WHERE shoelace_data.sl_name = 'sl7';
```

There is a rule log_shoelace that is ON UPDATE with the rule qualification expression:

```sql
NEW.sl_avail <> OLD.sl_avail
```
and the action:

```
INSERT INTO shoelace_log VALUES (
  *NEW*.sl_name, *NEW*.sl_avail,
  current_user, current_timestamp )
FROM shoelace_data *NEW*, shoelace_data *OLD*;
```

(This looks a little strange since you cannot normally write `INSERT ... VALUES ... FROM`. The `FROM` clause here is just to indicate that there are range-table entries in the query tree for *NEW* and *OLD*. These are needed so that they can be referenced by variables in the `INSERT` command’s query tree.)

The rule is a qualified `ALSO` rule, so the rule system has to return two query trees: the modified rule action and the original query tree. In step 1, the range table of the original query is incorporated into the rule’s action query tree. This results in:

```
INSERT INTO shoelace_log VALUES (
  *NEW*.sl_name, *NEW*.sl_avail,
  current_user, current_timestamp )
FROM shoelace_data *NEW*, shoelace_data *OLD*,
  shoelace_data shoelace_data;
```

In step 2, the rule qualification is added to it, so the result set is restricted to rows where `sl_avail` changes:

```
INSERT INTO shoelace_log VALUES (
  *NEW*.sl_name, *NEW*.sl_avail,
  current_user, current_timestamp )
FROM shoelace_data *NEW*, shoelace_data *OLD*,
  shoelace_data shoelace_data
WHERE *NEW*.sl_avail <> *OLD*.sl_avail;
```

(This looks even stranger, since `INSERT ... VALUES` doesn’t have a `WHERE` clause either, but the planner and executor will have no difficulty with it. They need to support this same functionality anyway for `INSERT ... SELECT`.)

In step 3, the original query tree’s qualification is added, restricting the result set further to only the rows that would have been touched by the original query:

```
INSERT INTO shoelace_log VALUES (
  *NEW*.sl_name, *NEW*.sl_avail,
  current_user, current_timestamp )
FROM shoelace_data *NEW*, shoelace_data *OLD*,
  shoelace_data shoelace_data
WHERE *NEW*.sl_avail <> *OLD*.sl_avail
  AND shoelace_data.sl_name = 'sl7';
```

Step 4 replaces references to `NEW` by the target list entries from the original query tree or by the matching variable references from the result relation:

```
INSERT INTO shoelace_log VALUES ( 
  shoelace_data.sl_name, 6,
  current_user, current_timestamp )
FROM shoelace_data *NEW*, shoelace_data *OLD*,
  shoelace_data shoelace_data
WHERE 6 <> *OLD*.sl_avail
```
AND shoelace_data.sl_name = 'sl7';

Step 5 changes OLD references into result relation references:

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data *NEW*, shoelace_data *OLD*,
shoelace_data shoelace_data
WHERE 6 <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

That’s it. Since the rule is ALSO, we also output the original query tree. In short, the output from the rule system is a list of two query trees that correspond to these statements:

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data
WHERE 6 <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
WHERE sl_name = 'sl7';
```

These are executed in this order, and that is exactly what the rule was meant to do.

The substitutions and the added qualifications ensure that, if the original query would be, say:

```
UPDATE shoelace_data SET sl_color = 'green'
WHERE sl_name = 'sl7';
```

no log entry would get written. In that case, the original query tree does not contain a target list entry for sl_avail, so NEW.sl_avail will get replaced by shoelace_data.si_avail. Thus, the extra command generated by the rule is:

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, shoelace_data.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data
WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

and that qualification will never be true.

It will also work if the original query modifies multiple rows. So if someone issued the command:

```
UPDATE shoelace_data SET sl_avail = 0
WHERE sl_color = 'black';
```

four rows in fact get updated (sl1, sl2, sl3, and sl4). But sl3 already has sl_avail = 0. In this case, the original query trees qualification is different and that results in the extra query tree:

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
    current_user, current_timestamp
FROM shoelace_data
WHERE shoelace_data.sl_name = 'sl7';
```
FROM shoelace_data
WHERE 0 <> shoelace_data.sl_avail
AND shoelace_data.sl_color = 'black';

being generated by the rule. This query tree will surely insert three new log entries. And that’s absolutely correct.

Here we can see why it is important that the original query tree is executed last. If the UPDATE had been executed first, all the rows would have already been set to zero, so the logging INSERT would not find any row where 0 <> shoelace_data.sl_avail.

### 36.3.2. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can try to run INSERT, UPDATE, or DELETE on them is to let those query trees get thrown away. So we could create the rules:

CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
  DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
  DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
  DO INSTEAD NOTHING;

If someone now tries to do any of these operations on the view relation shoe, the rule system will apply these rules. Since the rules have no actions and are INSTEAD, the resulting list of query trees will be empty and the whole query will become nothing because there is nothing left to be optimized or executed after the rule system is done with it.

A more sophisticated way to use the rule system is to create rules that rewrite the query tree into one that does the right operation on the real tables. To do that on the shoelace view, we create the following rules:

CREATE RULE shoelace_ins AS ON INSERT TO shoelace
  DO INSTEAD
    INSERT INTO shoelace_data VALUES ( NEW.sl_name, 
                                          NEW.sl_avail, 
                                          NEW.sl_color, 
                                          NEW.sl_len, 
                                          NEW.sl_unit );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
  DO INSTEAD
    UPDATE shoelace_data
      SET sl_name = NEW.sl_name, 
          sl_avail = NEW.sl_avail, 
          sl_color = NEW.sl_color, 
          sl_len = NEW.sl_len, 
          sl_unit = NEW.sl_unit 
      WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
  DO INSTEAD
DELETE FROM shoelace_data
WHERE sl_name = OLD.sl_name;

If you want to support RETURNING queries on the view, you need to make the rules include
RETURNING clauses that compute the view rows. This is usually pretty trivial for views on a single
table, but it’s a bit tedious for join views such as shoelace. An example for the insert case is:

CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
  INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit
  )
RETURNING
  shoelace_data.*,
  (SELECT shoelace_data.sl_len * u.un_fact
   FROM unit u WHERE shoelace_data.sl_unit = u.un_name);

Note that this one rule supports both INSERT and INSERT RETURNING queries on the view — the
RETURNING clause is simply ignored for INSERT.

Now assume that once in a while, a pack of shoelaces arrives at the shop and a big parts list along
with it. But you don’t want to manually update the shoelace view every time. Instead we setup two
little tables: one where you can insert the items from the part list, and one with a special trick. The
creation commands for these are:

CREATE TABLE shoelace_arrive (  
  arr_name  text,
  arr_quant integer
);

CREATE TABLE shoelace_ok (  
  ok_name  text,
  ok_quant integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
DO INSTEAD
  UPDATE shoelace
  SET sl_avail = sl_avail + NEW.ok_quant
  WHERE sl_name = NEW.ok_name;

Now you can fill the table shoelace_arrive with the data from the parts list:

SELECT * FROM shoelace_arrive;

arr_name | arr_quant
----------+----------
sl3       | 10       
sl6       | 20       
sl8       | 20       
(3 rows)
Take a quick look at the current data:

```
SELECT * FROM shoelace;
```

<table>
<thead>
<tr>
<th>sl_name</th>
<th>sl_avail</th>
<th>sl_color</th>
<th>sl_len</th>
<th>sl_unit</th>
<th>sl_len_cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl1</td>
<td>5</td>
<td>black</td>
<td>80</td>
<td>cm</td>
<td>80</td>
</tr>
<tr>
<td>sl2</td>
<td>6</td>
<td>black</td>
<td>100</td>
<td>cm</td>
<td>100</td>
</tr>
<tr>
<td>sl7</td>
<td>6</td>
<td>brown</td>
<td>60</td>
<td>cm</td>
<td>60</td>
</tr>
<tr>
<td>sl3</td>
<td>0</td>
<td>black</td>
<td>35</td>
<td>inch</td>
<td>88.9</td>
</tr>
<tr>
<td>sl4</td>
<td>8</td>
<td>black</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl8</td>
<td>1</td>
<td>brown</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl5</td>
<td>4</td>
<td>brown</td>
<td>1</td>
<td>m</td>
<td>100</td>
</tr>
<tr>
<td>sl6</td>
<td>0</td>
<td>brown</td>
<td>0.9</td>
<td>m</td>
<td>90</td>
</tr>
</tbody>
</table>

(8 rows)

Now move the arrived shoelaces in:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

and check the results:

```
SELECT * FROM shoelace ORDER BY sl_name;
```

<table>
<thead>
<tr>
<th>sl_name</th>
<th>sl_avail</th>
<th>sl_color</th>
<th>sl_len</th>
<th>sl_unit</th>
<th>sl_len_cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl1</td>
<td>5</td>
<td>black</td>
<td>80</td>
<td>cm</td>
<td>80</td>
</tr>
<tr>
<td>sl2</td>
<td>6</td>
<td>black</td>
<td>100</td>
<td>cm</td>
<td>100</td>
</tr>
<tr>
<td>sl7</td>
<td>6</td>
<td>brown</td>
<td>60</td>
<td>cm</td>
<td>60</td>
</tr>
<tr>
<td>sl4</td>
<td>8</td>
<td>black</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl3</td>
<td>10</td>
<td>black</td>
<td>35</td>
<td>inch</td>
<td>88.9</td>
</tr>
<tr>
<td>sl8</td>
<td>21</td>
<td>brown</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl5</td>
<td>4</td>
<td>brown</td>
<td>1</td>
<td>m</td>
<td>100</td>
</tr>
<tr>
<td>sl6</td>
<td>20</td>
<td>brown</td>
<td>0.9</td>
<td>m</td>
<td>90</td>
</tr>
</tbody>
</table>

(8 rows)

```
SELECT * FROM shoelace_log;
```

<table>
<thead>
<tr>
<th>sl_name</th>
<th>sl_avail</th>
<th>log_who</th>
<th>log_when</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl7</td>
<td>6</td>
<td>Al</td>
<td>Tue Oct 20 19:14:45 1998 MET DST</td>
</tr>
<tr>
<td>sl3</td>
<td>10</td>
<td>Al</td>
<td>Tue Oct 20 19:25:16 1998 MET DST</td>
</tr>
<tr>
<td>sl6</td>
<td>20</td>
<td>Al</td>
<td>Tue Oct 20 19:25:16 1998 MET DST</td>
</tr>
<tr>
<td>sl8</td>
<td>21</td>
<td>Al</td>
<td>Tue Oct 20 19:25:16 1998 MET DST</td>
</tr>
</tbody>
</table>

(4 rows)

It’s a long way from the one INSERT ... SELECT to these results. And the description of the query-tree transformation will be the last in this chapter. First, there is the parser’s output:

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
    FROM shoelace_arrive, shoelace_ok;
```

Now the first rule shoelace_ok_ins is applied and turns this into:

```
UPDATE shoelace
```
SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
    shoelace_ok *OLD*, shoelace_ok *NEW*,
shoelace shoelace
WHERE shoelace.sl_name = shoelace_arrive.arr_name;

and throws away the original INSERT on shoelace_ok. This rewritten query is passed to the rule system again, and the second applied rule shoelace_upd produces:

UPDATE shoelace_data
SET sl_name = shoelace.sl_name,
    sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
    sl_color = shoelace.sl_color,
    sl_len = shoelace.sl_len,
    sl_unit = shoelace.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
    shoelace_ok *OLD*, shoelace_ok *NEW*,
shoelace shoelace, shoelace *OLD*,
shoelace *NEW*, shoelace_data shoelace_data
WHERE shoelace.sl_name = shoelace_arrive.arr_name
    AND shoelace_data.sl_name = shoelace.sl_name;

Again it’s an INSTEAD rule and the previous query tree is trashed. Note that this query still uses the view shoelace. But the rule system isn’t finished with this step, so it continues and applies the _RETURN rule on it, and we get:

UPDATE shoelace_data
SET sl_name = s.sl_name,
    sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
    sl_color = s.sl_color,
    sl_len = s.sl_len,
    sl_unit = s.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
    shoelace_ok *OLD*, shoelace_ok *NEW*,
shoelace shoelace, shoelace *OLD*,
shoelace *NEW*, shoelace_data shoelace_data,
shoelace_data s, unit u
WHERE s.sl_name = shoelace_arrive.arr_name
    AND shoelace_data.sl_name = s.sl_name;

Finally, the rule log_shoelace gets applied, producing the extra query tree:

INSERT INTO shoelace_log
SELECT s.sl_name,
    s.sl_avail + shoelace_arrive.arr_quant,
    current_user,
    current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
    shoelace_ok *OLD*, shoelace_ok *NEW*,
shoelace shoelace, shoelace *OLD*,
shoelace *NEW*, shoelace_data shoelace_data,
shoelace *OLD*, shoelace *NEW*,
shoelace_data s, unit u,
shoelace_data *OLD*, shoelace_data *NEW*
shoelace_log shoelace_log
WHERE s.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = s.sl_name
AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;

After that the rule system runs out of rules and returns the generated query trees.
So we end up with two final query trees that are equivalent to the SQL statements:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
     s.sl_avail + shoelace_arrive.arr_quant,
     current_user,
     current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
     shoelace_data s
WHERE s.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = s.sl_name
AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;
```

```
UPDATE shoelace_data
SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive,
     shoelace_data shoelace_data,
     shoelace_data s
WHERE s.sl_name = shoelace_arrive.sl_name
AND shoelace_data.sl_name = s.sl_name;
```

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that’s a bit ugly. Looking at the two queries, it turns out that the `shoelace_data` relation appears twice in the range table where it could definitely be reduced to one. The planner does not handle it and so the execution plan for the rule systems output of the `INSERT` will be

```
Nested Loop
  -> Merge Join
     -> Seq Scan
        -> Sort
        -> Seq Scan on s
     -> Seq Scan
        -> Sort
        -> Seq Scan on shoelace_arrive
     -> Seq Scan on shoelace_data
```

while omitting the extra range table entry would result in a

```
Merge Join
  -> Seq Scan
     -> Sort
     -> Seq Scan on s
     -> Seq Scan
        -> Sort
        -> Seq Scan on shoelace_arrive
```

which produces exactly the same entries in the log table. Thus, the rule system caused one extra scan on the table `shoelace_data` that is absolutely not necessary. And the same redundant scan is done once more in the `UPDATE`. But it was a really hard job to make that all possible at all.

Now we make a final demonstration of the PostgreSQL rule system and its power. Say you add some shoelaces with extraordinary colors to your database:
We would like to make a view to check which shoelace entries do not fit any shoe in color. The view for this is:

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Its output is:

```
SELECT * FROM shoelace_mismatch;
```

```
<table>
<thead>
<tr>
<th>sl_name</th>
<th>sl_avail</th>
<th>sl_color</th>
<th>sl_len</th>
<th>sl_unit</th>
<th>sl_len_cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl9</td>
<td>0</td>
<td>pink</td>
<td>35</td>
<td>inch</td>
<td>88.9</td>
</tr>
<tr>
<td>sl10</td>
<td>1000</td>
<td>magenta</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
</tbody>
</table>
```

Now we want to set it up so that mismatching shoelaces that are not in stock are deleted from the database. To make it a little harder for PostgreSQL, we don’t delete it directly. Instead we create one more view:

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

and do it this way:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
        WHERE sl_name = shoelace.sl_name);
```

Voilà:

```
SELECT * FROM shoelace;
```

```
<table>
<thead>
<tr>
<th>sl_name</th>
<th>sl_avail</th>
<th>sl_color</th>
<th>sl_len</th>
<th>sl_unit</th>
<th>sl_len_cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl1</td>
<td>5</td>
<td>black</td>
<td>80</td>
<td>cm</td>
<td>80</td>
</tr>
<tr>
<td>sl2</td>
<td>6</td>
<td>black</td>
<td>100</td>
<td>cm</td>
<td>100</td>
</tr>
<tr>
<td>sl7</td>
<td>6</td>
<td>brown</td>
<td>60</td>
<td>cm</td>
<td>60</td>
</tr>
<tr>
<td>sl4</td>
<td>8</td>
<td>black</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl3</td>
<td>10</td>
<td>black</td>
<td>35</td>
<td>inch</td>
<td>88.9</td>
</tr>
<tr>
<td>sl8</td>
<td>21</td>
<td>brown</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl10</td>
<td>1000</td>
<td>magenta</td>
<td>40</td>
<td>inch</td>
<td>101.6</td>
</tr>
<tr>
<td>sl5</td>
<td>4</td>
<td>brown</td>
<td>1</td>
<td>m</td>
<td>100</td>
</tr>
<tr>
<td>sl6</td>
<td>20</td>
<td>brown</td>
<td>0.9</td>
<td>m</td>
<td>90</td>
</tr>
</tbody>
</table>
```

A DELETE on a view, with a subquery qualification that in total uses 4 nesting/joined views, where one of them itself has a subquery qualification containing a view and where calculated view columns are used, gets rewritten into one single query tree that deletes the requested data from a real table.

There are probably only a few situations out in the real world where such a construct is necessary. But it makes you feel comfortable that it works.
Chapter 36. The Rule System

36.4. Rules and Privileges

Due to rewriting of queries by the PostgreSQL rule system, other tables/views than those used in the original query get accessed. When update rules are used, this can include write access to tables.

Rewrite rules don’t have a separate owner. The owner of a relation (table or view) is automatically the owner of the rewrite rules that are defined for it. The PostgreSQL rule system changes the behavior of the default access control system. Relations that are used due to rules get checked against the privileges of the rule owner, not the user invoking the rule. This means that a user only needs the required privileges for the tables/views that he names explicitly in his queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the secretary of the office. He can construct the following:

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO secretary;
```

Nobody except him (and the database superusers) can access the `phone_data` table. But because of the `GRANT`, the secretary can run a `SELECT` on the `phone_number` view. The rule system will rewrite the `SELECT` from `phone_number` into a `SELECT` from `phone_data`. Since the user is the owner of `phone_number` and therefore the owner of the rule, the read access to `phone_data` is now checked against his privileges and the query is permitted. The check for accessing `phone_number` is also performed, but this is done against the invoking user, so nobody but the user and the secretary can use it.

The privileges are checked rule by rule. So the secretary is for now the only one who can see the public phone numbers. But the secretary can setup another view and grant access to that to the public. Then, anyone can see the `phone_number` data through the secretary’s view. What the secretary cannot do is to create a view that directly accesses `phone_data`. (Actually he can, but it will not work since every access will be denied during the permission checks.) And as soon as the user will notice, that the secretary opened his `phone_number` view, he can revoke his access. Immediately, any access to the secretary’s view would fail.

One might think that this rule-by-rule checking is a security hole, but in fact it isn’t. But if it did not work this way, the secretary could set up a table with the same columns as `phone_number` and copy the data to there once per day. Then it’s his own data and he can grant access to everyone he wants. A `GRANT` command means, “I trust you”. If someone you trust does the thing above, it’s time to think it over and then use `REVOKE`.

Note that while views can be used to hide the contents of certain columns using the technique shown above, they cannot be used to reliably conceal the data in unseen rows. For example, the following view is insecure:

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

This view might seem secure, since the rule system will rewrite any `SELECT` from `phone_number` into a `SELECT` from `phone_data` and add the qualification that only entries where `phone` does not begin with 412 are wanted. But if the user can create his or her own functions, it is not difficult to convince the planner to execute the user-defined function prior to the `NOT LIKE` expression.

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
$$;
```

Note that while views can be used to hide the contents of certain columns using the technique shown above, they cannot be used to reliably conceal the data in unseen rows. For example, the following view is insecure:

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

This view might seem secure, since the rule system will rewrite any `SELECT` from `phone_number` into a `SELECT` from `phone_data` and add the qualification that only entries where `phone` does not begin with 412 are wanted. But if the user can create his or her own functions, it is not difficult to convince the planner to execute the user-defined function prior to the `NOT LIKE` expression.

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
$$
```
Chapter 36. The Rule System

```sql
RETURN true;
END
$$ LANGUAGE plpgsql COST 0.0000000000000000000001;
SELECT * FROM phone_number WHERE tricky(person, phone);
```

Every person and phone number in the `phone_data` table will be printed as a NOTICE, because the planner will choose to execute the inexpensive `tricky` function before the more expensive NOT LIKE. Even if the user is prevented from defining new functions, built-in functions can be used in similar attacks. (For example, casting functions include their inputs in the error messages they produce.)

Similar considerations apply to update rules. In the examples of the previous section, the owner of the tables in the example database could grant the privileges SELECT, INSERT, UPDATE, and DELETE on the `shoelace` view to someone else, but only SELECT on `shoelace_log`. The rule action to write log entries will still be executed successfully, and that other user could see the log entries. But he cannot create fake entries, nor could he manipulate or remove existing ones. In this case, there is no possibility of subverting the rules by convincing the planner to alter the order of operations, because the only rule which references `shoelace_log` is an unqualified INSERT. This might not be true in more complex scenarios.

36.5. Rules and Command Status

The PostgreSQL server returns a command status string, such as `INSERT 149592 1`, for each command it receives. This is simple enough when there are no rules involved, but what happens when the query is rewritten by rules?

Rules affect the command status as follows:

- If there is no unconditional INSTEAD rule for the query, then the originally given query will be executed, and its command status will be returned as usual. (But note that if there were any conditional INSTEAD rules, the negation of their qualifications will have been added to the original query. This might reduce the number of rows it processes, and if so the reported status will be affected.)
- If there is any unconditional INSTEAD rule for the query, then the original query will not be executed at all. In this case, the server will return the command status for the last query that was inserted by an INSTEAD rule (conditional or unconditional) and is of the same command type (INSERT, UPDATE, or DELETE) as the original query. If no query meeting those requirements is added by any rule, then the returned command status shows the original query type and zeroes for the row-count and OID fields.

(This system was established in PostgreSQL 7.3. In versions before that, the command status might show different results when rules exist.)

The programmer can ensure that any desired INSTEAD rule is the one that sets the command status in the second case, by giving it the alphabetically last rule name among the active rules, so that it gets applied last.

36.6. Rules versus Triggers

Many things that can be done using triggers can also be implemented using the PostgreSQL rule system. One of the things that cannot be implemented by rules are some kinds of constraints, especially foreign keys. It is possible to place a qualified rule that rewrites a command to NOTHING if the value
Chapter 36. The Rule System

of a column does not appear in another table. But then the data is silently thrown away and that’s not a good idea. If checks for valid values are required, and in the case of an invalid value an error message should be generated, it must be done by a trigger.

On the other hand, a trigger that is fired on INSERT on a view can do the same as a rule: put the data somewhere else and suppress the insert in the view. But it cannot do the same thing on UPDATE or DELETE, because there is no real data in the view relation that could be scanned, and thus the trigger would never get called. Only a rule will help.

For the things that can be implemented by both, which is best depends on the usage of the database. A trigger is fired for any affected row once. A rule manipulates the query or generates an additional query. So if many rows are affected in one statement, a rule issuing one extra command is likely to be faster than a trigger that is called for every single row and must execute its operations many times. However, the trigger approach is conceptually far simpler than the rule approach, and is easier for novices to get right.

Here we show an example of how the choice of rules versus triggers plays out in one situation. There are two tables:

```sql
CREATE TABLE computer {
    hostname text, -- indexed
    manufacturer text -- indexed
};

CREATE TABLE software {
    software text, -- indexed
    hostname text -- indexed
};
```

Both tables have many thousands of rows and the indexes on `hostname` are unique. The rule or trigger should implement a constraint that deletes rows from `software` that reference a deleted computer. The trigger would use this command:

```sql
DELETE FROM software WHERE hostname = $1;
```

Since the trigger is called for each individual row deleted from `computer`, it can prepare and save the plan for this command and pass the `hostname` value in the parameter. The rule would be written as:

```sql
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Now we look at different types of deletes. In the case of a:

```sql
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

the table `computer` is scanned by index (fast), and the command issued by the trigger would also use an index scan (also fast). The extra command from the rule would be:

```sql
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
    AND software.hostname = computer.hostname;
```

Since there are appropriate indexes setup, the planner will create a plan of

```
Nestloop
  -> Index Scan using comp_hostidx on computer
  -> Index Scan using soft_hostidx on software
```
So there would be not that much difference in speed between the trigger and the rule implementation.

With the next delete we want to get rid of all the 2000 computers where the hostname starts with old. There are two possible commands to do that. One is:

```sql
DELETE FROM computer WHERE hostname >= 'old'
    AND hostname < 'ole'
```

The command added by the rule will be:

```sql
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
    AND software.hostname = computer.hostname;
```

with the plan

```
Hash Join
  -> Seq Scan on software
  -> Hash
    -> Index Scan using comp_hostidx on computer
```

The other possible command is:

```sql
DELETE FROM computer WHERE hostname ~ '^old';
```

which results in the following executing plan for the command added by the rule:

```
Nestloop
  -> Index Scan using comp_hostidx on computer
  -> Index Scan using soft_hostidx on software
```

This shows, that the planner does not realize that the qualification for hostname in computer could also be used for an index scan on software when there are multiple qualification expressions combined with AND, which is what it does in the regular-expression version of the command. The trigger will get invoked once for each of the 2000 old computers that have to be deleted, and that will result in one index scan over computer and 2000 index scans over software. The rule implementation will do it with two commands that use indexes. And it depends on the overall size of the table software whether the rule will still be faster in the sequential scan situation. 2000 command executions from the trigger over the SPI manager take some time, even if all the index blocks will soon be in the cache.

The last command we look at is:

```sql
DELETE FROM computer WHERE manufacturer = 'bim';
```

Again this could result in many rows to be deleted from computer. So the trigger will again run many commands through the executor. The command generated by the rule will be:

```sql
DELETE FROM software WHERE computer.manufacturer = 'bim'
    AND software.hostname = computer.hostname;
```

The plan for that command will again be the nested loop over two index scans, only using a different index on computer:

```
Nestloop
  -> Index Scan using comp_manufidx on computer
  -> Index Scan using soft_hostidx on software
```

In any of these cases, the extra commands from the rule system will be more or less independent from the number of affected rows in a command.
The summary is, rules will only be significantly slower than triggers if their actions result in large and badly qualified joins, a situation where the planner fails.
Chapter 37. Procedural Languages

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called procedural languages (PLs). For a function written in a procedural language, the database server has no built-in knowledge about how to interpret the function’s source text. Instead, the task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, etc. itself, or it could serve as “glue” between PostgreSQL and an existing implementation of a programming language. The handler itself is a C language function compiled into a shared object and loaded on demand, just like any other C function.

There are currently four procedural languages available in the standard PostgreSQL distribution: PL/pgSQL (Chapter 38), PL/Tcl (Chapter 39), PL/Perl (Chapter 40), and PL/Python (Chapter 41). There are additional procedural languages available that are not included in the core distribution. Appendix G has information about finding them. In addition other languages can be defined by users; the basics of developing a new procedural language are covered in Chapter 48.

37.1. Installing Procedural Languages

A procedural language must be “installed” into each database where it is to be used. But procedural languages installed in the database template1 are automatically available in all subsequently created databases, since their entries in template1 will be copied by CREATE DATABASE. So the database administrator can decide which languages are available in which databases and can make some languages available by default if he chooses.

For the languages supplied with the standard distribution, it is only necessary to execute CREATE LANGUAGE language_name to install the language into the current database. Alternatively, the program createlang can be used to do this from the shell command line. For example, to install the language PL/pgSQL into the database template1, use:

```sql
createlang plpgsql template1
```

The manual procedure described below is only recommended for installing custom languages that CREATE LANGUAGE does not know about.

Manual Procedural Language Installation

A procedural language is installed in a database in four steps, which must be carried out by a database superuser. (For languages known to CREATE LANGUAGE, the second and third steps can be omitted, because they will be carried out automatically if needed.)

1. The shared object for the language handler must be compiled and installed into an appropriate library directory. This works in the same way as building and installing modules with regular user-defined C functions does; see Section 34.9.6. Often, the language handler will depend on an external library that provides the actual programming language engine; if so, that must be installed as well.

2. The handler must be declared with the command

```sql
CREATE FUNCTION handler_function_name() RETURNS language_handler AS 'path-to-shared-object'
LANGUAGE C;
```
Chapter 37. Procedural Languages

The special return type of `language_handler` tells the database system that this function does not return one of the defined SQL data types and is not directly usable in SQL statements.

3. Optionally, the language handler can provide a “validator” function that checks a function definition for correctness without actually executing it. The validator function is called by `CREATE FUNCTION` if it exists. If a validator function is provided by the handler, declare it with a command like

   ```sql
   CREATE FUNCTION validator_function_name(oid)
   RETURNS void
   AS 'path-to-shared-object'
   LANGUAGE C;
   ```

4. The PL must be declared with the command

   ```sql
   CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
   HANDLER handler_function_name
   [VALIDATOR validator_function_name] ;
   ```

   The optional key word `TRUSTED` specifies that ordinary database users that have no superuser privileges should be allowed to use this language to create functions and trigger procedures. Since PL functions are executed inside the database server, the `TRUSTED` flag should only be given for languages that do not allow access to database server internals or the file system. The languages PL/pgSQL, PL/Tcl, and PL/Perl are considered trusted; the languages PL/TclU, PL/PerlU, and PL/PythonU are designed to provide unlimited functionality and should not be marked trusted.

Example 37-1 shows how the manual installation procedure would work with the language PL/pgSQL.

**Example 37-1. Manual Installation of PL/pgSQL**

The following command tells the database server where to find the shared object for the PL/pgSQL language’s call handler function.

   ```sql
   CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS
   '$libdir/plpgsql' LANGUAGE C;
   ```

   PL/pgSQL has a validator function, so we declare that too:

   ```sql
   CREATE FUNCTION plpgsql_validator(oid) RETURNS void AS
   '$libdir/plpgsql' LANGUAGE C;
   ```

   The command:

   ```sql
   CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
   HANDLER plpgsql_call_handler
   VALIDATOR plpgsql_validator;
   ```

   then defines that the previously declared functions should be invoked for functions and trigger procedures where the language attribute is `plpgsql`.

In a default PostgreSQL installation, the handler for the PL/pgSQL language is built and installed into the “library” directory. If Tcl support is configured in, the handlers for PL/Tcl and PL/TclU are also built and installed in the same location. Likewise, the PL/Perl and PL/PerlU handlers are built and installed if Perl support is configured, and the PL/PythonU handler is installed if Python support is configured.
Chapter 38. PL/pgSQL - SQL Procedural Language

38.1. Overview

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user-defined types, functions, and operators,
- can be defined to be trusted by the server,
- is easy to use.

Functions created with PL/pgSQL can be used anywhere that built-in functions could be used. For example, it is possible to create complex conditional computation functions and later use them to define operators or use them in index expressions.

38.1.1. Advantages of Using PL/pgSQL

SQL is the language PostgreSQL and most other relational databases use as query language. It’s portable and easy to learn. But every SQL statement must be executed individually by the database server.

That means that your client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server. All this incurs interprocess communication and will also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL you can group a block of computation and a series of queries inside the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

- Extra round trips between client and server are eliminated
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
- Multiple rounds of query parsing can be avoided

This can result in a considerable performance increase as compared to an application that does not use stored functions.

Also, with PL/pgSQL you can use all the data types, operators and functions of SQL.
38.1.2. Supported Argument and Result Data Types

Functions written in PL/pgSQL can accept as arguments any scalar or array data type supported by the server, and they can return a result of any of these types. They can also accept or return any composite type (row type) specified by name. It is also possible to declare a PL/pgSQL function as returning record, which means that the result is a row type whose columns are determined by specification in the calling query, as discussed in Section 7.2.1.4.

PL/pgSQL functions can also be declared to accept and return the polymorphic types arelement, anyarray, anynonarray, and anyenum. The actual data types handled by a polymorphic function can vary from call to call, as discussed in Section 34.2.5. An example is shown in Section 38.3.1.

PL/pgSQL functions can also be declared to return a “set” (or table) of any data type that can be returned as a single instance. Such a function generates its output by executing RETURN NEXT for each desired element of the result set, or by using RETURN QUERY to output the result of evaluating a query.

Finally, a PL/pgSQL function can be declared to return void if it has no useful return value.

PL/pgSQL functions can also be declared with output parameters in place of an explicit specification of the return type. This does not add any fundamental capability to the language, but it is often convenient, especially for returning multiple values.

Specific examples appear in Section 38.3.1 and Section 38.6.1.

38.2. Structure of PL/pgSQL

PL/pgSQL is a block-structured language. The complete text of a function definition must be a block. A block is defined as:

```
[ [<<label>>] ]
[ DECLARE declarations ]
BEGIN
 statements
END [ [ label ] ];
```

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after END, as shown above; however the final END that concludes a function body does not require a semicolon.

**Tip:** A common mistake is to write a semicolon immediately after BEGIN. This is incorrect and will result in a syntax error.

A *label* is only needed if you want to identify the block for use in an EXIT statement, or to qualify the names of the variables declared in the block. If a label is given after END, it must match the label at the block’s beginning.

All key words are case-insensitive. Identifiers are implicitly converted to lowercase unless double-quoted, just as they are in ordinary SQL commands.

There are two types of comments in PL/pgSQL. A double dash (--) starts a comment that extends to the end of the line. A /* starts a block comment that extends to the next occurrence of */. Block
comments cannot be nested, but double dash comments can be enclosed into a block comment and a
double dash can hide the block comment delimiters /* and */.

Any statement in the statement section of a block can be a subblock. Subblocks can be used for logical
grouping or to localize variables to a small group of statements. Variables declared in a subblock mask
any similarly-named variables of outer blocks for the duration of the subblock; but you can access the
outer variables anyway if you qualify their names with their block’s label. For example:

```plpgsql
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
  quantity integer := 30;
BEGIN
  RAISE NOTICE 'Quantity here is %', quantity; -- Prints 30
  quantity := 50;
  --
  -- Create a subblock
  --
  DECLARE
    quantity integer := 80;
  BEGIN
    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80
    RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints 50
  END;
  RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50

RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

**Note:** There is actually a hidden “outer block” surrounding the body of any PL/pgSQL function.
This block provides the declarations of the function’s parameters (if any), as well as some special
variables such as `FOUND` (see Section 38.5.5). The outer block is labeled with the function’s name,
meaning that parameters and special variables can be qualified with the function’s name.

It is important not to confuse the use of `BEGIN/END` for grouping statements in PL/pgSQL with the
similarly-named SQL commands for transaction control. PL/pgSQL’s `BEGIN/END` are only for group-
ing; they do not start or end a transaction. Functions and trigger procedures are always executed within
a transaction established by an outer query — they cannot start or commit that transaction, since there
would be no context for them to execute in. However, a block containing an `EXCEPTION` clause effec-
tively forms a subtransaction that can be rolled back without affecting the outer transaction. For more
about that see Section 38.6.5.

### 38.3. Declarations

All variables used in a block must be declared in the declarations section of the block. (The only
exception is that the loop variable of a `FOR` loop iterating over a range of integer values is automatically
declared as an integer variable.)

PL/pgSQL variables can have any SQL data type, such as `integer`, `varchar`, and `char`. 
Chapter 38. PL/pgSQL - SQL Procedural Language

Here are some examples of variable declarations:

user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
aro RECORD;

The general syntax of a variable declaration is:


The DEFAULT clause, if given, specifies the initial value assigned to the variable when the block is entered. If the DEFAULT clause is not given then the variable is initialized to the SQL null value. The CONSTANT option prevents the variable from being assigned to, so that its value remains constant for the duration of the block. If NOT NULL is specified, an assignment of a null value results in a run-time error. All variables declared as NOT NULL must have a nonnull default value specified.

A variable’s default value is evaluated and assigned to the variable each time the block is entered (not just once per function call). So, for example, assigning now() to a variable of type timestamp causes the variable to have the time of the current function call, not the time when the function was precompiled.

Examples:

quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;

38.3.1. Aliases for Function Parameters

Parameters passed to functions are named with the identifiers $1, $2, etc. Optionally, aliases can be declared for $n parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.

There are two ways to create an alias. The preferred way is to give a name to the parameter in the CREATE FUNCTION command, for example:

CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;

The other way, which was the only way available before PostgreSQL 8.0, is to explicitly declare an alias, using the declaration syntax

name ALIAS FOR $n;

The same example in this style looks like:

CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
Chapter 38. PL/pgSQL - SQL Procedural Language

subtotal ALIAS FOR $1;
BEGIN
RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;

Note: These two examples are not perfectly equivalent. In the first case, subtotal could be referenced as sales_tax.subtotal, but in the second case it could not. (Had we attached a label to the block, subtotal could be qualified with that label, instead.)

Some more examples:

CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
  v_string ALIAS FOR $1;
  index ALIAS FOR $2;
BEGIN
  -- some computations using v_string and index here
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;

When a PL/pgSQL function is declared with output parameters, the output parameters are given names and optional aliases in just the same way as the normal input parameters. An output parameter is effectively a variable that starts out NULL; it should be assigned to during the execution of the function. The final value of the parameter is what is returned. For instance, the sales-tax example could also be done this way:

CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
  tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;

Notice that we omitted RETURNS real — we could have included it, but it would be redundant.

Output parameters are most useful when returning multiple values. A trivial example is:

CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
  sum := x + y;
  prod := x * y;
END;
$$ LANGUAGE plpgsql;

As discussed in Section 34.4.3, this effectively creates an anonymous record type for the function’s results. If a RETURNS clause is given, it must say RETURNS record.
When the return type of a PL/pgSQL function is declared as a polymorphic type (anycolumn, anyarray, anynonarray, or anyenum), a special parameter $0$ is created. Its data type is the actual return type of the function, as deduced from the actual input types (see Section 34.2.5). This allows the function to access its actual return type as shown in Section 38.3.2. $0$ is initialized to null and can be modified by the function, so it can be used to hold the return value if desired, though that is not required. $0$ can also be given an alias. For example, this function works on any data type that has a + operator:

```sql
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

The same effect can be had by declaring one or more output parameters as polymorphic types. In this case the special $0$ parameter is not used; the output parameters themselves serve the same purpose. For example:

```sql
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
    OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

### 38.3.2. Copying Types

**variable%TYPE**

%TYPE provides the data type of a variable or table column. You can use this to declare variables that will hold database values. For example, let’s say you have a column named `user_id` in your `users` table. To declare a variable with the same data type as `users.user_id` you write:

```sql
user_id users.user_id%TYPE;
```

By using %TYPE you don’t need to know the data type of the structure you are referencing, and most importantly, if the data type of the referenced item changes in the future (for instance: you change the type of `user_id` from `integer` to `real`), you might not need to change your function definition.

%TYPE is particularly valuable in polymorphic functions, since the data types needed for internal variables can change from one call to the next. Appropriate variables can be created by applying %TYPE to the function’s arguments or result placeholders.
38.3.3. Row Types

A variable of a composite type is called a row variable (or row-type variable). Such a variable can hold a whole row of a SELECT or FOR query result, so long as that query’s column set matches the declared type of the variable. The individual fields of the row value are accessed using the usual dot notation, for example rowvar.field.

A row variable can be declared to have the same type as the rows of an existing table or view, by using the table_name%ROWTYPE notation; or it can be declared by giving a composite type’s name. (Since every table has an associated composite type of the same name, it actually does not matter in PostgreSQL whether you write %ROWTYPE or not. But the form with %ROWTYPE is more portable.)

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a row variable, and fields can be selected from it, for example $1.user_id.

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a row variable, and fields can be selected from it, for example $1.user_id.

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a row variable, and fields can be selected from it, for example $1.user_id.

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a row variable, and fields can be selected from it, for example $1.user_id.

Here is an example of using composite types. table1 and table2 are existing tables having at least the mentioned fields:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

38.3.4. Record Types

Record variables are similar to row-type variables, but they have no predefined structure. They take on the actual row structure of the row they are assigned during a SELECT or FOR command. The substructure of a record variable can change each time it is assigned to. A consequence of this is that until a record variable is first assigned to, it has no substructure, and any attempt to access a field in it will draw a run-time error.

Note that RECORD is not a true data type, only a placeholder. One should also realize that when a PL/pgSQL function is declared to return type record, this is not quite the same concept as a record variable, even though such a function might use a record variable to hold its result. In both cases the actual row structure is unknown when the function is written, but for a function returning record the actual structure is determined when the calling query is parsed, whereas a record variable can change its row structure on-the-fly.
38.3.5. RENAME

RENAME oldname TO newname;

Using the RENAME declaration you can change the name of a variable, record or row. This is primarily useful if NEW or OLD should be referenced by another name inside a trigger procedure. See also ALIAS.

Examples:

RENAME id TO user_id;
RENAME this_var TO that_var;

Note: RENAME appears to be broken as of PostgreSQL 7.3. Fixing this is of low priority, since ALIAS covers most of the practical uses of RENAME.

38.4. Expressions

All expressions used in PL/pgSQL statements are processed using the server’s main SQL executor. For example, when you write a PL/pgSQL statement like

IF expression THEN ...

PL/pgSQL will evaluate the expression by feeding a query like

SELECT expression

to the main SQL engine. While forming the SELECT command, any occurrences of PL/pgSQL variable names are replaced by parameters, as discussed in detail in Section 38.10.1. This allows the query plan for the SELECT to be prepared just once and then reused for subsequent evaluations with different values of the variables. Thus, what really happens on first use of an expression is essentially a PREPARE command. For example, if we have declared two integer variables \( x \) and \( y \), and we write

IF \( x < y \) THEN ...

what happens behind the scenes is

PREPARE statement_name(integer, integer) AS SELECT $1 < $2;

and then this prepared statement is EXECUTED for each execution of the IF statement, with the current values of the PL/pgSQL variables supplied as parameter values. The query plan prepared in this way is saved for the life of the database connection, as described in Section 38.10.2. Normally these details are not important to a PL/pgSQL user, but they are useful to know when trying to diagnose a problem.

38.5. Basic Statements

In this section and the following ones, we describe all the statement types that are explicitly understood by PL/pgSQL. Anything not recognized as one of these statement types is presumed to be an
SQL command and is sent to the main database engine to execute, as described in Section 38.5.2 and Section 38.5.3.

### 38.5.1. Assignment

An assignment of a value to a PL/pgSQL variable or row/record field is written as:

```sql
variable := expression;
```

As explained above, the expression in such a statement is evaluated by means of an SQL `SELECT` command sent to the main database engine. The expression must yield a single value.

If the expression’s result data type doesn’t match the variable’s data type, or the variable has a specific size/precision (like `char(20)`), the result value will be implicitly converted by the PL/pgSQL interpreter using the result type’s output-function and the variable type’s input-function. Note that this could potentially result in run-time errors generated by the input function, if the string form of the result value is not acceptable to the input function.

Examples:

```sql
tax := subtotal * 0.06;
my_record.user_id := 20;
```

### 38.5.2. Executing a Command With No Result

For any SQL command that does not return rows, for example `INSERT` without a `RETURNING` clause, you can execute the command within a PL/pgSQL function just by writing the command.

Any PL/pgSQL variable name appearing in the command text is replaced by a parameter symbol, and then the current value of the variable is provided as the parameter value at run time. This is exactly like the processing described earlier for expressions; for details see Section 38.10.1. As an example, if you write:

```sql
DECLARE
  key TEXT;
  delta INTEGER;
BEGIN
  ...
  UPDATE mytab SET val = val + delta WHERE id = key;
```

the command text seen by the main SQL engine will look like:

```sql
UPDATE mytab SET val = val + $1 WHERE id = $2;
```

Although you don’t normally have to think about this, it’s helpful to know it when you need to make sense of syntax-error messages.
Caution

PL/pgSQL will substitute for any identifier matching one of the function’s declared variables; it is not bright enough to know whether that’s what you meant! Thus, it is a bad idea to use a variable name that is the same as any table, column, or function name that you need to reference in commands within the function. For more discussion see Section 38.10.1.

When executing a SQL command in this way, PL/pgSQL plans the command just once and re-uses the plan on subsequent executions, for the life of the database connection. The implications of this are discussed in detail in Section 38.10.2.

Sometimes it is useful to evaluate an expression or SELECT query but discard the result, for example when calling a function that has side-effects but no useful result value. To do this in PL/pgSQL, use the PERFORM statement:

```
PERFORM query;
```

This executes `query` and discards the result. Write the `query` the same way you would write an SQL SELECT command, but replace the initial keyword SELECT with PERFORM. PL/pgSQL variables will be substituted into the query just as for commands that return no result, and the plan is cached in the same way. Also, the special variable FOUND is set to true if the query produced at least one row, or false if it produced no rows (see Section 38.5.5).

**Note:** One might expect that writing SELECT directly would accomplish this result, but at present the only accepted way to do it is PERFORM. A SQL command that can return rows, such as SELECT, will be rejected as an error unless it has an INTO clause as discussed in the next section.

An example:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

### 38.5.3. Executing a Query with a Single-Row Result

The result of a SQL command yielding a single row (possibly of multiple columns) can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an INTO clause. For example,

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

where `target` can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields. PL/pgSQL variables will be substituted into the rest of the query, and the plan is cached, just as described above for commands that do not return rows. This works for SELECT, INSERT/UPDATE/DELETE with RETURNING, and utility commands that return row-set results (such as EXPLAIN). Except for the INTO clause, the SQL command is the same as it would be written outside PL/pgSQL.
Tip: Note that this interpretation of `SELECT with INTO` is quite different from PostgreSQL's regular `SELECT INTO` command, wherein the INTO target is a newly created table. If you want to create a table from a SELECT result inside a PL/pgSQL function, use the syntax `CREATE TABLE ... AS SELECT`.

If a row or a variable list is used as target, the query’s result columns must exactly match the structure of the target as to number and data types, or a run-time error occurs. When a record variable is the target, it automatically configures itself to the row type of the query result columns.

The INTO clause can appear almost anywhere in the SQL command. Customarily it is written either just before or just after the list of `select_expressions` in a `SELECT` command, or at the end of the command for other command types. It is recommended that you follow this convention in case the PL/pgSQL parser becomes stricter in future versions.

If `STRICT` is not specified in the INTO clause, then `target` will be set to the first row returned by the query, or to nulls if the query returned no rows. (Note that “the first row” is not well-defined unless you’ve used `ORDER BY`.) Any result rows after the first row are discarded. You can check the special `FOUND` variable (see Section 38.5.5) to determine whether a row was returned:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

If the `STRICT` option is specified, the query must return exactly one row or a run-time error will be reported, either `NO_DATA_FOUND` (no rows) or `TOO_MANY_ROWS` (more than one row). You can use an exception block if you wish to catch the error, for example:

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE EXCEPTION 'employee % not found', myname;
    WHEN TOO_MANY_ROWS THEN
        RAISE EXCEPTION 'employee % not unique', myname;
END;
```

Successful execution of a command with `STRICT` always sets `FOUND` to true.

For `INSERT/UPDATE/DELETE` with `RETURNING`, PL/pgSQL reports an error for more than one returned row, even when `STRICT` is not specified. This is because there is no option such as `ORDER BY` with which to determine which affected row should be returned.

Note: The `STRICT` option matches the behavior of Oracle PL/SQL's `SELECT INTO` and related statements.

To handle cases where you need to process multiple result rows from a SQL query, see Section 38.6.4.

### 38.5.4. Executing Dynamic Commands

Oftentimes you will want to generate dynamic commands inside your PL/pgSQL functions, that is, commands that will involve different tables or different data types each time they are executed.
Chapter 38. PL/pgSQL - SQL Procedural Language

PL/pgSQL’s normal attempts to cache plans for commands (as discussed in Section 38.10.2) will not work in such scenarios. To handle this sort of problem, the EXECUTE statement is provided:

```sql
EXECUTE command-string [ INTO [STRICT] target ];
```

where `command-string` is an expression yielding a string (of type `text`) containing the command to be executed and `target` is a record variable, row variable, or a comma-separated list of simple variables and record/row fields.

No substitution of PL/pgSQL variables is done on the computed command string. Any required variable values must be inserted in the command string as it is constructed.

Also, there is no plan caching for commands executed via EXECUTE. Instead, the command is prepared each time the statement is run. Thus the command string can be dynamically created within the function to perform actions on different tables and columns.

The `INTO` clause specifies where the results of a SQL command returning rows should be assigned. If a row or variable list is provided, it must exactly match the structure of the query’s results (when a record variable is used, it will configure itself to match the result structure automatically). If multiple rows are returned, only the first will be assigned to the `INTO` variable. If no rows are returned, NULL is assigned to the `INTO` variable(s). If no `INTO` clause is specified, the query results are discarded.

If the `STRICT` option is given, an error is reported unless the query produces exactly one row.

SELECT INTO is not currently supported within EXECUTE; instead, execute a plain SELECT command and specify INTO as part of the EXECUTE itself.

**Note:** The PL/pgSQL EXECUTE statement is not related to the EXECUTE SQL statement supported by the PostgreSQL server. The server’s EXECUTE statement cannot be used directly within PL/pgSQL functions (and is not needed).

When working with dynamic commands you will often have to handle escaping of single quotes. The recommended method for quoting fixed text in your function body is dollar quoting. (If you have legacy code that does not use dollar quoting, please refer to the overview in Section 38.11.1, which can save you some effort when translating said code to a more reasonable scheme.)

Dynamic values that are to be inserted into the constructed query require careful handling since they might themselves contain quote characters. An example (this assumes that you are using dollar quoting for the function as a whole, so the quote marks need not be doubled):

```sql
EXECUTE 'UPDATE tbl SET '
  || quote_ident(colname)
  || ' = ' || quote_literal(newvalue)
  || ' WHERE key = ' || quote_literal(keyvalue);
```

This example demonstrates the use of the `quote_ident` and `quote_literal` functions. For safety, expressions containing column and table identifiers should be passed to `quote_ident`. Expressions containing values that should be literal strings in the constructed command should be passed to `quote_literal`. Both take the appropriate steps to return the input text enclosed in double or single quotes respectively, with any embedded special characters properly escaped.

Note that dollar quoting is only useful for quoting fixed text. It would be a very bad idea to try to do the above example as:
EXECUTE 'UPDATE tbl SET ' || quote_ident(colname) || ' = $$' || newvalue || '$$ WHERE key = ' || quote_literal(keyvalue);  

because it would break if the contents of newvalue happened to contain $$$. The same objection would apply to any other dollar-quoting delimiter you might pick. So, to safely quote text that is not known in advance, you must use quote_literal.

A much larger example of a dynamic command and EXECUTE can be seen in Example 38-6, which builds and executes a CREATE FUNCTION command to define a new function.

### 38.5.5. Obtaining the Result Status

There are several ways to determine the effect of a command. The first method is to use the GET DIAGNOSTICS command, which has the form:

```
GET DIAGNOSTICS variable = item [ , ... ];
```

This command allows retrieval of system status indicators. Each item is a key word identifying a state value to be assigned to the specified variable (which should be of the right data type to receive it). The currently available status items are ROW_COUNT, the number of rows processed by the last SQL command sent down to the SQL engine, and RESULT_OID, the OID of the last row inserted by the most recent SQL command. Note that RESULT_OID is only useful after an INSERT command into a table containing OIDs.

An example:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

The second method to determine the effects of a command is to check the special variable named FOUND, which is of type boolean. FOUND starts out false within each PL/pgSQL function call. It is set by each of the following types of statements:

- A SELECT INTO statement sets FOUND true if a row is assigned, false if no row is returned.
- A PERFORM statement sets FOUND true if it produces (and discards) one or more rows, false if no row is produced.
- UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected.
- A FETCH statement sets FOUND true if it returns a row, false if no row is returned.
- A MOVE statement sets FOUND true if it successfully repositions the cursor, false otherwise.
- A FOR statement sets FOUND true if it iterates one or more times, else false. This applies to all three variants of the FOR statement (integer FOR loops, record-set FOR loops, and dynamic record-set FOR loops). FOUND is set this way when the FOR loop exits; inside the execution of the loop, FOUND is not modified by the FOR statement, although it might be changed by the execution of other statements within the loop body.

FOUND is a local variable within each PL/pgSQL function; any changes to it affect only the current function.
38.5.6. Doing Nothing At All

Sometimes a placeholder statement that does nothing is useful. For example, it can indicate that one arm of an if/then/else chain is deliberately empty. For this purpose, use the `NULL` statement:

```
NULL;
```

For example, the following two fragments of code are equivalent:

```
BEGIN
  y := x / 0;
EXCEPTION
  WHEN division_by_zero THEN
    NULL; -- ignore the error
END;

BEGIN
  y := x / 0;
EXCEPTION
  WHEN division_by_zero THEN -- ignore the error
END;
```

Which is preferable is a matter of taste.

**Note:** In Oracle’s PL/SQL, empty statement lists are not allowed, and so `NULL` statements are *required* for situations such as this. PL/pgSQL allows you to just write nothing, instead.

38.6. Control Structures

Control structures are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL’s control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

38.6.1. Returning From a Function

There are two commands available that allow you to return data from a function: `RETURN` and `RETURN NEXT`.

38.6.1.1. RETURN

```
RETURN expression;
```

`RETURN` with an expression terminates the function and returns the value of `expression` to the caller. This form is to be used for PL/pgSQL functions that do not return a set.

When returning a scalar type, any expression can be used. The expression’s result will be automatically cast into the function’s return type as described for assignments. To return a composite (row) value, you must write a record or row variable as the `expression`. 
Chapter 38. PL/pgSQL - SQL Procedural Language

If you declared the function with output parameters, write just `RETURN` with no expression. The current values of the output parameter variables will be returned.

If you declared the function to return `void`, a `RETURN` statement can be used to exit the function early; but do not write an expression following `RETURN`.

The return value of a function cannot be left undefined. If control reaches the end of the top-level block of the function without hitting a `RETURN` statement, a run-time error will occur. This restriction does not apply to functions with output parameters and functions returning `void`, however. In those cases a `RETURN` statement is automatically executed if the top-level block finishes.

### 38.6.1.2. RETURN NEXT and RETURN QUERY

```sql
RETURN NEXT expression;
RETURN QUERY query;
```

When a PL/pgSQL function is declared to return `SETOF sometype`, the procedure to follow is slightly different. In that case, the individual items to return are specified by a sequence of `RETURN NEXT` or `RETURN QUERY` commands, and then a final `RETURN` command with no argument is used to indicate that the function has finished executing. `RETURN NEXT` can be used with both scalar and composite data types; with a composite result type, an entire “table” of results will be returned. `RETURN QUERY` appends the results of executing a query to the function’s result set. `RETURN NEXT` and `RETURN QUERY` can be freely intermixed in a single set-returning function, in which case their results will be concatenated.

`RETURN NEXT` and `RETURN QUERY` do not actually return from the function — they simply append zero or more rows to the function’s result set. Execution then continues with the next statement in the PL/pgSQL function. As successive `RETURN NEXT` or `RETURN QUERY` commands are executed, the result set is built up. A final `RETURN`, which should have no argument, causes control to exit the function (or you can just let control reach the end of the function).

If you declared the function with output parameters, write just `RETURN NEXT` with no expression. On each execution, the current values of the output parameter variable(s) will be saved for eventual return as a row of the result. Note that you must declare the function as returning `SETOF record` when there are multiple output parameters, or `SETOF sometype` when there is just one output parameter of type `sometype`, in order to create a set-returning function with output parameters.

Here is an example of a function using `RETURN NEXT`:

```sql
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS $BODY$
DECLARE
  r foo%rowtype;
BEGIN
  FOR r IN SELECT * FROM foo
  WHERE fooid > 0
  LOOP
    -- can do some processing here
    RETURN NEXT r; -- return current row of SELECT
  END LOOP;
RETURN;
END
$BODY$
```

---

...
Chapter 38. PL/pgSQL - SQL Procedural Language

Note that functions using `RETURN NEXT` or `RETURN QUERY` must be called as a table source in a `FROM` clause.

Note: The current implementation of `RETURN NEXT` and `RETURN QUERY` stores the entire result set before returning from the function, as discussed above. That means that if a PL/pgSQL function produces a very large result set, performance might be poor: data will be written to disk to avoid memory exhaustion, but the function itself will not return until the entire result set has been generated. A future version of PL/pgSQL might allow users to define set-returning functions that do not have this limitation. Currently, the point at which data begins being written to disk is controlled by the `work_mem` configuration variable. Administrators who have sufficient memory to store larger result sets in memory should consider increasing this parameter.

38.6.2. Conditionals

`IF` statements let you execute commands based on certain conditions. PL/pgSQL has five forms of `IF`:

• `IF ... THEN`
• `IF ... THEN ... ELSE`
• `IF ... THEN ... ELSE IF`
• `IF ... THEN ... ELSIF ... THEN ... ELSE`
• `IF ... THEN ... ELSEIF ... THEN ... ELSE`

38.6.2.1. IF–THEN

`IF `boolean-expression` THEN
  statements
END IF;`

IF–THEN statements are the simplest form of `IF`. The statements between `THEN` and `END IF` will be executed if the condition is true. Otherwise, they are skipped.

Example:

```sql
IF v_user_id <> 0 THEN
  UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```
38.6.2.2. IF-THEN-ELSE

IF boolean-expression THEN
    statements
ELSE
    statements
END IF;

IF-THEN-ELSE statements add to IF-THEN by letting you specify an alternative set of statements that should be executed if the condition evaluates to false.

Examples:

IF parentid IS NULL OR parentid = "" THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;

IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;

38.6.2.3. IF-THEN-ELSE IF

IF statements can be nested, as in the following example:

IF demo_row.sex = 'm' THEN
    prettySex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        prettySex := 'woman';
    END IF;
END IF;

When you use this form, you are actually nesting an IF statement inside the ELSE part of an outer IF statement. Thus you need one END IF statement for each nested IF and one for the parent IF-ELSE. This is workable but grows tedious when there are many alternatives to be checked. Hence the next form.

38.6.2.4. IF-THEN-ELSIF-ELSE

IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements

IF-THEN-ELSEIF-ELSE provides a more convenient method of checking many alternatives in one statement. Functionally it is equivalent to nested IF-THEN-ELSE-IF-THEN commands, but only one END IF is needed.

Here is an example:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```

38.6.2.5. IF-THEN-ELSEIF-ELSE

ELSEIF is an alias for ELSIF.

38.6.3. Simple Loops

With the LOOP, EXIT, CONTINUE, WHILE, and FOR statements, you can arrange for your PL/pgSQL function to repeat a series of commands.

38.6.3.1. LOOP

```
[ <<<label>>> ]
LOOP
    statements
END LOOP [ label ];
```

LOOP defines an unconditional loop that is repeated indefinitely until terminated by an EXIT or RETURN statement. The optional label can be used by EXIT and CONTINUE statements in nested loops to specify which loop the statement should be applied to.

38.6.3.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

If no label is given, the innermost loop is terminated and the statement following END LOOP is executed next. If label is given, it must be the label of the current or some outer level of nested loop.
or block. Then the named loop or block is terminated and control continues with the statement after the loop's/block's corresponding END.

If WHEN is specified, the loop exit occurs only if boolean-expression is true. Otherwise, control passes to the statement after EXIT.

EXIT can be used with all types of loops; it is not limited to use with unconditional loops. When used with a BEGIN block, EXIT passes control to the next statement after the end of the block.

Examples:

```sql
LOOP
  -- some computations
  IF count > 0 THEN
    EXIT;  -- exit loop
  END IF;
END LOOP;

LOOP
  -- some computations
  EXIT WHEN count > 0;  -- same result as previous example
END LOOP;

BEGIN
  -- some computations
  IF stocks > 100000 THEN
    EXIT;  -- causes exit from the BEGIN block
  END IF;
END;
```

38.6.3.3. CONTINUE

CONTINUE [ label ] [ WHEN boolean-expression ];

If no label is given, the next iteration of the innermost loop is begun. That is, all statements remaining in the loop body are skipped, and control returns to the loop control expression (if any) to determine whether another loop iteration is needed. If label is present, it specifies the label of the loop whose execution will be continued.

If WHEN is specified, the next iteration of the loop is begun only if boolean-expression is true. Otherwise, control passes to the statement after CONTINUE.

CONTINUE can be used with all types of loops; it is not limited to use with unconditional loops.

Examples:

```sql
LOOP
  -- some computations
  EXIT WHEN count > 100;
  CONTINUE WHEN count < 50;
  -- some computations for count IN [50 .. 100]
END LOOP;
```
### 38.6.3.4. WHILE

```sql
[ <<{label}>> ]
WHILE boolean-expression LOOP
  statements
END LOOP [ {label} ];
```

The **WHILE** statement repeats a sequence of statements so long as the `boolean-expression` evaluates to true. The expression is checked just before each entry to the loop body.

For example:

```sql
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
  -- some computations here
END LOOP;
WHILE NOT done LOOP
  -- some computations here
END LOOP;
```

### 38.6.3.5. FOR (integer variant)

```sql
[ <<{label}>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
  statements
END LOOP [ {label} ];
```

This form of **FOR** creates a loop that iterates over a range of integer values. The variable `name` is automatically defined as type `integer` and exists only inside the loop (any existing definition of the variable name is ignored within the loop). The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the `BY` clause isn’t specified the iteration step is 1, otherwise it’s the value specified in the `BY` clause, which again is evaluated once on loop entry.

If `REVERSE` is specified then the step value is subtracted, rather than added, after each iteration.

Some examples of integer **FOR** loops:

```sql
FOR i IN 1..10 LOOP
  -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;
FOR i IN REVERSE 10..1 LOOP
  -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;
FOR i IN REVERSE 10..1 BY 2 LOOP
  -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

If the lower bound is greater than the upper bound (or less than, in the `REVERSE` case), the loop body is not executed at all. No error is raised.

If a `label` is attached to the **FOR** loop then the integer loop variable can be referenced with a qualified name, using that `label`. 

742
### 38.6.4. Looping Through Query Results

Using a different type of `FOR` loop, you can iterate through the results of a query and manipulate that data accordingly. The syntax is:

```sql
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

The `target` is a record variable, row variable, or comma-separated list of scalar variables. The `target` is successively assigned each row resulting from the `query` and the loop body is executed for each row. Here is an example:

```sql
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    PERFORM cs_log('Refreshing materialized views...');
    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP
        -- Now "mviews" has one record from cs_materialized_views
        PERFORM cs_log('Refreshing materialized view ' || quote_ident(mviews.mv_name) || ' ...');
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO ' || quote_ident(mviews.mv_name) || ' ' || mviews.mv_query;
    END LOOP;
    PERFORM cs_log('Done refreshing materialized views.');
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

If the loop is terminated by an `EXIT` statement, the last assigned row value is still accessible after the loop.

The `query` used in this type of `FOR` statement can be any SQL command that returns rows to the caller: `SELECT` is the most common case, but you can also use `INSERT`, `UPDATE`, or `DELETE` with a `RETURNING` clause. Some utility commands such as `EXPLAIN` will work too.

`PL/pgSQL` variables are substituted into the query text, and the query plan is cached for possible re-use, as discussed in detail in Section 38.10.1 and Section 38.10.2.

The `FOR-IN-EXECUTE` statement is another way to iterate over rows:

```sql
[ <<label>> ]
FOR target IN EXECUTE text_expression LOOP
    statements
END LOOP [ label ];
```

This is like the previous form, except that the source query is specified as a string expression, which is evaluated and replanned on each entry to the `FOR` loop. This allows the programmer to choose the speed of a preplanned query or the flexibility of a dynamic query, just as with a plain `EXECUTE` statement.
38.6.5. Trapping Errors

By default, any error occurring in a PL/pgSQL function aborts execution of the function, and indeed of the surrounding transaction as well. You can trap errors and recover from them by using a `BEGIN` block with an `EXCEPTION` clause. The syntax is an extension of the normal syntax for a `BEGIN` block:

```sql
[ <<label>> ]
[ DECLARE declarations ]
BEGIN
  statements
EXCEPTION
  WHEN condition [ OR condition ... ] THEN
    handler_statements
  [ WHEN condition [ OR condition ... ] THEN
    handler_statements
  ... ]
END;
```

If no error occurs, this form of block simply executes all the `statements`, and then control passes to the next statement after `END`. But if an error occurs within the `statements`, further processing of the `statements` is abandoned, and control passes to the `EXCEPTION` list. The list is searched for the first `condition` matching the error that occurred. If a match is found, the corresponding `handler_statements` are executed, and then control passes to the next statement after `END`. If no match is found, the error propagates out as though the `EXCEPTION` clause were not there at all: the error can be caught by an enclosing block with `EXCEPTION`, or if there is none it aborts processing of the function.

The `condition` names can be any of those shown in Appendix A. A category name matches any error within its category. The special condition name `OTHERS` matches every error type except `QUERY_CANCELED`. (It is possible, but often unwise, to trap `QUERY_CANCELED` by name.) Condition names are not case-sensitive.

If a new error occurs within the selected `handler_statements`, it cannot be caught by this `EXCEPTION` clause, but is propagated out. A surrounding `EXCEPTION` clause could catch it.

When an error is caught by an `EXCEPTION` clause, the local variables of the PL/pgSQL function remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back. As an example, consider this fragment:

```sql
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
  UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
  x := x + 1;
  y := x / 0;
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'caught division_by_zero';
    RETURN x;
END;
```

When control reaches the assignment to `y`, it will fail with a `division_by_zero` error. This will be caught by the `EXCEPTION` clause. The value returned in the `RETURN` statement will be the incremented value of `x`, but the effects of the `UPDATE` command will have been rolled back. The `INSERT` command preceding the block is not rolled back, however, so the end result is that the database contains Tom Jones not Joe Jones.
Tip: A block containing an `EXCEPTION` clause is significantly more expensive to enter and exit than a block without one. Therefore, don’t use `EXCEPTION` without need.

Within an exception handler, the `SQLSTATE` variable contains the error code that corresponds to the exception that was raised (refer to Table A-1 for a list of possible error codes). The `SQLERRM` variable contains the error message associated with the exception. These variables are undefined outside exception handlers.

Example 38-1. Exceptions with `UPDATE/INSERT`

This example uses exception handling to perform either `UPDATE` or `INSERT`, as appropriate:

```sql
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);
CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS $$
BEGIN
  LOOP
    -- first try to update the key
    UPDATE db SET b = data WHERE a = key;
    IF found THEN
      RETURN;
    END IF;
    -- not there, so try to insert the key
    -- if someone else inserts the same key concurrently,
    -- we could get a unique-key failure
    BEGIN
      INSERT INTO db(a,b) VALUES (key, data);
      RETURN;
    EXCEPTION WHEN unique_violation THEN
      -- do nothing, and loop to try the UPDATE again
    END;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

38.7. Cursors

Rather than executing a whole query at once, it is possible to set up a cursor that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since `FOR` loops automatically use a cursor internally to avoid memory problems.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.
38.7.1. Declaring Cursor Variables

All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special data type refcursor. One way to create a cursor variable is just to declare it as a variable of type refcursor. Another way is to use the cursor declaration syntax, which in general is:

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

(for can be replaced by IS for Oracle compatibility.) If SCROLL is specified, the cursor will be capable of scrolling backward; if NO SCROLL is specified, backward fetches will be rejected; if neither specification appears, it is query-dependent whether backward fetches will be allowed. arguments, if specified, is a comma-separated list of pairs name datatype that define names to be replaced by parameter values in the given query. The actual values to substitute for these names will be specified later, when the cursor is opened.

Some examples:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

All three of these variables have the data type refcursor, but the first can be used with any query, while the second has a fully specified query already bound to it, and the last has a parameterized query bound to it. (key will be replaced by an integer parameter value when the cursor is opened.) The variable curs1 is said to be unbound since it is not bound to any particular query.

38.7.2. Opening Cursors

Before a cursor can be used to retrieve rows, it must be opened. (This is the equivalent action to the SQL command DECLARE CURSOR.) PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

38.7.2.1. OPEN FOR query

```
OPEN unbound_cursor [ [ NO ] SCROLL ] FOR query;
```

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor (that is, as a simple refcursor variable). The query must be a SELECT, or something else that returns rows (such as EXPLAIN). The query is treated in the same way as other SQL commands in PL/pgSQL: PL/pgSQL variable names are substituted, and the query plan is cached for possible reuse. When a PL/pgSQL variable is substituted into the cursor query, the value that is substituted is the one it has at the time of the OPEN; subsequent changes to the variable will not affect the cursor’s behavior. The SCROLL and NO SCROLL options have the same meanings as for a bound cursor.

An example:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```
38.7.2.2. OPEN FOR EXECUTE

OPEN unbound_cursor [ [ NO ] SCROLL ] FOR EXECUTE query_string;

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor (that is, as a simple refcursor variable). The query is specified as a string expression, in the same way as in the EXECUTE command. As usual, this gives flexibility so the query plan can vary from one run to the next (see Section 38.10.2), and it also means that variable substitution is not done on the command string. The SCROLL and NO SCROLL options have the same meanings as for a bound cursor.

An example:

OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);

38.7.2.3. Opening a Bound Cursor

OPEN bound_cursor [ ( argument_values ) ];

This form of OPEN is used to open a cursor variable whose query was bound to it when it was declared. The cursor cannot be open already. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted in the query. The query plan for a bound cursor is always considered cacheable; there is no equivalent of EXECUTE in this case. Notice that SCROLL and NO SCROLL cannot be specified, as the cursor’s scrolling behavior was already determined.

Note that because variable substitution is done on the bound cursor’s query, there are two ways to pass values into the cursor: either with an explicit argument to OPEN, or implicitly by referencing a PL/pgSQL variable in the query. However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the OPEN.

Examples:

OPEN curs2;
OPEN curs3(42);

38.7.3. Using Cursors

Once a cursor has been opened, it can be manipulated with the statements described here.

These manipulations need not occur in the same function that opened the cursor to begin with. You can return a refcursor value out of a function and let the caller operate on the cursor. (Internally, a refcursor value is simply the string name of a so-called portal containing the active query for the cursor. This name can be passed around, assigned to other refcursor variables, and so on, without disturbing the portal.)

All portals are implicitly closed at transaction end. Therefore a refcursor value is usable to reference an open cursor only until the end of the transaction.
38.7.3.1. FETCH

FETCH [ direction { FROM | IN } ] cursor INTO target;

FETCH retrieves the next row from the cursor into a target, which might be a row variable, a record variable, or a comma-separated list of simple variables, just like SELECT INTO. If there is no next row, the target is set to NULL(s). As with SELECT INTO, the special variable FOUND can be checked to see whether a row was obtained or not.

The direction clause can be any of the variants allowed in the SQL FETCH command except the ones that can fetch more than one row; namely, it can be NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARD, or BACKWARD. Omitting direction is the same as specifying NEXT. direction values that require moving backward are likely to fail unless the cursor was declared or opened with the SCROLL option.

cursor must be the name of a refcursor variable that references an open cursor portal.

Examples:

FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;

38.7.3.2. MOVE

MOVE [ direction { FROM | IN } ] cursor;

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to. As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.

The options for the direction clause are the same as for FETCH, namely NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARD, or BACKWARD. Omitting direction is the same as specifying NEXT. direction values that require moving backward are likely to fail unless the cursor was declared or opened with the SCROLL option.

Examples:

MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;

38.7.3.3. UPDATE/DELETE WHERE CURRENT OF

UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;

When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row. Note that this only works for simple (non-join, non-grouping) cursor queries. For additional information see the DECLARE reference page.
An example:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

### 38.7.3.4. close

```
CLOSE cursor;
```

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

An example:

```
CLOSE curs1;
```

### 38.7.3.5. Returning Cursors

PL/pgSQL functions can return cursors to the caller. This is useful to return multiple rows or columns, especially with very large result sets. To do this, the function opens the cursor and returns the cursor name to the caller (or simply opens the cursor using a portal name specified by or otherwise known to the caller). The caller can then fetch rows from the cursor. The cursor can be closed by the caller, or it will be closed automatically when the transaction closes.

The portal name used for a cursor can be specified by the programmer or automatically generated. To specify a portal name, simply assign a string to the `refcursor` variable before opening it. The string value of the `refcursor` variable will be used by `OPEN` as the name of the underlying portal. However, if the `refcursor` variable is null, `OPEN` automatically generates a name that does not conflict with any existing portal, and assigns it to the `refcursor` variable.

**Note:** A bound cursor variable is initialized to the string value representing its name, so that the portal name is the same as the cursor variable name, unless the programmer overrides it by assignment before opening the cursor. But an unbound cursor variable defaults to the null value initially, so it will receive an automatically-generated unique name, unless overridden.

The following example shows one way a cursor name can be supplied by the caller:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS 'BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funcursor');
FETCH ALL IN funcursor;
```
The following example uses automatic cursor name generation:

```sql
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc2();

reffunc2
-------------------
<unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;

The following example shows one way to return multiple cursors from a single function:

```sql
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;
SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```
38.8. Errors and Messages

Use the RAISE statement to report messages and raise errors.

RAISE level 'format' [, expression [, ...]];

Possible levels are DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION. EXCEPTION raises an error (which normally aborts the current transaction); the other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

Inside the format string, % is replaced by the next optional argument’s string representation. Write %% to emit a literal %. Arguments can be simple variables or expressions, but the format must be a simple string literal.

In this example, the value of v_job_id will replace the % in the string:

RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;

This example will abort the transaction with the given error message:

RAISE EXCEPTION 'Nonexistent ID --> %', user_id;

RAISE EXCEPTION presently always generates the same SQLSTATE code, P0001, no matter what message it is invoked with. It is possible to trap this exception with EXCEPTION ... WHEN RAISE_EXCEPTION THEN ... but there is no way to tell one RAISE from another.

38.9. Trigger Procedures

PL/pgSQL can be used to define trigger procedures. A trigger procedure is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger. Note that the function must be declared with no arguments even if it expects to receive arguments specified in CREATE TRIGGER — trigger arguments are passed via TG_ARGV, as described below.

When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

NEW

Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is NULL in statement-level triggers.

OLD

Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is NULL in statement-level triggers.

TG_NAME

Data type name; variable that contains the name of the trigger actually fired.

TG_WHEN

Data type text; a string of either BEFORE or AFTER depending on the trigger’s definition.
TG_LEVEL
Data type text; a string of either ROW or STATEMENT depending on the trigger’s definition.

TG_OP
Data type text; a string of INSERT, UPDATE, or DELETE telling for which operation the trigger was fired.

TG_RELID
Data type oid; the object ID of the table that caused the trigger invocation.

TG_RELNAME
Data type name; the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use TG_TABLE_NAME instead.

TG_TABLE_NAME
Data type name; the name of the table that caused the trigger invocation.

TG_TABLE_SCHEMA
Data type name; the name of the schema of the table that caused the trigger invocation.

TG_NARGS
Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.

TG_ARGV[]
Data type array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0. Invalid indices (less than 0 or greater than or equal to tg_nargs) result in a null value.

A trigger function must return either NULL or a record/row value having exactly the structure of the table the trigger was fired for.

Row-level triggers fired BEFORE can return null to signal the trigger manager to skip the rest of the operation for this row (i.e., subsequent triggers are not fired, and the INSERT/UPDATE/DELETE does not occur for this row). If a nonnull value is returned then the operation proceeds with that row value. Returning a row value different from the original value of NEW alters the row that will be inserted or updated (but has no direct effect in the DELETE case). To alter the row to be stored, it is possible to replace single values directly in NEW and return the modified NEW, or to build a complete new record/row to return.

The return value of a BEFORE or AFTER statement-level trigger or an AFTER row-level trigger is always ignored; it might as well be null. However, any of these types of triggers might still abort the entire operation by raising an error.

Example 38-2 shows an example of a trigger procedure in PL/pgSQL.

**Example 38-2. A PL/pgSQL Trigger Procedure**

This example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it checks that an employee’s name is given and that the salary is a positive value.

```sql
CREATE TABLE emp (  
    empname text,  
    salary integer,  
    last_date timestamp,
```
Chapter 38. PL/pgSQL - SQL Procedural Language

last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
BEGIN
  -- Check that empname and salary are given
  IF NEW.empname IS NULL THEN
    RAISE EXCEPTION 'empname cannot be null';
  END IF;
  IF NEW.salary IS NULL THEN
    RAISE EXCEPTION '% cannot have null salary', NEW.empname;
  END IF;

  -- Who works for us when she must pay for it?
  IF NEW.salary < 0 THEN
    RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
  END IF;

  -- Remember who changed the payroll when
  NEW.last_date := current_timestamp;
  NEW.last_user := current_user;
  RETURN NEW;
END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
  FOR EACH ROW EXECUTE PROCEDURE emp_stamp();

Another way to log changes to a table involves creating a new table that holds a row for each insert, update, or delete that occurs. This approach can be thought of as auditing changes to a table. Example 38-3 shows an example of an audit trigger procedure in PL/pgSQL.

Example 38-3. A PL/pgSQL Trigger Procedure For Auditing

This example trigger ensures that any insert, update or delete of a row in the emp table is recorded (i.e., audited) in the emp_audit table. The current time and user name are stamped into the row, together with the type of operation performed on it.

CREATE TABLE emp (
  empname text NOT NULL,
  salary integer
);

CREATE TABLE emp_audit(
  operation char(1) NOT NULL,
  stamp timestamp NOT NULL,
  userid text NOT NULL,
  empname text NOT NULL,
  salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
  -- Create a row in emp_audit to reflect the operation performed on emp,
  -- make use of the special variable TG_OP to work out the operation.
  --
Chapter 38. PL/pgSQL - SQL Procedural Language

IF (TG_OP = 'DELETE') THEN
    INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
    RETURN OLD;
ELSIF (TG_OP = 'UPDATE') THEN
    INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
    RETURN NEW;
ELSIF (TG_OP = 'INSERT') THEN
    INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
    RETURN NEW;
END IF;
RETURN NULL; -- result is ignored since this is an AFTER trigger
END;

$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();

One use of triggers is to maintain a summary table of another table. The resulting summary can be used in place of the original table for certain queries — often with vastly reduced run times. This technique is commonly used in Data Warehousing, where the tables of measured or observed data (called fact tables) might be extremely large. Example 38-4 shows an example of a trigger procedure in PL/pgSQL that maintains a summary table for a fact table in a data warehouse.

Example 38-4. A PL/pgSQL Trigger Procedure For Maintaining A Summary Table

The schema detailed here is partly based on the Grocery Store example from The Data Warehouse Toolkit by Ralph Kimball.

```
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (  
time_key integer NOT NULL,
day_of_week integer NOT NULL,
day_of_month integer NOT NULL,
month integer NOT NULL,
quarter integer NOT NULL,
year integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (  
time_key integer NOT NULL,
product_key integer NOT NULL,
store_key integer NOT NULL,
amount_sold numeric(12,2) NOT NULL,
units_sold integer NOT NULL,
amount_cost numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (  
time_key integer NOT NULL,
```

```
CREATE UNINDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);
--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS $maint_sales_summary_bytime$
DECLARE
    delta_time_key integer;
    delta_amount_sold numeric(15,2); 
    delta_units_sold numeric(12);
    delta_amount_cost numeric(15,2);
BEGIN
    -- Work out the increment/decrement amount(s).
    IF (TG_OP = 'DELETE') THEN
        delta_time_key = OLD.time_key;
        delta_amount_sold = -1 * OLD.amount_sold;
        delta_units_sold = -1 * OLD.units_sold;
        delta_amount_cost = -1 * OLD.amount_cost;
    ELSIF (TG_OP = 'UPDATE') THEN
        -- forbid updates that change the time_key -
        -- (probably not too onerous, as DELETE + INSERT is how most
        -- changes will be made).
        IF ( OLD.time_key != NEW.time_key) THEN
            RAISE EXCEPTION 'Update of time_key : % -> % not allowed', OLD.time_key, NEW.time_key;
        END IF;
        delta_time_key = OLD.time_key;
        delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
        delta_units_sold = NEW.units_sold - OLD.units_sold;
        delta_amount_cost = NEW.amount_cost - OLD.amount_cost;
    ELSIF (TG_OP = 'INSERT') THEN
        delta_time_key = NEW.time_key;
        delta_amount_sold = NEW.amount_sold;
        delta_units_sold = NEW.units_sold;
        delta_amount_cost = NEW.amount_cost;
    END IF;

    -- Insert or update the summary row with the new values.
    <<insert_update>>
    LOOP
        UPDATE sales_summary_bytime
        SET amount_sold = amount_sold + delta_amount_sold,
            units_sold = units_sold + delta_units_sold,
            amount_cost = amount_cost + delta_amount_cost
        WHERE time_key = delta_time_key;

END IF;

END}$maint_sales_summary_bytime$;

755
EXIT insert_update WHEN found;

BEGIN
    INSERT INTO sales_summary_bytime (
        time_key,
        amount_sold,
        units_sold,
        amount_cost
    ) VALUES (
        delta_time_key,
        delta_amount_sold,
        delta_units_sold,
        delta_amount_cost
    );

EXIT insert_update;

EXCEPTION
    WHEN UNIQUE_VIOLATION THEN
        -- do nothing
END;
END LOOP insert_update;

RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;

38.10. PL/pgSQL Under the Hood

This section discusses some implementation details that are frequently important for PL/pgSQL users to know.
38.10.1. Variable Substitution

When PL/pgSQL prepares a SQL statement or expression for execution, any PL/pgSQL variable name appearing in the statement or expression is replaced by a parameter symbol, \$n. The current value of the variable is then provided as the value for the parameter whenever the statement or expression is executed. As an example, consider the function

```
CREATE FUNCTION logfunc(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp := now();
    BEGIN
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

The `INSERT` statement will effectively be processed as

```
PREPARE statement_name(text, timestamp) AS
    INSERT INTO logtable VALUES ($1, $2);
```

followed on each execution by `EXECUTE` with the current actual values of the two variables. (Note: here we are speaking of the main SQL engine's `EXECUTE` command, not PL/pgSQL's `EXECUTE`.)

The substitution mechanism will replace any token that matches a known variable's name. This poses various traps for the unwary. For example, it is a bad idea to use a variable name that is the same as any table or column name that you need to reference in queries within the function, because what you think is a table or column name will still get replaced. In the above example, suppose that `logtable` has column names `logtxt` and `logtime`, and we try to write the `INSERT` as

```
    INSERT INTO logtable (logtxt, logtime) VALUES (logtxt, curtime);
```

This will be fed to the main SQL parser as

```
    INSERT INTO logtable ($1, logtime) VALUES ($1, $2);
```

resulting in a syntax error like this:

```
ERROR: syntax error at or near "$1"
LINE 1: INSERT INTO logtable ( $1 , logtime) VALUES ( $1 , $2 ) ^
QUERY: INSERT INTO logtable ( $1 , logtime) VALUES ( $1 , $2 )
CONTEXT: SQL statement in PL/PgSQL function "logfunc2" near line 5
```

This example is fairly easy to diagnose, since it leads to an obvious syntax error. Much nastier are cases where the substitution is syntactically permissible, since the only symptom may be misbehavior of the function. In one case, a user wrote something like this:

```
DECLARE
    val text;
    search_key integer;
BEGIN
    ...
    FOR val IN SELECT val FROM table WHERE key = search_key LOOP ...
```

and wondered why all his table entries seemed to be NULL. Of course what happened here was that the query became

```
DECLARE
    val text;
    search_key integer;
BEGIN
    ...
    FOR val IN SELECT $1 FROM table WHERE key = $1 LOOP ...
```
A commonly used coding rule for avoiding such traps is to use a different naming convention for PL/pgSQL variables than you use for table and column names. For example, if all your variables are named `v_something` while none of your table or column names start with `v_`, you’re pretty safe.

Another workaround is to use qualified (dotted) names for SQL entities. For instance we could safely have written the above example as

```sql
FOR val IN SELECT table.val FROM table WHERE key = search_key LOOP ...
```

because PL/pgSQL will not substitute a variable for a trailing component of a qualified name. However this solution does not work in every case — you can’t qualify a name in an `INSERT`’s column name list, for instance. Another point is that record and row variable names will be matched to the first components of qualified names, so a qualified SQL name is still vulnerable in some cases. In such cases choosing a non-conflicting variable name is the only way.

Another technique you can use is to attach a label to the block in which your variables are declared, and then qualify the variable names in your SQL commands (see Section 38.2). For example,

```plsql
DECLARE
  val text;
BEGIN
  ...
  UPDATE table SET col = pl.val WHERE ...
```

This is not in itself a solution to the problem of conflicts, since an unqualified name in a SQL command is still at risk of being interpreted the “wrong” way. But it is useful for clarifying the intent of potentially-ambiguous code.

Variable substitution does not happen in the command string given to `EXECUTE` or one of its variants. If you need to insert a varying value into such a command, do so as part of constructing the string value, as illustrated in Section 38.5.4.

Variable substitution currently works only in `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands, because the main SQL engine allows parameter symbols only in these commands. To use a non-constant name or value in other statement types (generically called utility statements), you must construct the utility statement as a string and `EXECUTE` it.

### 38.10.2. Plan Caching

The PL/pgSQL interpreter parses the function’s source text and produces an internal binary instruction tree the first time the function is called (within each session). The instruction tree fully translates the PL/pgSQL statement structure, but individual SQL expressions and SQL commands used in the function are not translated immediately.

As each expression and SQL command is first executed in the function, the PL/pgSQL interpreter creates a prepared execution plan (using the SPI manager’s `SPI_prepare` and `SPI_saveplan` functions). Subsequent visits to that expression or command reuse the prepared plan. Thus, a function with conditional code that contains many statements for which execution plans might be required will only prepare and save those plans that are really used during the lifetime of the database connection. This can substantially reduce the total amount of time required to parse and generate execution plans for the statements in a PL/pgSQL function. A disadvantage is that errors in a specific expression or
command cannot be detected until that part of the function is reached in execution. (Trivial syntax
errors will be detected during the initial parsing pass, but anything deeper will not be detected until
execution.)

Once PL/pgSQL has made an execution plan for a particular command in a function, it will reuse that
plan for the life of the database connection. This is usually a win for performance, but it can cause
some problems if you dynamically alter your database schema. For example:

```plaintext
CREATE FUNCTION populate() RETURNS integer AS $$
DECLARE
    -- declarations
BEGIN
    PERFORM my_function();
END;
$$ LANGUAGE plpgsql;
```

If you execute the above function, it will reference the OID for `my_function()` in the execution
plan produced for the `PERFORM` statement. Later, if you drop and recreate `my_function()`, then
`populate()` will not be able to find `my_function()` anymore. You would then have to start a
new database session so that `populate()` will be compiled afresh, before it will work again. You
can avoid this problem by using `CREATE OR REPLACE FUNCTION` when updating the definition of
`my_function`, since when a function is “replaced”, its OID is not changed.

**Note:** In PostgreSQL 8.3 and later, saved plans will be replaced whenever any schema changes
have occurred to any tables they reference. This eliminates one of the major disadvantages of
saved plans. However, there is no such mechanism for function references, and thus the above
example involving a reference to a deleted function is still valid.

Because PL/pgSQL saves execution plans in this way, SQL commands that appear directly in a
PL/pgSQL function must refer to the same tables and columns on every execution; that is, you cannot
use a parameter as the name of a table or column in an SQL command. To get around this restriction,
you can construct dynamic commands using the PL/pgSQL `EXECUTE` statement — at the price of
constructing a new execution plan on every execution.

Another important point is that the prepared plans are parameterized to allow the values of PL/pgSQL
variables to change from one use to the next, as discussed in detail above. Sometimes this means
that a plan is less efficient than it would be if generated for a specific variable value. As an example,
consider

```plaintext
SELECT * INTO myrec FROM dictionary WHERE word LIKE search_term;
```

where `search_term` is a PL/pgSQL variable. The cached plan for this query will never use an index
on `word`, since the planner cannot assume that the `LIKE` pattern will be left-anchored at run time.
To use an index the query must be planned with a specific constant `LIKE` pattern provided. This is
another situation where `EXECUTE` can be used to force a new plan to be generated for each execution.

The mutable nature of record variables presents another problem in this connection. When fields of
a record variable are used in expressions or statements, the data types of the fields must not change
from one call of the function to the next, since each expression will be planned using the data type
that is present when the expression is first reached. `EXECUTE` can be used to get around this problem
when necessary.

If the same function is used as a trigger for more than one table, PL/pgSQL prepares and caches
plans independently for each such table — that is, there is a cache for each trigger function and table
combination, not just for each function. This alleviates some of the problems with varying data types;
for instance, a trigger function will be able to work successfully with a column named `key` even if it happens to have different types in different tables.

Likewise, functions having polymorphic argument types have a separate plan cache for each combination of actual argument types they have been invoked for, so that data type differences do not cause unexpected failures.

Plan caching can sometimes have surprising effects on the interpretation of time-sensitive values. For example there is a difference between what these two functions do:

```sql
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

and:

```sql
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

In the case of `logfunc1`, the PostgreSQL main parser knows when preparing the plan for the `INSERT` that the string `'now'` should be interpreted as `timestamp`, because the target column of `logtable` is of that type. Thus, `'now'` will be converted to a constant when the `INSERT` is planned, and then used in all invocations of `logfunc1` during the lifetime of the session. Needless to say, this isn’t what the programmer wanted.

In the case of `logfunc2`, the PostgreSQL main parser does not know what type `'now'` should become and therefore it returns a data value of type `text` containing the string `now`. During the ensuing assignment to the local variable `curtime`, the PL/pgSQL interpreter casts this string to the `timestamp` type by calling the `text_out` and `timestamp_in` functions for the conversion. So, the computed time stamp is updated on each execution as the programmer expects.

### 38.11. Tips for Developing in PL/pgSQL

One good way to develop in PL/pgSQL is to use the text editor of your choice to create your functions, and in another window, use `psql` to load and test those functions. If you are doing it this way, it is a good idea to write the function using `CREATE OR REPLACE FUNCTION`. That way you can just reload the file to update the function definition. For example:

```sql
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
    ....
$$ LANGUAGE plpgsql;
```

While running `psql`, you can load or reload such a function definition file with:
and then immediately issue SQL commands to test the function.

Another good way to develop in PL/pgSQL is with a GUI database access tool that facilitates development in a procedural language. One example of such a tool is pgAdmin, although others exist. These tools often provide convenient features such as escaping single quotes and making it easier to recreate and debug functions.

### 38.11.1. Handling of Quotation Marks

The code of a PL/pgSQL function is specified in `CREATE FUNCTION` as a string literal. If you write the string literal in the ordinary way with surrounding single quotes, then any single quotes inside the function body must be doubled; likewise any backslashes must be doubled (assuming escape string syntax is used). Doubling quotes is at best tedious, and in more complicated cases the code can become downright incomprehensible, because you can easily find yourself needing half a dozen or more adjacent quote marks. It’s recommended that you instead write the function body as a “dollar-quoting” string literal (see Section 4.1.2.2). In the dollar-quoting approach, you never double any quote marks, but instead take care to choose a different dollar-quoting delimiter for each level of nesting you need. For example, you might write the `CREATE FUNCTION` command as:

```sql
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
    ....
$PROC$ LANGUAGE plpgsql;
```

Within this, you might use quote marks for simple literal strings in SQL commands and $$ to delimit fragments of SQL commands that you are assembling as strings. If you need to quote text that includes $$, you could use $Q$, and so on.

The following chart shows what you have to do when writing quote marks without dollar quoting. It might be useful when translating pre-dollar quoting code into something more comprehensible.

<table>
<thead>
<tr>
<th>Quotation Mark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 quotation mark</td>
<td>To begin and end the function body, for example:</td>
</tr>
<tr>
<td></td>
<td><code>CREATE FUNCTION foo() RETURNS integer AS '.....' LANGUAGE plpgsql;</code> Anywhere within a single-quoted function body, quote marks must appear in pairs.</td>
</tr>
<tr>
<td>2 quotation marks</td>
<td>For string literals inside the function body, for example:</td>
</tr>
<tr>
<td></td>
<td><code>a_output := &quot;Blah&quot;; SELECT * FROM users WHERE f_name=&quot;foobar&quot;;</code> In the dollar-quoting approach, you’d just write:</td>
</tr>
<tr>
<td></td>
<td><code>a_output := 'Blah'; SELECT * FROM users WHERE f_name='foobar';</code> which is exactly what the PL/pgSQL parser would see in either case.</td>
</tr>
<tr>
<td>4 quotation marks</td>
<td>When you need a single quotation mark in a string constant inside the function body, for example:</td>
</tr>
<tr>
<td></td>
<td>`a_output := a_output</td>
</tr>
</tbody>
</table>
In the dollar-quoting approach, you’d write:

```sql
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

being careful that any dollar-quote delimiters around this are not just $$. 6 quotation marks

When a single quotation mark in a string inside the function body is adjacent to the end of that string constant, for example:

```sql
a_output := a_output || " AND name LIKE "'foobar'"
```

The value appended to `a_output` would then be: `AND name LIKE 'foobar'.`

In the dollar-quoting approach, this becomes:

```sql
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 quotation marks

When you want two single quotation marks in a string constant (which accounts for 8 quotation marks) and this is adjacent to the end of that string constant (2 more). You will probably only need that if you are writing a function that generates other functions, as in Example 38-6. For example:

```sql
a_output := a_output || " if v_" ||
    referrer_keys.kind || " like """
    || referrer_keys.key_string || """
then return """ || referrer_keys.referrer_type
    || """; end if;"
```

The value of `a_output` would then be:

```sql
if v_... like "..." then return "..."; end if;
```

In the dollar-quoting approach, this becomes:

```sql
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
then return $$ || referrer_keys.referrer_type
    || $$'; end if;$$;
```

where we assume we only need to put single quote marks into `a_output`, because it will be re-quoted before use.

### 38.12. Porting from Oracle PL/SQL

This section explains differences between PostgreSQL’s PL/pgSQL language and Oracle’s PL/SQL language, to help developers who port applications from Oracle® to PostgreSQL.

PL/pgSQL is similar to PL/SQL in many aspects. It is a block-structured, imperative language, and all variables have to be declared. Assignments, loops, conditionals are similar. The main differences you should keep in mind when porting from PL/SQL to PL/pgSQL are:

- There are no default values for parameters in PostgreSQL.
- You can overload function names in PostgreSQL. This is often used to work around the lack of default parameters.
- If a name used in a SQL command could be either a column name of a table or a reference to a variable of the function, PL/SQL treats it as a column name, while PL/pgSQL treats it as a variable name. It’s best to avoid such ambiguities in the first place, but if necessary you can fix them by properly qualifying the ambiguous name. (See Section 38.10.1.)
• No need for cursors in PL/pgSQL, just put the query in the FOR statement. (See Example 38-6.)
• In PostgreSQL the function body must be written as a string literal. Therefore you need to use dollar quoting or escape single quotes in the function body. (See Section 38.11.1.)
• Instead of packages, use schemas to organize your functions into groups.
• Since there are no packages, there are no package-level variables either. This is somewhat annoying. You can keep per-session state in temporary tables instead.
• Integer FOR loops with REVERSE work differently: PL/SQL counts down from the second number to the first, while PL/pgSQL counts down from the first number to the second, requiring the loop bounds to be swapped when porting. This incompatibility is unfortunate but is unlikely to be changed. (See Section 38.6.3.5.)

38.12.1. Porting Examples

Example 38-5 shows how to port a simple function from PL/SQL to PL/pgSQL.

Example 38-5. Porting a Simple Function from PL/SQL to PL/pgSQL

Here is an Oracle PL/SQL function:

```sql
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
v_version varchar)
RETURN varchar IS
BEGIN
  IF v_version IS NULL THEN
    RETURN v_name;
  END IF;
  RETURN v_name || '/' || v_version;
END;
/
```

Let’s go through this function and see the differences compared to PL/pgSQL:

• The RETURN key word in the function prototype (not the function body) becomes RETURNS in PostgreSQL. Also, IS becomes AS, and you need to add a LANGUAGE clause because PL/pgSQL is not the only possible function language.
• In PostgreSQL, the function body is considered to be a string literal, so you need to use quote marks or dollar quotes around it. This substitutes for the terminating / in the Oracle approach.
• The show errors command does not exist in PostgreSQL, and is not needed since errors are reported automatically.

This is how this function would look when ported to PostgreSQL:

```sql
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
v_version varchar)
RETURNS varchar AS $$
BEGIN
  IF v_version IS NULL THEN
    RETURN v_name;
  END IF;
  RETURN v_name || '/' || v_version;
$$
```

763
Example 38-6 shows how to port a function that creates another function and how to handle the ensuing quoting problems.

**Example 38-6. Porting a Function that Creates Another Function from PL/SQL to PL/pgSQL**

The following procedure grabs rows from a `SELECT` statement and builds a large function with the results in `IF` statements, for the sake of efficiency. Notice particularly the differences in the cursor and the `FOR` loop.

This is the Oracle version:

```sql
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
        v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';
    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
            ' IF v_' || referrer_key.kind
            || ' LIKE ' || quote_literal(referrer_key.key_string)
            || ' THEN RETURN ' || referrer_key.referrer_type
            || ' END IF;';
    END LOOP;
    func_cmd := func_cmd || ' RETURN NULL; END;';
    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

Here is how this function would end up in PostgreSQL:

```sql
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_key RECORD; -- declare a generic record to be used in a FOR
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';
    -- Notice how we scan through the results of a query in a FOR loop
    -- using the FOR <record> construct.
    FOR referrer_key IN SELECT * FROM cs_referrer_keys ORDER BY try_order LOOP
        func_body := func_body ||
            ' IF v_' || referrer_key.kind
            || ' LIKE ' || quote_literal(referrer_key.key_string)
```
Chapter 38. PL/pgSQL - SQL Procedural Language

|| ' THEN RETURN ' || quote_literal(referrer_key.referrer_type) || '; END IF;' ;
END LOOP;

func_body := func_body || ' RETURN NULL; END;' ;

func_cmd :=
'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
 v_domain varchar,
 v_url varchar)
 RETURNS varchar AS ' || quote_literal(func_body) || ' LANGUAGE plpgsql;' ;

EXECUTE func_cmd;

END;

$func$ LANGUAGE plpgsql;

Notice how the body of the function is built separately and passed through quote_literal to double any quote marks in it. This technique is needed because we cannot safely use dollar quoting for defining the new function: we do not know for sure what strings will be interpolated from the referrer_key.key_string field. (We are assuming here that referrer_key.kind can be trusted to always be host, domain, or url, but referrer_key.key_string might be anything, in particular it might contain dollar signs.) This function is actually an improvement on the Oracle original, because it will not generate broken code when referrer_key.key_string or referrer_key.referrer_type contain quote marks.

Example 38-7 shows how to port a function with OUT parameters and string manipulation. PostgreSQL does not have a built-in instr function, but you can create one using a combination of other functions. In Section 38.12.3 there is a PL/pgSQL implementation of instr that you can use to make your porting easier.

Example 38-7. Porting a Procedure With String Manipulation and OUT Parameters from PL/SQL to PL/pgSQL

The following Oracle PL/SQL procedure is used to parse a URL and return several elements (host, path, and query).

This is the Oracle version:

CREATE OR REPLACE PROCEDURE cs_parse_url(
 v_url IN VARCHAR,
 v_host OUT VARCHAR, -- This will be passed back
 v_path OUT VARCHAR, -- This one too
 v_query OUT VARCHAR) -- And this one
IS
 a_pos1 INTEGER;
 a_pos2 INTEGER;
BEGIN
 v_host := NULL;
 v_path := NULL;
 v_query := NULL;
 a_pos1 := instr(v_url, '//');

 IF a_pos1 = 0 THEN
 RETURN;

 Example 38-7 shows how to port a function with OUT parameters and string manipulation. PostgreSQL does not have a built-in instr function, but you can create one using a combination of other functions. In Section 38.12.3 there is a PL/pgSQL implementation of instr that you can use to make your porting easier.

 Example 38-7. Porting a Procedure With String Manipulation and OUT Parameters from PL/SQL to PL/pgSQL

 The following Oracle PL/SQL procedure is used to parse a URL and return several elements (host, path, and query).

 This is the Oracle version:

 CREATE OR REPLACE PROCEDURE cs_parse_url(
   v_url IN VARCHAR,
   v_host OUT VARCHAR, -- This will be passed back
   v_path OUT VARCHAR, -- This one too
   v_query OUT VARCHAR) -- And this one
   IS
     a_pos1 INTEGER;
     a_pos2 INTEGER;
   BEGIN
     v_host := NULL;
     v_path := NULL;
     v_query := NULL;
     a_pos1 := instr(v_url, '//');

     IF a_pos1 = 0 THEN
       RETURN;

     func_body := func_body || ' RETURN NULL; END;' ;

     func_cmd :=
       'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
         v_domain varchar,
         v_url varchar)
         RETURNS varchar AS ' || quote_literal(func_body) || ' LANGUAGE plpgsql;' ;

     EXECUTE func_cmd;

     END;

     END;

     $func$ LANGUAGE plpgsql;

 Notice how the body of the function is built separately and passed through quote_literal to double any quote marks in it. This technique is needed because we cannot safely use dollar quoting for defining the new function: we do not know for sure what strings will be interpolated from the referrer_key.key_string field. (We are assuming here that referrer_key.kind can be trusted to always be host, domain, or url, but referrer_key.key_string might be anything, in particular it might contain dollar signs.) This function is actually an improvement on the Oracle original, because it will not generate broken code when referrer_key.key_string or referrer_key.referrer_type contain quote marks.

 Example 38-7 shows how to port a function with OUT parameters and string manipulation. PostgreSQL does not have a built-in instr function, but you can create one using a combination of other functions. In Section 38.12.3 there is a PL/pgSQL implementation of instr that you can use to make your porting easier.
Chapter 38. PL/pgSQL - SQL Procedural Language

END IF;

END;

Here is a possible translation into PL/pgSQL:

CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR, -- This will be passed back
    v_path OUT VARCHAR, -- This one too
    v_query OUT VARCHAR) -- And this one
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;

    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/

show errors;
Chapter 38. PL/pgSQL - SQL Procedural Language

This function could be used like this:

```sql
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Example 38-8 shows how to port a procedure that uses numerous features that are specific to Oracle.

**Example 38-8. Porting a Procedure from PL/SQL to PL/pgSQL**

The Oracle version:

```sql
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
  a_running_job_count INTEGER;
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

  SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

  IF a_running_job_count > 0 THEN
    COMMIT; -- free lock
    RAISE_APPLICATION_ERROR(-20000, 'Unable to create a new job: a job is currently running.');</n
  END IF;

  DELETE FROM cs_active_job;
  INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

  BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
  EXCEPTION
    WHEN dup_val_on_index THEN NULL; -- don’t worry if it already exists
  END;
  COMMIT;
END;
/
```

Procedures like this can easily be converted into PostgreSQL functions returning `void`. This procedure in particular is interesting because it can teach us some things:

1. There is no `PRAGMA` statement in PostgreSQL.
2. If you do a `LOCK TABLE` in PL/pgSQL, the lock will not be released until the calling transaction is finished.
3. You cannot issue `COMMIT` in a PL/pgSQL function. The function is running within some outer transaction and so `COMMIT` would imply terminating the function’s execution. However, in this particular case it is not necessary anyway, because the lock obtained by the `LOCK TABLE` will be released when we raise an error.

This is how we could port this procedure to PL/pgSQL:

```sql
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
  a_running_job_count integer;
BEGIN
  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

  SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

  IF a_running_job_count > 0 THEN
    COMMIT; -- free lock
    RAISE_APPLICATION_ERROR(-20000, 'Unable to create a new job: a job is currently running.');</n
  END IF;

  DELETE FROM cs_active_job;
  INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

  BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
  EXCEPTION
    WHEN dup_val_on_index THEN NULL; -- don’t worry if it already exists
  END;
  COMMIT;
END;
$$;
```

767
BEGIN
  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

  SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

  IF a_running_job_count > 0 THEN
    RAISE EXCEPTION 'Unable to create a new job: a job is currently running';  
  END IF;

  DELETE FROM cs_active_job;
  INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

  BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
  EXCEPTION
    WHEN unique_violation THEN
      -- don't worry if it already exists
  END;
END;
$$ LANGUAGE plpgsql;

1. The syntax of RAISE is considerably different from Oracle’s similar statement.
2. The exception names supported by PL/pgSQL are different from Oracle’s. The set of built-in exception names is much larger (see Appendix A). There is not currently a way to declare user-defined exception names.

The main functional difference between this procedure and the Oracle equivalent is that the exclusive lock on the cs_jobs table will be held until the calling transaction completes. Also, if the caller later aborts (for example due to an error), the effects of this procedure will be rolled back.

38.12.2. Other Things to Watch For

This section explains a few other things to watch for when porting Oracle PL/SQL functions to PostgreSQL.

38.12.2.1. Implicit Rollback after Exceptions

In PL/pgSQL, when an exception is caught by an EXCEPTION clause, all database changes since the block’s BEGIN are automatically rolled back. That is, the behavior is equivalent to what you’d get in Oracle with:

BEGIN
  SAVEPOINT s1;
  ... code here ...
EXCEPTION
  WHEN ... THEN
    ROLLBACK TO s1;
    ... code here ...
  WHEN ... THEN
    ROLLBACK TO s1;
    ... code here ...
END;
If you are translating an Oracle procedure that uses `SAVEPOINT` and `ROLLBACK TO` in this style, your task is easy: just omit the `SAVEPOINT` and `ROLLBACK TO`. If you have a procedure that uses `SAVEPOINT` and `ROLLBACK TO` in a different way then some actual thought will be required.

### 38.12.2.2. EXECUTE

The PL/pgSQL version of `EXECUTE` works similarly to the PL/SQL version, but you have to remember to use `quote_literal` and `quote_ident` as described in Section 38.5.4. Constructs of the type `EXECUTE ‘SELECT * FROM $1’;` will not work reliably unless you use these functions.

### 38.12.2.3. Optimizing PL/pgSQL Functions

PostgreSQL gives you two function creation modifiers to optimize execution: “volatility” (whether the function always returns the same result when given the same arguments) and “strictness” (whether the function returns null if any argument is null). Consult the `CREATE FUNCTION` reference page for details.

When making use of these optimization attributes, your `CREATE FUNCTION` statement might look something like this:

```sql
CREATE FUNCTION foo(...) RETURNS integer AS $$
  ...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

### 38.12.3. Appendix

This section contains the code for a set of Oracle-compatible `instr` functions that you can use to simplify your porting efforts.

```sql
-- instr functions that mimic Oracle’s counterpart
-- Syntax: instr(string1, string2, [n], [m]) where [] denotes optional parameters.
-- Searches string1 beginning at the nth character for the mth occurrence
-- of string2. If n is negative, search backwards. If m is not passed,
-- assume 1 (search starts at first character).

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
  DECLARE
    pos integer;
  BEGIN
    pos := instr($1, $2, 1);
    RETURN pos;
  END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_index integer)
  RETURNS integer AS $$
```

769
DECLARE
pos integer NOT NULL DEFAULT 0;
temp_str varchar;
beg integer;
length integer;
ss_length integer;
BEGIN
IF beg_index > 0 THEN
    temp_str := substring(string FROM beg_index);
    pos := position(string_to_search IN temp_str);
    IF pos = 0 THEN
        RETURN 0;
    ELSE
        RETURN pos + beg_index - 1;
    END IF;
ELSE
    ss_length := char_length(string_to_search);
    length := char_length(string);
    beg := length + beg_index - ss_length + 2;
    WHILE beg > 0 LOOP
        temp_str := substring(string FROM beg FOR ss_length);
        pos := position(string_to_search IN temp_str);
        IF pos > 0 THEN
            RETURN beg;
        END IF;
        beg := beg - 1;
    END LOOP;
    RETURN 0;
END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string_to_search varchar,
beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
pos integer NOT NULL DEFAULT 0;
occur_number integer NOT NULL DEFAULT 0;
temp_str varchar;
beg integer;
i integer;
length integer;
ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);
        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);
        END LOOP;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
Chapter 38. PL/pgSQL - SQL Procedural Language

```
IF i = 1 THEN
    beg := beg + pos - 1;
ELSE
    beg := beg + pos;
END IF;

temp_str := substring(string FROM beg + 1);
END LOOP;

IF pos = 0 THEN
    RETURN 0;
ELSE
    RETURN beg;
END IF;
ELSE
    ss_length := char_length(string_to_search);
    length := char_length(string);
    beg := length + beg_index - ss_length + 2;

    WHILE beg > 0 LOOP
        temp_str := substring(string FROM beg FOR ss_length);
        pos := position(string_to_search IN temp_str);

        IF pos > 0 THEN
            occur_number := occur_number + 1;

            IF occur_number = occur_index THEN
                RETURN beg;
            END IF;
        END IF;
        beg := beg - 1;
    END LOOP;

    RETURN 0;
END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```
Chapter 39. PL/Tcl - Tcl Procedural Language

PL/Tcl is a loadable procedural language for the PostgreSQL database system that enables the Tcl language\(^1\) to be used to write functions and trigger procedures.

39.1. Overview

PL/Tcl offers most of the capabilities a function writer has in the C language, with a few restrictions, and with the addition of the powerful string processing libraries that are available for Tcl.

One compelling good restriction is that everything is executed from within the safety of the context of a Tcl interpreter. In addition to the limited command set of safe Tcl, only a few commands are available to access the database via SPI and to raise messages via `elog()`. PL/Tcl provides no way to access internals of the database server or to gain OS-level access under the permissions of the PostgreSQL server process, as a C function can do. Thus, unprivileged database users can be trusted to use this language; it does not give them unlimited authority.

The other notable implementation restriction is that Tcl functions cannot be used to create input/output functions for new data types.

Sometimes it is desirable to write Tcl functions that are not restricted to safe Tcl. For example, one might want a Tcl function that sends email. To handle these cases, there is a variant of PL/Tcl called PL/TclU (for untrusted Tcl). This is the exact same language except that a full Tcl interpreter is used. If PL/TclU is used, it must be installed as an untrusted procedural language so that only database superusers can create functions in it. The writer of a PL/TclU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator.

The shared object code for the PL/Tcl and PL/TclU call handlers is automatically built and installed in the PostgreSQL library directory if Tcl support is specified in the configuration step of the installation procedure. To install PL/Tcl and/or PL/TclU in a particular database, use the `createlang` program, for example `createlang pltcl dbname` or `createlang pltclu dbname`.

39.2. PL/Tcl Functions and Arguments

To create a function in the PL/Tcl language, use the standard `CREATE FUNCTION` syntax:

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
# PL/Tcl function body
$$ LANGUAGE pltcl;
```

PL/TclU is the same, except that the language has to be specified as `pltclu`.

The body of the function is simply a piece of Tcl script. When the function is called, the argument values are passed as variables `$1` ... `$n` to the Tcl script. The result is returned from the Tcl code in the usual way, with a `return` statement.

For example, a function returning the greater of two integer values could be defined as:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
if ($1 > $2) {return $1}
return $2
$$
```

---

1. http://www.tcl.tk/
Note the clause **STRICT**, which saves us from having to think about null input values: if a null value is passed, the function will not be called at all, but will just return a null result automatically.

In a nonstrict function, if the actual value of an argument is null, the corresponding $n variable will be set to an empty string. To detect whether a particular argument is null, use the function `argisnull`. For example, suppose that we wanted `tcl_max` with one null and one nonnull argument to return the nonnull argument, rather than null:

```tcl
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

As shown above, to return a null value from a PL/Tcl function, execute `return_null`. This can be done whether the function is strict or not.

Composite-type arguments are passed to the function as Tcl arrays. The element names of the array are the attribute names of the composite type. If an attribute in the passed row has the null value, it will not appear in the array. Here is an example:

```tcl
CREATE TABLE employee {
    name text,
    salary integer,
    age integer
};

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

There is currently no support for returning a composite-type result value, nor for returning sets.

PL/Tcl does not currently have full support for domain types: it treats a domain the same as the underlying scalar type. This means that constraints associated with the domain will not be enforced. This is not an issue for function arguments, but it is a hazard if you declare a PL/Tcl function as returning a domain type.
39.3. Data Values in PL/Tcl

The argument values supplied to a PL/Tcl function’s code are simply the input arguments converted to text form (just as if they had been displayed by a SELECT statement). Conversely, the return command will accept any string that is acceptable input format for the function’s declared return type. So, within the PL/Tcl function, all values are just text strings.

39.4. Global Data in PL/Tcl

Sometimes it is useful to have some global data that is held between two calls to a function or is shared between different functions. This is easily done in PL/Tcl, but there are some restrictions that must be understood.

For security reasons, PL/Tcl executes functions called by any one SQL role in a separate Tcl interpreter for that role. This prevents accidental or malicious interference by one user with the behavior of another user’s PL/Tcl functions. Each such interpreter will have its own values for any “global” Tcl variables. Thus, two PL/Tcl functions will share the same global variables if and only if they are executed by the same SQL role. In an application wherein a single session executes code under multiple SQL roles (via SECURITY DEFINER functions, use of SET ROLE, etc) you may need to take explicit steps to ensure that PL/Tcl functions can share data. To do that, make sure that functions that should communicate are owned by the same user, and mark them SECURITY DEFINER. You must of course take care that such functions can’t be used to do anything unintended.

All PL/Tcl functions used in a session execute in the same Tcl interpreter, which of course is distinct from the interpreter(s) used for PL/Tcl functions. So global data is automatically shared between PL/Tcl functions. This is not considered a security risk because all PL/Tcl functions execute at the same trust level, namely that of a database superuser.

To help protect PL/Tcl functions from unintentionally interfering with each other, a global array is made available to each function via the upvar command. The global name of this variable is the function’s internal name, and the local name is GD. It is recommended that GD be used for persistent private data of a function. Use regular Tcl global variables only for values that you specifically intend to be shared among multiple functions. (Note that the GD arrays are only global within a particular interpreter, so they do not bypass the security restrictions mentioned above.)

An example of using GD appears in the spi_execp example below.

39.5. Database Access from PL/Tcl

The following commands are available to access the database from the body of a PL/Tcl function:

```
spi_exec ?-count n? ?-array name? command ?-loop-body?
```

Executes an SQL command given as a string. An error in the command causes an error to be raised. Otherwise, the return value of spi_exec is the number of rows processed (selected, inserted, updated, or deleted) by the command, or zero if the command is a utility statement. In addition, if the command is a SELECT statement, the values of the selected columns are placed in Tcl variables as described below.

The optional -count value tells spi_exec the maximum number of rows to process in the command. The effect of this is comparable to setting up a query as a cursor and then saying FETCH n.
If the command is a `SELECT` statement, the values of the result columns are placed into Tcl variables named after the columns. If the `-array` option is given, the column values are instead stored into the named associative array, with the column names used as array indexes.

If the command is a `SELECT` statement and no `loop-body` script is given, then only the first row of results are stored into Tcl variables; remaining rows, if any, are ignored. No storing occurs if the query returns no rows. (This case can be detected by checking the result of `spi_exec`.) For example:

```tcl
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

will set the Tcl variable `$cnt` to the number of rows in the `pgProc` system catalog.

If the optional `loop-body` argument is given, it is a piece of Tcl script that is executed once for each row in the query result. (`loop-body` is ignored if the given command is not a `SELECT`.) The values of the current row’s columns are stored into Tcl variables before each iteration. For example:

```tcl
spi_exec -array C "SELECT * FROM pg_class" {
  elog DEBUG "have table $C(relname)"
}
```

will print a log message for every row of `pg_class`. This feature works similarly to other Tcl looping constructs; in particular `continue` and `break` work in the usual way inside the loop body.

If a column of a query result is null, the target variable for it is “unset” rather than being set.

```tcl
spi_prepare query typelist
```

Prepares and saves a query plan for later execution. The saved plan will be retained for the life of the current session.

The query can use parameters, that is, placeholders for values to be supplied whenever the plan is actually executed. In the query string, refer to parameters by the symbols `$1` ... `$n`. If the query uses parameters, the names of the parameter types must be given as a Tcl list. (Write an empty list for `typelist` if no parameters are used.)

The return value from `spi_prepare` is a query ID to be used in subsequent calls to `spi_execp`. See `spi_execp` for an example.

```tcl
```

Executes a query previously prepared with `spi_prepare`. `queryid` is the ID returned by `spi_prepare`. If the query references parameters, a `value-list` must be supplied. This is a Tcl list of actual values for the parameters. The list must be the same length as the parameter type list previously given to `spi_prepare`. `Omit value-list` if the query has no parameters.

The optional value for `-nulls` is a string of spaces and `'n'` characters telling `spi_execp` which of the parameters are null values. If given, it must have exactly the same length as the `value-list`. If it is not given, all the parameter values are nonnull.

Except for the way in which the query and its parameters are specified, `spi_execp` works just like `spi_exec`. The `-count`, `-array`, and `loop-body` options are the same, and so is the result value.

Here’s an example of a PL/Tcl function using a prepared plan:

```tcl
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
  if {! [info exists GD(plan) ]} {
    # prepare the saved plan on the first call
    set GD(plan) [ spi_prepare "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
```
spi_execp

```
spi_execp -count 1 $GD(plan) [ list $1 $2 ]
return $cnt
```

```$ LANGUAGE pltcl;
We need backslashes inside the query string given to spi_prepare to ensure that the $n markers will be passed through to spi_prepare as-is, and not replaced by Tcl variable substitution.
```

spi_lastoid

Returns the OID of the row inserted by the last spi_exec or spi_execp, if the command was a single-row INSERT and the modified table contained OIDs. (If not, you get zero.)

```
quote string

Doubles all occurrences of single quote and backslash characters in the given string. This can be used to safely quote strings that are to be inserted into SQL commands given to spi_exec or spi_prepare. For example, think about an SQL command string like:

"SELECT '$val' AS ret"

where the Tcl variable val actually contains doesn’t. This would result in the final command string:

SELECT 'doesn’t' AS ret

which would cause a parse error during spi_exec or spi_prepare. To work properly, the submitted command should contain:

SELECT 'doesn”t' AS ret

which can be formed in PL/Tcl using:

"SELECT [ quote $val ]' AS ret"

One advantage of spi_execp is that you don’t have to quote parameter values like this, since the parameters are never parsed as part of an SQL command string.

elog level msg

Emits a log or error message. Possible levels are DEBUG, LOG, INFO, NOTICE, WARNING, ERROR, and FATAL. ERROR raises an error condition; if this is not trapped by the surrounding Tcl code, the error propagates out to the calling query, causing the current transaction or subtransaction to be aborted. This is effectively the same as the Tcl error command. FATAL aborts the transaction and causes the current session to shut down. (There is probably no good reason to use this error level in PL/Tcl functions, but it’s provided for completeness.) The other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

39.6. Trigger Procedures in PL/Tcl

Trigger procedures can be written in PL/Tcl. PostgreSQL requires that a procedure that is to be called as a trigger must be declared as a function with no arguments and a return type of trigger.

The information from the trigger manager is passed to the procedure body in the following variables:

```
$TG_name
```

The name of the trigger from the CREATE TRIGGER statement.
$TG_relid
The object ID of the table that caused the trigger procedure to be invoked.

$TG_table_name
The name of the table that caused the trigger procedure to be invoked.

$TG_table_schema
The schema of the table that caused the trigger procedure to be invoked.

$TG_relatts
A Tcl list of the table column names, prefixed with an empty list element. So looking up a column name in the list with Tcl’s lsearch command returns the element’s number starting with 1 for the first column, the same way the columns are customarily numbered in PostgreSQL. (Empty list elements also appear in the positions of columns that have been dropped, so that the attribute numbering is correct for columns to their right.)

$TG_when
The string \texttt{BEFORE} or \texttt{AFTER} depending on the type of trigger call.

$TG_level
The string \texttt{ROW} or \texttt{STATEMENT} depending on the type of trigger call.

$TG_op
The string \texttt{INSERT}, \texttt{UPDATE}, or \texttt{DELETE} depending on the type of trigger call.

$NEW
An associative array containing the values of the new table row for \texttt{INSERT} or \texttt{UPDATE} actions, or empty for \texttt{DELETE}. The array is indexed by column name. Columns that are null will not appear in the array.

$OLD
An associative array containing the values of the old table row for \texttt{UPDATE} or \texttt{DELETE} actions, or empty for \texttt{INSERT}. The array is indexed by column name. Columns that are null will not appear in the array.

$args
A Tcl list of the arguments to the procedure as given in the \texttt{CREATE TRIGGER} statement. These arguments are also accessible as \$1 ... \$n in the procedure body.

The return value from a trigger procedure can be one of the strings \texttt{OK} or \texttt{SKIP}, or a list as returned by the array get Tcl command. If the return value is \texttt{OK}, the operation (\texttt{INSERT/UPDATE/DELETE}) that fired the trigger will proceed normally. \texttt{SKIP} tells the trigger manager to silently suppress the operation for this row. If a list is returned, it tells PL/Tcl to return a modified row to the trigger manager that will be inserted instead of the one given in $NEW. (This works for \texttt{INSERT} and \texttt{UPDATE} only.) Needless to say that all this is only meaningful when the trigger is \texttt{BEFORE} and \texttt{FOR EACH ROW}; otherwise the return value is ignored.

Here’s a little example trigger procedure that forces an integer value in a table to keep track of the number of updates that are performed on the row. For new rows inserted, the value is initialized to 0 and then incremented on every update operation.

\begin{verbatim}
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
switch $TG_op {
   INSERT {
      $NEW[modcount] = 0;
      $NEW[modcount] = $NEW[modcount] + 1;
   };
   UPDATE {
      $NEW[modcount] = 0;
      $NEW[modcount] = $NEW[modcount] + 1;
   };
   DELETE {
      $OLD[modcount] = 0;
      $OLD[modcount] = $OLD[modcount] - 1;
   };
}$$;
\end{verbatim}
Chapter 39. PL/Tcl - Tcl Procedural Language

```
set NEW($1) 0
}
UPDATE {
    set NEW($1) $OLD($1)
    incr NEW($1)
}
default {
    return OK
}
return [array get NEW]
```

```bash
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);
CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Notice that the trigger procedure itself does not know the column name; that’s supplied from the trigger arguments. This lets the trigger procedure be reused with different tables.

### 39.7. Modules and the `unknown` command

PL/Tcl has support for autoloading Tcl code when used. It recognizes a special table, `pltcl_modules`, which is presumed to contain modules of Tcl code. If this table exists, the module `unknown` is fetched from the table and loaded into the Tcl interpreter immediately before the first execution of a PL/Tcl function in a database session. (This happens separately for each Tcl interpreter, if more than one is used in a session; see Section 39.4.)

While the `unknown` module could actually contain any initialization script you need, it normally defines a Tcl `unknown` procedure that is invoked whenever Tcl does not recognize an invoked procedure name. PL/Tcl’s standard version of this procedure tries to find a module in `pltcl_modules` that will define the required procedure. If one is found, it is loaded into the interpreter, and then execution is allowed to proceed with the originally attempted procedure call. A secondary table `pltcl_modfuncs` provides an index of which functions are defined by which modules, so that the lookup is reasonably quick.

The PostgreSQL distribution includes support scripts to maintain these tables: `pltcl_loadmod`, `pltcl_listmod`, `pltcl_delmod`, as well as source for the standard `unknown` module in `share/unknown.pltcl`. This module must be loaded into each database initially to support the autoloading mechanism.

The tables `pltcl_modules` and `pltcl_modfuncs` must be readable by all, but it is wise to make them owned and writable only by the database administrator. As a security precaution, PL/Tcl will ignore `pltcl_modules` (and thus, not attempt to load the `unknown` module) unless it is owned by a superuser. But update privileges on this table can be granted to other users, if you trust them sufficiently.

### 39.8. Tcl Procedure Names

In PostgreSQL, the same function name can be used for different function definitions as long as the number of arguments or their types differ. Tcl, however, requires all procedure names to be distinct.
PL/Tcl deals with this by making the internal Tcl procedure names contain the object ID of the function from the system table `pg_proc` as part of their name. Thus, PostgreSQL functions with the same name and different argument types will be different Tcl procedures, too. This is not normally a concern for a PL/Tcl programmer, but it might be visible when debugging.
Chapter 40. PL/Perl - Perl Procedural Language

PL/Perl is a loadable procedural language that enables you to write PostgreSQL functions in the Perl programming language\(^1\).

The usual advantage to using PL/Perl is that this allows use, within stored functions, of the manyfold “string munging” operators and functions available for Perl. Parsing complex strings might be easier using Perl than it is with the string functions and control structures provided in PL/pgSQL.

To install PL/Perl in a particular database, use `createlang plperl dbname`.

**Tip:** If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

**Note:** Users of source packages must specially enable the build of PL/Perl during the installation process. (Refer to Chapter 15 for more information.) Users of binary packages might find PL/Perl in a separate subpackage.

### 40.1. PL/Perl Functions and Arguments

To create a function in the PL/Perl language, use the standard `CREATE FUNCTION` syntax:

```sql
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Perl function body
$$ LANGUAGE plperl;
```

The body of the function is ordinary Perl code. In fact, the PL/Perl glue code wraps it inside a Perl subroutine. A PL/Perl function must always return a scalar value. You can return more complex structures (arrays, records, and sets) by returning a reference, as discussed below. Never return a list.

**Note:** The use of named nested subroutines is dangerous in Perl, especially if they refer to lexical variables in the enclosing scope. Because a PL/Perl function is wrapped in a subroutine, any named subroutine you create will be nested. In general, it is far safer to create anonymous subroutines which you call via a coderef. See the `perldiag` man page for more details.

The syntax of the `CREATE FUNCTION` command requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see Section 4.1.2.2) for the string constant. If you choose to use escape string syntax `E"`, you must double any single quote marks (`'`) and backslashes (`\`) used in the body of the function (see Section 4.1.2.1).

Arguments and results are handled as in any other Perl subroutine: arguments are passed in `@_`, and a result value is returned with `return` or as the last expression evaluated in the function.

For example, a function returning the greater of two integer values could be defined as:

```sql
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    # PL/Perl function body
$$ LANGUAGE plperl;
```

---

1. [http://www.perl.com](http://www.perl.com)

780
Chapter 40. PL/Perl - Perl Procedural Language

if ($_[0] > $_[1]) { return $_[0]; }
return $_[1];

```plperl
my ($x,$y) = @_; if (! defined $x) { if (! defined $y) { return undef; } return $y; }
if (! defined $y) { return $x; }
if ($x > $y) { return $x; }
return $y;
```

As shown above, to return an SQL null value from a PL/Perl function, return an undefined value. This can be done whether the function is strict or not.

Anything in a function argument that is not a reference is a string, which is in the standard PostgreSQL external text representation for the relevant data type. In the case of ordinary numeric or text types, Perl will just do the right thing and the programmer will normally not have to worry about it. However, in other cases the argument will need to be converted into a form that is more usable in Perl. For example, here is how to convert an argument of type `bytea` into unescaped binary data:

```perl
my $arg = shift;
$arg =~ s!(\|[^ -~])!chr(oct($1))!ge;
```

Similarly, values passed back to PostgreSQL must be in the external text representation format. For example, here is how to escape binary data for a return value of type `bytea`:

```perl
$retval =~ s!(\|[^ -~])!sprintf("\%03o",ord($1))!ge;
return $retval;
```

Perl can return PostgreSQL arrays as references to Perl arrays. Here is an example:

```plperl
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
  return [ ['a''b','c,d'], ['e\f','g'] ];
$$ LANGUAGE plperl;

select returns_array();
```
Composite-type arguments are passed to the function as references to hashes. The keys of the hash are the attribute names of the composite type. Here is an example:

```sql
CREATE TABLE employee {
    name text,
    basesalary integer,
    bonus integer
};

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
my ($emp) = @_; 
return $emp->{basesalary} + $emp->{bonus};$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

A PL/Perl function can return a composite-type result using the same approach: return a reference to a hash that has the required attributes. For example:

```sql
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
return {f2 = 'hello', f1 = 1, f3 = 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

Any columns in the declared result data type that are not present in the hash will be returned as null values.

PL/Perl functions can also return sets of either scalar or composite types. Usually you’ll want to return rows one at a time, both to speed up startup time and to keep from queueing up the entire result set in memory. You can do this with `return_next` as illustrated below. Note that after the last `return_next`, you must put either `return` or (better) `return undef`.

```sql
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
foreach (0..$_[0]) {
    return_next($_);
} 
return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);
```

For small result sets, you can return a reference to an array that contains either scalars, references to arrays, or references to hashes for simple types, array types, and composite types, respectively. Here are some simple examples of returning the entire result set as an array reference:

```sql
CREATE OR REPLACE FUNCTION perl_set() 
RETURNS SETOF testrowperl AS $$
return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
return [0..$_[0]]; $$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
return [
    { f1 => 1, f2 => 'Hello', f3 => 'World' },
    { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
    { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
]; $$ LANGUAGE plperl;

SELECT * FROM perl_set();

If you wish to use the strict pragma with your code, the easiest way to do so is to SET plperl.use_strict to true. This parameter affects subsequent compilations of PL/Perl functions, but not functions already compiled in the current session. To set the parameter before PL/Perl has been loaded, it is necessary to have added “plperl” to the custom_variable_classes list in postgresql.conf.

Another way to use the strict pragma is to put:

use strict;

in the function body.

### 40.2. Database Access from PL/Perl

Access to the database itself from your Perl function can be done via the function `spi_exec_query` described below, or via an experimental module `DBD::PgSPI` (also available at CPAN mirror sites). This module makes available a DBI-compliant database-handle named `$pg_dbh` that can be used to perform queries with normal DBI syntax.

PL/Perl provides additional Perl commands:

```
spi_exec_query(query [, max-rows])
spi_query(command)
spi_fetchrow(cursor)
spi_prepare(command, argument types)
spi_exec_prepared(plan)
spi_query_prepared(plan [, attributes], arguments)
spi_cursor_close(cursor)
spi_freeplan(plan)
```

`spi_exec_query` executes an SQL command and returns the entire row set as a reference to an array of hash references. You should only use this command when you know that the result set will be relatively small. Here is an example of a query (SELECT command) with the optional maximum number of rows:

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

---

This returns up to 5 rows from the table `my_table`. If `my_table` has a column `my_column`, you can get that value from row `$i` of the result like this:

```
$foo = $rv->{rows}[$i]->{my_column};
```

The total number of rows returned from a `SELECT` query can be accessed like this:

```
$nrows = $rv->{processed}
```

Here is an example using a different command type:

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
You can then access the command status (e.g., SPI_OK_INSERT) like this:

```
$res = $rv->{status};
```

To get the number of rows affected, do:

```
$nrows = $rv->{processed};
```

Here is a complete example:

```perl
CREATE TABLE test {
  i int,
  v varchar
};

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
my $rv = spi_exec_query('select i, v from test;');
my $status = $rv->{status};
my $nrows = $rv->{processed};
foreach my $rn (0 .. $nrows - 1) {
  my $row = $rv->{rows}[$rn];
  $row->{i} += 200 if defined($row->{i});
  $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
  return_next($row);
}
return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

`spi_query` and `spi_fetchrow` work together as a pair for row sets which might be large, or for cases where you wish to return rows as they arrive. `spi_fetchrow` works only with `spi_query`. The following example illustrates how you use them together:

```sql
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);
CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
use Digest::MD5 qw(md5_hex);
my $file = '/usr/share/dict/words';
my $t = localtime;
elog(NOTICE, "opening file $file at $t" );
open my $fh, '<', $file # ooh, it's a file access!
  or elog(ERROR, "cannot open $file for reading: $!");
my @words = <$fh>;
close $fh;
$t = localtime;
```
elog(NOTICE, "closed file $file at $t");
chomp(@words);
my $row;
my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)"),
while (defined ($row = spi_fetchrow($sth))) {
  return_next({
    the_num => $row->{a},
    the_text => md5_hex($words[rand @words])
  });
}
return;
$$ LANGUAGE plperlu;
SELECT * from lotsa_md5(500);

spi_prepare, spi_query_prepared, spi_exec_prepared, and spi_freeplan implement the same functionality but for prepared queries. Once a query plan is prepared by a call to spi_prepare, the plan can be used instead of the string query, either in spi_exec_prepared, where the result is the same as returned by spi_exec_query, or in spi_query_prepared which returns a cursor exactly as spi_query does, which can be later passed to spi_fetchrow.

The advantage of prepared queries is that it is possible to use one prepared plan for more than one query execution. After the plan is not needed anymore, it can be freed with spi_freeplan:

CREATE OR REPLACE FUNCTION init() RETURNS INTEGER AS $$
  $_SHARED{my_plan} = spi_prepare( 'SELECT (now() + $1)::date AS now', 'INTERVAL');
$$ LANGUAGE plperl;
CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
  return spi_exec_prepared(
    $_SHARED{my_plan},
    $_[0],
  )->{rows}->[0]->{now};
$$ LANGUAGE plperl;
CREATE OR REPLACE FUNCTION done() RETURNS INTEGER AS $$
  spi_freeplan( $_SHARED{my_plan});
  undef $_SHARED{my_plan};
$$ LANGUAGE plperl;
SELECT init();
SELECT add_time(‘1 day’), add_time(‘2 days’), add_time(‘3 days’);
SELECT done();

<table>
<thead>
<tr>
<th>add_time</th>
<th>add_time</th>
<th>add_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005-12-10</td>
<td>2005-12-11</td>
<td>2005-12-12</td>
</tr>
</tbody>
</table>

Note that the parameter subscript in spi_prepare is defined via $1, $2, $3, etc, so avoid declaring query strings in double quotes that might easily lead to hard-to-catch bugs.

Normally, spi_fetchrow should be repeated until it returns undef, indicating that there are no more rows to read. The cursor is automatically freed when spi_fetchrow returns undef. If you do not wish to read all the rows, instead call spi_cursor_close to free the cursor. Failure to do so will result in memory leaks.
elog(level, msg)

Emit a log or error message. Possible levels are DEBUG, LOG, INFO, NOTICE, WARNING, and ERROR. ERROR raises an error condition; if this is not trapped by the surrounding Perl code, the error propagates out to the calling query, causing the current transaction or subtransaction to be aborted. This is effectively the same as the Perl die command. The other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

40.3. Data Values in PL/Perl

The argument values supplied to a PL/Perl function’s code are simply the input arguments converted to text form (just as if they had been displayed by a SELECT statement). Conversely, the return command will accept any string that is acceptable input format for the function’s declared return type. So, within the PL/Perl function, all values are just text strings.

40.4. Global Values in PL/Perl

You can use the global hash %_SHARED to store data, including code references, between function calls for the lifetime of the current session.

Here is a simple example for shared data:

```perl
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
  if ($_SHARED{$_[0]} = $_[1]) {
      return 'ok';
  } else {
      return "cannot set shared variable $_[0] to $_[1]";
  }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
  return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl! How’s tricks?');
SELECT get_var('sample');
```

Here is a slightly more complicated example using a code reference:

```perl
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
  $_SHARED{myquote} = sub {
    my $arg = shift;
    $arg =~ s/(['\])/\$1/g;
    return "'$arg'";
  };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* initializes the function */
```
CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
my $text_to_quote = shift;
my $qfunc = $_SHARED{myquote};
return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;

(Return could have replaced the above with the one-liner return $_SHARED{myquote}->($_[0]); at the expense of readability.)

For security reasons, PL/Perl executes functions called by any one SQL role in a separate Perl interpreter for that role. This prevents accidental or malicious interference by one user with the behavior of another user’s PL/Perl functions. Each such interpreter has its own value of the %_SHARED variable and other global state. Thus, two PL/Perl functions will share the same value of %_SHARED if and only if they are executed by the same SQL role. In an application wherein a single session executes code under multiple SQL roles (via SECURITY DEFINER functions, use of SET ROLE, etc) you may need to take explicit steps to ensure that PL/Perl functions can share data via %_SHARED. To do that, make sure that functions that should communicate are owned by the same user, and mark them SECURITY DEFINER. You must of course take care that such functions can’t be used to do anything unintended.

### 40.5. Trusted and Untrusted PL/Perl

Normally, PL/Perl is installed as a “trusted” programming language named plperl. In this setup, certain Perl operations are disabled to preserve security. In general, the operations that are restricted are those that interact with the environment. This includes file handle operations, require, and use (for external modules). There is no way to access internals of the database server process or to gain OS-level access with the permissions of the server process, as a C function can do. Thus, any unprivileged database user can be permitted to use this language.

Here is an example of a function that will not work because file system operations are not allowed for security reasons:

CREATE FUNCTION badfunc() RETURNS integer AS $$
my $tmpfile = "~/tmp/badfile";
open my $fh, ‘>', $tmpfile
or elog(ERROR, qq{could not open the file "$tmpfile": $!});
print $fh "Testing writing to a file
";
close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
return 1;
$$ LANGUAGE plperl;

The creation of this function will fail as its use of a forbidden operation will be caught by the validator.

Sometimes it is desirable to write Perl functions that are not restricted. For example, one might want a Perl function that sends mail. To handle these cases, PL/Perl can also be installed as an “untrusted” language (usually called PL/PerlU). In this case the full Perl language is available. If the createlang program is used to install the language, the language name plperlu will select the untrusted PL/Perl variant.

The writer of a PL/PerlU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator. Note that the database system allows only database superusers to create functions in untrusted languages.
If the above function was created by a superuser using the language `plperlu`, execution would succeed.

**Note:** While PL/Perl functions run in a separate Perl interpreter for each SQL role, all PL/PerlU functions executed in a given session run in a single Perl interpreter (which is not any of the ones used for PL/Perl functions). This allows PL/PerlU functions to share data freely, but no communication can occur between PL/Perl and PL/PerlU functions.

**Note:** Perl cannot support multiple interpreters within one process unless it was built with the appropriate flags, namely either `usemultiplicity` or `useithreads`. (`usemultiplicity` is preferred unless you actually need to use threads. For more details, see the `perlembed` man page.) If PL/Perl is used with a copy of Perl that was not built this way, then it is only possible to have one Perl interpreter per session, and so any one session can only execute either PL/PerlU functions, or PL/Perl functions that are all called by the same SQL role.

### 40.6. PL/Perl Triggers

PL/Perl can be used to write trigger functions. In a trigger function, the hash reference `$_TD` contains information about the current trigger event. `$_TD` is a global variable, which gets a separate local value for each invocation of the trigger. The fields of the `$_TD` hash reference are:

- `$_TD->{new}{foo}`: NEW value of column `foo`
- `$_TD->{old}{foo}`: OLD value of column `foo`
- `$_TD->{name}`: Name of the trigger being called
- `$_TD->{event}`: Trigger event: `INSERT`, `UPDATE`, `DELETE`, or `UNKNOWN`
- `$_TD->{when}`: When the trigger was called: `BEFORE`, `AFTER`, or `UNKNOWN`
- `$_TD->{level}`: The trigger level: `ROW`, `STATEMENT`, or `UNKNOWN`
- `$_TD->{relid}`: OID of the table on which the trigger fired
- `$_TD->{table_name}`: Name of the table on which the trigger fired
- `$_TD->{relname}`: Name of the table on which the trigger fired. This has been deprecated, and could be removed in a future release. Please use `$_TD->{table_name}` instead.
$_TD->{table_schema}

Name of the schema in which the table on which the trigger fired, is

$_TD->{argc}

Number of arguments of the trigger function

@{$_TD->{args}}

Arguments of the trigger function. Does not exist if $_TD->{argc} is 0.

Triggers can return one of the following:

return;

Execute the statement

"SKIP"

Don’t execute the statement

"MODIFY"

Indicates that the NEW row was modified by the trigger function

Here is an example of a trigger function, illustrating some of the above:

CREATE TABLE test (
  i int,
  v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$_
if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
  return "SKIP";  # skip INSERT/UPDATE command
} elsif ($_TD->{new}{v} ne "immortal") {
  $_TD->{new}{v} .= "(modified by trigger)";
  return "MODIFY";  # modify row and execute INSERT/UPDATE command
} else {
  return;  # execute INSERT/UPDATE command
}
$_ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
  BEFORE INSERT OR UPDATE ON test
  FOR EACH ROW EXECUTE PROCEDURE valid_id();
40.7. Limitations and Missing Features

The following features are currently missing from PL/Perl, but they would make welcome contributions.

- PL/Perl functions cannot call each other directly (because they are anonymous subroutines inside Perl).
- SPI is not yet fully implemented.
- If you are fetching very large data sets using `spi_exec_query`, you should be aware that these will all go into memory. You can avoid this by using `spi_query/spi_fetchrow` as illustrated earlier.

A similar problem occurs if a set-returning function passes a large set of rows back to PostgreSQL via `return`. You can avoid this problem too by instead using `return_next` for each row returned, as shown previously.
Chapter 41. PL/Python - Python Procedural Language

The PL/Python procedural language allows PostgreSQL functions to be written in the Python language\(^1\).

To install PL/Python in a particular database, use `createlang plpythonu dbname`.

**Tip:** If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

As of PostgreSQL 7.4, PL/Python is only available as an “untrusted” language (meaning it does not offer any way of restricting what users can do in it). It has therefore been renamed to `plpythonu`. The trusted variant `plpython` might become available again in future, if a new secure execution mechanism is developed in Python.

**Note:** Users of source packages must specially enable the build of PL/Python during the installation process. (Refer to the installation instructions for more information.) Users of binary packages might find PL/Python in a separate subpackage.

### 41.1. PL/Python Functions

Functions in PL/Python are declared via the standard `CREATE FUNCTION` syntax:

```sql
CREATE FUNCTION funcname (argument-list)
RETURNS return-type
AS $$
# PL/Python function body
$$ LANGUAGE plpythonu;
```

The body of a function is simply a Python script. When the function is called, its arguments are passed as elements of the array `args[]`; named arguments are also passed as ordinary variables to the Python script. The result is returned from the Python code in the usual way, with `return` or `yield` (in case of a result-set statement).

For example, a function to return the greater of two integers can be defined as:

```python
CREATE FUNCTION pymax (a integer, b integer)
RETURNS integer
AS $$
if a > b:
    return a
return b
$$ LANGUAGE plpythonu;
```

The Python code that is given as the body of the function definition is transformed into a Python function. For example, the above results in:

---

1. [http://www.python.org](http://www.python.org)
def __plpython_procedure_pymax_23456():
    if a > b:
        return a
    return b

assuming that 23456 is the OID assigned to the function by PostgreSQL.

The PostgreSQL function parameters are available in the global args list. In the pymax example, args[0] contains whatever was passed in as the first argument and args[1] contains the second argument’s value. Alternatively, one can use named parameters as shown in the example above. Use of named parameters is usually more readable.

If an SQL null value is passed to a function, the argument value will appear as None in Python. The above function definition will return the wrong answer for null inputs. We could add STRICT to the function definition to make PostgreSQL do something more reasonable: if a null value is passed, the function will not be called at all, but will just return a null result automatically. Alternatively, we could check for null inputs in the function body:

CREATE FUNCTION pymax (a integer, b integer)
    RETURNS integer
AS $$
if (a is None) or (b is None):
    return None
if a > b:
    return a
return b
$$ LANGUAGE plpythonu;

As shown above, to return an SQL null value from a PL/Python function, return the value None. This can be done whether the function is strict or not.

Composite-type arguments are passed to the function as Python mappings. The element names of the mapping are the attribute names of the composite type. If an attribute in the passed row has the null value, it has the value None in the mapping. Here is an example:

CREATE TABLE employee {
    name text,
    salary integer,
    age integer
};

CREATE FUNCTION overpaid (e employee)
    RETURNS boolean
AS $$
if e["salary"] > 200000:
    return True
if (e["age"] < 30) and (e["salary"] > 100000):
    return True
return False
$$ LANGUAGE plpythonu;

There are multiple ways to return row or composite types from a Python function. The following examples assume we have:

CREATE TYPE named_value AS {
    name text,
A composite result can be returned as a:

Sequence type (a tuple or list, but not a set because it is not indexable)

  Returned sequence objects must have the same number of items as the composite result type has
  fields. The item with index 0 is assigned to the first field of the composite type, 1 to the second
  and so on. For example:

  CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
  AS $$_
  return [ name, value ]
  # or alternatively, as tuple: return ( name, value )
  $$ LANGUAGE plpythonu;

  To return a SQL null for any column, insert None at the corresponding position.

Mapping (dictionary)

  The value for each result type column is retrieved from the mapping with the column name as
  key. Example:

  CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
  AS $$_
  return { "name": name, "value": value }
  $$ LANGUAGE plpythonu;

  Any extra dictionary key/value pairs are ignored. Missing keys are treated as errors. To return a
  SQL null value for any column, insert None with the corresponding column name as the key.

Object (any object providing method __getattr__)

  This works the same as a mapping. Example:

  CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
  AS $$_
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
    return named_value(name, value)
  
  # or simply
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
  $$ LANGUAGE plpythonu;

If you do not provide a return value, Python returns the default None. PL/Python translates Python’s
None into the SQL null value.

A PL/Python function can also return sets of scalar or composite types. There are several ways to
achieve this because the returned object is internally turned into an iterator. The following examples
assume we have composite type:
CREATE TYPE greeting AS (  
    how text,  
    who text  
);  

A set result can be returned from a:

Sequence type (tuple, list, set)

CREATE FUNCTION greet (how text)  
RETURNS SETOF greeting  
AS $$  
# return tuple containing lists as composite types  
# all other combinations work also  
return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )  $$ LANGUAGE plpythonu;

Iterator (any object providing __iter__ and next methods)

CREATE FUNCTION greet (how text)  
RETURNS SETOF greeting  
AS $$  
class producer:  
    def __init__ (self, how, who):  
        self.how = how  
        self.who = who  
        self.ndx = -1  
    
    def __iter__ (self):  
        return self  
    
    def next (self):  
        self.ndx += 1  
        if self.ndx == len(self.who):  
            raise StopIteration  
        return ( self.how, self.who[self.ndx] )  

return producer(how, [ "World", "PostgreSQL", "PL/Python" ])  $$ LANGUAGE plpythonu;

Generator (yield)

CREATE FUNCTION greet (how text)  
RETURNS SETOF greeting  
AS $$  
for who in [ "World", "PostgreSQL", "PL/Python" ]:  
    yield ( how, who )  $$ LANGUAGE plpythonu;

Warning
Currently, due to Python bug #1483133, some debug versions of Python 2.4 (configured and compiled with option --with-pydebug) are known to crash the PostgreSQL server when using an iterator to return a set result. Unpatched versions of Fedora 4 contain this bug. It does not happen in production versions of Python or on patched versions of Fedora 4.
The global dictionary SD is available to store data between function calls. This variable is private static data. The global dictionary GD is public data, available to all Python functions within a session. Use with care.

Each function gets its own execution environment in the Python interpreter, so that global data and function arguments from myfunc are not available to myfunc2. The exception is the data in the GD dictionary, as mentioned above.

### 41.2. Trigger Functions

When a function is used as a trigger, the dictionary TD contains trigger-related values. The trigger rows are in TD["new"] and/or TD["old"] depending on the trigger event. TD["event"] contains the event as a string (INSERT, UPDATE, DELETE, or UNKNOWN). TD["when"] contains one of BEFORE, AFTER, and UNKNOWN. TD["level"] contains one of ROW, STATEMENT, and UNKNOWN. TD["name"] contains the trigger name, TD["table_name"] contains the name of the table on which the trigger occurred, TD["table_schema"] contains the schema of the table on which the trigger occurred, TD["name"] contains the trigger name, and TD["relid"] contains the OID of the table on which the trigger occurred. If the CREATE TRIGGER command included arguments, they are available in TD["args"][0] to TD["args"][(n-1)].

If TD["when"] is BEFORE, you can return None or "OK" from the Python function to indicate the row is unmodified, "SKIP" to abort the event, or "MODIFY" to indicate you've modified the row.

### 41.3. Database Access

The PL/Python language module automatically imports a Python module called plpy. The functions and constants in this module are available to you in the Python code as plpy.foo. At present plpy implements the functions plpy.debug(msg), plpy.log(msg), plpy.info(msg), plpy.notice(msg), plpy.warning(msg), plpy.error(msg), and plpy.fatal(msg). plpy.error and plpy.fatal actually raise a Python exception which, if uncaught, propagates out to the calling query, causing the current transaction or subtransaction to be aborted. raise plpy.ERROR(msg) and raise plpy.FATAL(msg) are equivalent to calling plpy.error and plpy.fatal, respectively. The other functions only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

Additionally, the plpy module provides two functions called execute and prepare. Calling plpy.execute with a query string and an optional limit argument causes that query to be run and the result to be returned in a result object. The result object emulates a list or dictionary object. The result object can be accessed by row number and column name. It has these additional methods: nrows which returns the number of rows returned by the query, and status which is the SPI_execute() return value. The result object can be modified.

For example:

```python
rv = plpy.execute("SELECT * FROM my_table", 5)
```
returns up to 5 rows from my_table. If my_table has a column my_column, it would be accessed as:

```python
foo = rv[i]["my_column"]
```

The second function, `plpy.prepare`, prepares the execution plan for a query. It is called with a query string and a list of parameter types, if you have parameter references in the query. For example:

```python
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", [ "text" ])
```

text is the type of the variable you will be passing for $1. After preparing a statement, you use the function `plpy.execute` to run it:

```python
rv = plpy.execute(plan, [ "name" ], 5)
```

The third argument is the limit and is optional.

When you prepare a plan using the PL/Python module it is automatically saved. Read the SPI documentation (Chapter 42) for a description of what this means. In order to make effective use of this across function calls one needs to use one of the persistent storage dictionaries SD or GD (see Section 41.1). For example:

```sql
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
if SD.has_key("plan"):
    plan = SD["plan"]
else:
    plan = plpy.prepare("SELECT 1")
    SD["plan"] = plan
# rest of function
$$ LANGUAGE plpythonu;
```
Chapter 42. Server Programming Interface

The Server Programming Interface (SPI) gives writers of user-defined C functions the ability to run SQL commands inside their functions. SPI is a set of interface functions to simplify access to the parser, planner, and executor. SPI also does some memory management.

Note: The available procedural languages provide various means to execute SQL commands from procedures. Most of these facilities are based on SPI, so this documentation might be of use for users of those languages as well.

To avoid misunderstanding we’ll use the term “function” when we speak of SPI interface functions and “procedure” for a user-defined C-function that is using SPI.

Note that if a command invoked via SPI fails, then control will not be returned to your procedure. Rather, the transaction or subtransaction in which your procedure executes will be rolled back. (This might seem surprising given that the SPI functions mostly have documented error-return conventions. Those conventions only apply for errors detected within the SPI functions themselves, however.) It is possible to recover control after an error by establishing your own subtransaction surrounding SPI calls that might fail. This is not currently documented because the mechanisms required are still in flux.

SPI functions return a nonnegative result on success (either via a returned integer value or in the global variable SPI_result, as described below). On error, a negative result or NULL will be returned.

Source code files that use SPI must include the header file executor/spi.h.

42.1. Interface Functions

SPI_connect

Name

SPI_connect — connect a procedure to the SPI manager

Synopsis

int SPI_connect(void)

Description

SPI_connect opens a connection from a procedure invocation to the SPI manager. You must call this function if you want to execute commands through SPI. Some utility SPI functions can be called from unconnected procedures.

If your procedure is already connected, SPI_connect will return the error code SPI_ERROR_CONNECT. This could happen if a procedure that has called SPI_connect directly
calls another procedure that calls `SPI_connect`. While recursive calls to the SPI manager are permitted when an SQL command called through SPI invokes another function that uses SPI, directly nested calls to `SPI_connect` and `SPI_finish` are forbidden. (But see `SPI_push` and `SPI_pop`.)

### Return Value

- **SPI_OK_CONNECT**
  - on success
- **SPI_ERROR_CONNECT**
  - on error
**SPI_finish**

**Name**

SPI_finish — disconnect a procedure from the SPI manager

**Synopsis**

```
int SPI_finish(void)
```

**Description**

SPI_finish closes an existing connection to the SPI manager. You must call this function after completing the SPI operations needed during your procedure’s current invocation. You do not need to worry about making this happen, however, if you abort the transaction via `elog(ERROR)`. In that case SPI will clean itself up automatically.

If SPI_finish is called without having a valid connection, it will return `SPI_ERROR_UNCONNECTED`. There is no fundamental problem with this; it means that the SPI manager has nothing to do.

**Return Value**

SPI_OK_FINISH

    if properly disconnected

SPI_ERROR_UNCONNECTED

    if called from an unconnected procedure
SPI_push

Name

SPI_push — push SPI stack to allow recursive SPI usage

Synopsis

void SPI_push(void)

Description

SPI_push should be called before executing another procedure that might itself wish to use SPI. After
SPI_push, SPI is no longer in a “connected” state, and SPI function calls will be rejected unless a
fresh SPI_connect is done. This ensures a clean separation between your procedure’s SPI state and
that of another procedure you call. After the other procedure returns, call SPI_pop to restore access
to your own SPI state.

Note that SPI_execute and related functions automatically do the equivalent of SPI_push before
passing control back to the SQL execution engine, so it is not necessary for you to worry about this
when using those functions. Only when you are directly calling arbitrary code that might contain
SPI_connect calls do you need to issue SPI_push and SPI_pop.
SPI_pop

Name
SPI_pop — pop SPI stack to return from recursive SPI usage

Synopsis
void SPI_pop(void)

Description
SPI_pop pops the previous environment from the SPI call stack. See SPI_push.
SPI_execute

Name
SPI_execute — execute a command

Synopsis
int SPI_execute(const char * command, bool read_only, long count)

Description
SPI_execute executes the specified SQL command for count rows. If read_only is true, the command must be read-only, and execution overhead is somewhat reduced.

This function can only be called from a connected procedure.

If count is zero then the command is executed for all rows that it applies to. If count is greater than 0, then the number of rows for which the command will be executed is restricted (much like a LIMIT clause). For example:

SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);

will allow at most 5 rows to be inserted into the table.

You can pass multiple commands in one string, but later commands cannot depend on the creation of objects earlier in the string, because the whole string will be parsed and planned before execution begins. SPI_execute returns the result for the command executed last. The count limit applies to each command separately, but it is not applied to hidden commands generated by rules.

When read_only is false, SPI_execute increments the command counter and computes a new snapshot before executing each command in the string. The snapshot does not actually change if the current transaction isolation level is SERIALIZABLE, but in READ COMMITTED mode the snapshot update allows each command to see the results of newly committed transactions from other sessions. This is essential for consistent behavior when the commands are modifying the database.

When read_only is true, SPI_execute does not update either the snapshot or the command counter, and it allows only plain SELECT commands to appear in the command string. The commands are executed using the snapshot previously established for the surrounding query. This execution mode is somewhat faster than the read/write mode due to eliminating per-command overhead. It also allows genuinely stable functions to be built: since successive executions will all use the same snapshot, there will be no change in the results.

It is generally unwise to mix read-only and read-write commands within a single function using SPI; that could result in very confusing behavior, since the read-only queries would not see the results of any database updates done by the read-write queries.

The actual number of rows for which the (last) command was executed is returned in the global variable SPI_processed. If the return value of the function is SPI_OK_SELECT, SPI_OK_INSERT_RETURNING, SPI_OK_DELETE_RETURNING, or SPI_OK_UPDATE_RETURNING, then you can use the global pointer SPITupleTable *SPI_tuptable to access the result rows. Some utility commands (such as EXPLAIN) also return row sets, and SPI_tuptable will contain the result in these cases too.
SPI_execute

The structure SPITupleTable is defined thus:

typedef struct
{
    MemoryContext tuptabcxt; /* memory context of result table */
    uint32 alloced; /* number of allocated vals */
    uint32 free; /* number of free vals */
    TupleDesc tupdesc; /* row descriptor */
    HeapTuple *vals; /* rows */
} SPITupleTable;

vals is an array of pointers to rows. (The number of valid entries is given by SPI_processed.)
tupdesc is a row descriptor which you can pass to SPI functions dealing with rows.
tuptabcxt, alloced, and free are internal fields not intended for use by SPI callers.
SPI_finish frees all SPITupleTables allocated during the current procedure. You can free a particular result table earlier, if you are done with it, by calling SPI_freetuptable.

Arguments

const char * command
    string containing command to execute
bool read_only
    true for read-only execution
long count
    maximum number of rows to process or return

Return Value

If the execution of the command was successful then one of the following (nonnegative) values will be returned:

SPI_OK_SELECT
    if a SELECT (but not SELECT INTO) was executed
SPI_OK_SELINTO
    if a SELECT INTO was executed
SPI_OK_INSERT
    if an INSERT was executed
SPI_OK_DELETE
    if a DELETE was executed
SPI_OK_UPDATE
    if an UPDATE was executed
SPI_OK_INSERT_RETURNING
    if an INSERT RETURNING was executed
SPI_OK_DELETE_RETURNING
    if a DELETE RETURNING was executed

SPI_OK_UPDATE_RETURNING
    if an UPDATE RETURNING was executed

SPI_OKUTILITY
    if a utility command (e.g., CREATE TABLE) was executed

On error, one of the following negative values is returned:

SPI_ERROR_ARGUMENT
    if command is NULL or count is less than 0

SPI_ERROR_COPY
    if COPY TO stdout or COPY FROM stdin was attempted

SPI_ERROR_TRANSACTION
    if a transaction manipulation command was attempted (BEGIN, COMMIT, ROLLBACK, SAVEPOINT, PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED, or any variant thereof)

SPI_ERROR_OPUNKNOWN
    if the command type is unknown (shouldn’t happen)

SPI_ERROR_UNCONNECTED
    if called from an unconnected procedure

Notes
The functions SPI_execute, SPI_exec, SPI_execute_plan, and SPI_execp change both SPI_processed and SPI_tuptable (just the pointer, not the contents of the structure). Save these two global variables into local procedure variables if you need to access the result table of SPI_execute or a related function across later calls.
SPI_exec

Name
SPI_exec — execute a read/write command

Synopsis
int SPI_exec(const char * command, long count)

Description
SPI_exec is the same as SPI_execute, with the latter's read_only parameter always taken as false.

Arguments
const char * command
    string containing command to execute
long count
    maximum number of rows to process or return

Return Value
See SPI_execute.
**SPI_prepare**

**Name**
SPI_prepare — prepare a plan for a command, without executing it yet

**Synopsis**
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)

**Description**
SPI_prepare creates and returns an execution plan for the specified command but doesn’t execute the command. This function should only be called from a connected procedure.

When the same or a similar command is to be executed repeatedly, it might be advantageous to perform the planning only once. SPI_prepare converts a command string into an execution plan that can be executed repeatedly using SPI_execute_plan.

A prepared command can be generalized by writing parameters ($1, $2, etc.) in place of what would be constants in a normal command. The actual values of the parameters are then specified when SPI_execute_plan is called. This allows the prepared command to be used over a wider range of situations than would be possible without parameters.

The plan returned by SPI_prepare can be used only in the current invocation of the procedure, since SPI_finish frees memory allocated for a plan. But a plan can be saved for longer using the function SPI_saveplan.

**Arguments**

**const char * command**
command string

**int nargs**
number of input parameters ($1, $2, etc.)

**Oid * argtypes**
pointer to an array containing the OIDs of the data types of the parameters

**Return Value**
SPI_prepare returns a non-null pointer to an execution plan. On error, NULL will be returned, and SPI_result will be set to one of the same error codes used by SPI_execute, except that it is set to SPI_ERROR_ARGUMENT if command is NULL, or if nargs is less than 0, or if nargs is greater than 0 and argtypes is NULL.
Notes

SPIPlanPtr is declared as a pointer to an opaque struct type in spi.h. It is unwise to try to access its contents directly, as that makes your code much more likely to break in future revisions of PostgreSQL.

There is a disadvantage to using parameters: since the planner does not know the values that will be supplied for the parameters, it might make worse planning choices than it would make for a normal command with all constants visible.
SPI_prepare_cursor

Name
SPI_prepare_cursor — prepare a plan for a command, without executing it yet

Synopsis
SPIPlanPtr SPI_prepare_cursor(const char *command, int nargs, Oid *argtypes, int cursorOptions)

Description
SPI_prepare_cursor is identical to SPI_prepare, except that it also allows specification of the planner’s “cursor options” parameter. This is a bitmask having the values shown in nodes/parsenodes.h for the options field of DeclareCursorStmt. SPI_prepare always takes these options as zero.

Arguments
const char * command
command string
int nargs
number of input parameters ($1, $2, etc.)
Oid * argtypes
pointer to an array containing the OIDs of the data types of the parameters
int cursorOptions
integer bitmask of cursor options; zero produces default behavior

Return Value
SPI_prepare_cursor has the same return conventions as SPI_prepare.

Notes
Useful bits to set in cursorOptions include CURSOR_OPT_SCROLL, CURSOR_OPT_NO_SCROLL, and CURSOR_OPT_FAST_PLAN. Note in particular that CURSOR_OPT_HOLD is ignored.
**SPI_getargcount**

**Name**

SPI_getargcount — return the number of arguments needed by a plan prepared by SPI_prepare

**Synopsis**

```c
int SPI_getargcount(SPIPlanPtr plan)
```

**Description**

SPI_getargcount returns the number of arguments needed to execute a plan prepared by SPI_prepare.

**Arguments**

SPIPlanPtr plan

execution plan (returned by SPI_prepare)

**Return Value**

The count of expected arguments for the plan. If the plan is NULL or invalid, SPI_result is set to SPI_ERROR_ARGUMENT and -1 is returned.
**SPI_getargtypeid**

**Name**

SPI_getargtypeid — return the data type OID for an argument of a plan prepared by SPI_prepare

**Synopsis**

Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)

**Description**

SPI_getargtypeid returns the OID representing the type id for the argIndex'th argument of a plan prepared by SPI_prepare. First argument is at index zero.

**Arguments**

SPIPlanPtr plan

execution plan (returned by SPI_prepare)

int argIndex

zero based index of the argument

**Return Value**

The type id of the argument at the given index. If the plan is NULL or invalid, or argIndex is less than 0 or not less than the number of arguments declared for the plan, SPI_result is set to SPI_ERROR_ARGUMENT and InvalidOid is returned.
SPI_is_cursor_plan

Name
SPI_is_cursor_plan — return true if a plan prepared by SPI_prepare can be used with SPI_cursor_open

Synopsis
bool SPI_is_cursor_plan(SPIPlanPtr plan)

Description
SPI_is_cursor_plan returns true if a plan prepared by SPI_prepare can be passed as an argument to SPI_cursor_open, or false if that is not the case. The criteria are that the plan represents one single command and that this command returns tuples to the caller; for example, SELECT is allowed unless it contains an INTO clause, and UPDATE is allowed only if it contains a RETURNING clause.

Arguments
SPIPlanPtr plan
   execution plan (returned by SPI_prepare)

Return Value
ture or false to indicate if the plan can produce a cursor or not, with SPI_result set to zero. If it is not possible to determine the answer (for example, if the plan is NULL or invalid, or if called when not connected to SPI), then SPI_result is set to a suitable error code and false is returned.
**SPI_execute_plan**

**Name**

SPI_execute_plan — execute a plan prepared by SPI_prepare

**Synopsis**

```c
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                      bool read_only, long count)
```

**Description**

SPI_execute_plan executes a plan prepared by SPI_prepare. `read_only` and `count` have the same interpretation as in SPI_execute.

**Arguments**

- **SPIPlanPtr plan**
  - execution plan (returned by SPI_prepare)

- **Datum * values**
  - An array of actual parameter values. Must have same length as the plan's number of arguments.

- **const char * nulls**
  - An array describing which parameters are null. Must have same length as the plan’s number of arguments. `n` indicates a null value (entry in `values` will be ignored); a space indicates anonnull value (entry in `values` is valid).

  If `nulls` is NULL then SPI_execute_plan assumes that no parameters are null.

- **bool read_only**
  - true for read-only execution

- **long count**
  - maximum number of rows to process or return

**Return Value**

The return value is the same as for SPI_execute, with the following additional possible error (negative) results:

SPI_ERROR_ARGUMENT

if `plan` is NULL or invalid, or `count` is less than 0
SPI_ERROR_PARAM

if values is NULL and plan was prepared with some parameters

SPI_processed and SPI_tuptable are set as in SPI_execute if successful.

Notes

If one of the objects (a table, function, etc.) referenced by the prepared plan is dropped during the session then the result of SPI_execute_plan for this plan will be unpredictable.
SPI_execp

Name

SPI_execp — execute a plan in read/write mode

Synopsis

int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)

Description

SPI_execp is the same as SPI_execute_plan, with the latter’s read_only parameter always taken as false.

Arguments

SPIPlanPtr plan
   execution plan (returned by SPI_prepare)
Datum * values
   An array of actual parameter values. Must have same length as the plan’s number of arguments.
const char * nulls
   An array describing which parameters are null. Must have same length as the plan’s number of arguments. n indicates a null value (entry in values will be ignored); a space indicates a nonnull value (entry in values is valid).
   If nulls is NULL then SPI_execp assumes that no parameters are null.
long count
   maximum number of rows to process or return

Return Value

See SPI_execute_plan.
SPI_processed and SPI_tuptable are set as in SPI_execute if successful.
SPI_cursor_open

Name
SPI_cursor_open — set up a cursor using a plan created with SPI_prepare

Synopsis
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
    Datum * values, const char * nulls,
    bool read_only)

Description
SPI_cursor_open sets up a cursor (internally, a portal) that will execute a plan prepared by
SPI_prepare. The parameters have the same meanings as the corresponding parameters to
SPI_execute_plan.

Using a cursor instead of executing the plan directly has two benefits. First, the result rows can be
retrieved a few at a time, avoiding memory overrun for queries that return many rows. Second, a portal
can outlive the current procedure (it can, in fact, live to the end of the current transaction). Returning
the portal name to the procedure’s caller provides a way of returning a row set as result.

The passed-in data will be copied into the cursor’s portal, so it can be freed while the cursor still
exists.

Arguments

const char * name
    name for portal, or NULL to let the system select a name

SPIPlanPtr plan
    execution plan (returned by SPI_prepare)

Datum * values
    An array of actual parameter values. Must have same length as the plan’s number of arguments.

const char * nulls
    An array describing which parameters are null. Must have same length as the plan’s number of
    arguments. n indicates a null value (entry in values will be ignored); a space indicates a nonnull
    value (entry in values is valid).

    If nulls is NULL then SPI_cursor_open assumes that no parameters are null.

bool read_only
    true for read-only execution
Return Value

Pointer to portal containing the cursor. Note there is no error return convention; any error will be reported via `elog`. 
SPI_cursor_find

**Name**
SPI_cursor_find — find an existing cursor by name

**Synopsis**
Portal SPI_cursor_find(const char * name)

**Description**
SPI_cursor_find finds an existing portal by name. This is primarily useful to resolve a cursor name returned as text by some other function.

**Arguments**

const char * name

name of the portal

**Return Value**

pointer to the portal with the specified name, or NULL if none was found
Name
SPI_cursor_fetch — fetch some rows from a cursor

Synopsis
void SPI_cursor_fetch(Portal portal, bool forward, long count)

Description
SPI_cursor_fetch fetches some rows from a cursor. This is equivalent to a subset of the SQL command FETCH (see SPI_scroll_cursor_fetch for more functionality).

Arguments
Portal portal
    portal containing the cursor
bool forward
    true for fetch forward, false for fetch backward
long count
    maximum number of rows to fetch

Return Value
SPI_processed and SPI_tuptable are set as in SPI_execute if successful.

Notes
Fetching backward may fail if the cursor’s plan was not created with the CURSOR_OPT_SCROLL option.
SPI_cursor_move

Name
SPI_cursor_move — move a cursor

Synopsis
void SPI_cursor_move(Portal portal, bool forward, long count)

Description
SPI_cursor_move skips over some number of rows in a cursor. This is equivalent to a subset of the SQL command MOVE (see SPI_scroll_cursor_move for more functionality).

Arguments
Portal portal
    portal containing the cursor
bool forward
    true for move forward, false for move backward
long count
    maximum number of rows to move

Notes
Moving backward may fail if the cursor’s plan was not created with the CURSOR_OPT_SCROLL option.
SPI_scroll_cursor_fetch

Name
SPI_scroll_cursor_fetch — fetch some rows from a cursor

Synopsis
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction, long count)

Description
SPI_scroll_cursor_fetch fetches some rows from a cursor. This is equivalent to the SQL command FETCH.

Arguments
Portal portal
portal containing the cursor
FetchDirection direction
one of FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE or FETCH_RELATIVE
long count
number of rows to fetch for FETCH_FORWARD or FETCH_BACKWARD; absolute row number to fetch for FETCH_ABSOLUTE; or relative row number to fetch for FETCH_RELATIVE

Return Value
SPI_processed and SPI_tuptable are set as in SPI_execute if successful.

Notes
See the SQL FETCH command for details of the interpretation of the direction and count parameters.
Direction values other than FETCH_FORWARD may fail if the cursor’s plan was not created with the CURSOR_OPT_SCROLL option.
SPI_scroll_cursor_move

Name
SPI_scroll_cursor_move — move a cursor

Synopsis
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction, long count)

Description
SPI_scroll_cursor_move skips over some number of rows in a cursor. This is equivalent to the SQL command MOVE.

Arguments
Portal portal
    portal containing the cursor
FetchDirection direction
    one of FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE or FETCH_RELATIVE
long count
    number of rows to move for FETCH_FORWARD or FETCH_BACKWARD; absolute row number to move to for FETCH_ABSOLUTE; or relative row number to move to for FETCH_RELATIVE

Return Value
SPI_processed is set as in SPI_execute if successful. SPI_tuptable is set to NULL, since no rows are returned by this function.

Notes
See the SQL FETCH command for details of the interpretation of the direction and count parameters.
Direction values other than FETCH_FORWARD may fail if the cursor’s plan was not created with the CURSOR_OPT_SCROLL option.
SPI_cursor_close

Name
SPI_cursor_close — close a cursor

Synopsis
void SPI_cursor_close(Portal portal)

Description
SPI_cursor_close closes a previously created cursor and releases its portal storage.
All open cursors are closed automatically at the end of a transaction. SPI_cursor_close need only be invoked if it is desirable to release resources sooner.

Arguments
Portal portal
   portal containing the cursor
SPI_saveplan

Name
SPI_saveplan — save a plan

Synopsis
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)

Description
SPI_saveplan saves a passed plan (prepared by SPI_prepare) in memory that will not be freed by SPI_finish nor by the transaction manager, and returns a pointer to the saved plan. This gives you the ability to reuse prepared plans in the subsequent invocations of your procedure in the current session.

Arguments
SPIPlanPtr plan
the plan to be saved

Return Value
Pointer to the saved plan; NULL if unsuccessful. On error, SPI_result is set thus:

SPI_ERROR_ARGUMENT
if plan is NULL or invalid
SPI_ERROR_UNCONNECTED
if called from an unconnected procedure

Notes
The passed-in plan is not freed, so you might wish to do SPI_freeplan on it to avoid leaking memory until SPI_finish.

If one of the objects (a table, function, etc.) referenced by the prepared plan is dropped or redefined, then future executions of SPI_execute_plan may fail or return different results than the plan initially indicates.
42.2. Interface Support Functions

The functions described here provide an interface for extracting information from result sets returned by SPI_execute and other SPI functions.

All functions described in this section can be used by both connected and unconnected procedures.

**SPI.fname**

**Name**

SPI_fname — determine the column name for the specified column number

**Synopsis**

```c
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

**Description**

SPI_fname returns a copy of the column name of the specified column. (You can use pfree to release the copy of the name when you don’t need it anymore.)

**Arguments**

- `TupleDesc rowdesc`: input row description
- `int colnumber`: column number (count starts at 1)

**Return Value**

The column name; NULL if colnumber is out of range. SPI_result set to SPI_ERROR_NOATTRIBUTE on error.
SPI_fnumber

Name
SPI_fnumber — determine the column number for the specified column name

Synopsis
int SPI_fnumber(TupleDesc rowdesc, const char * colname)

Description
SPI_fnumber returns the column number for the column with the specified name.
If colname refers to a system column (e.g., oid) then the appropriate negative column number will be returned. The caller should be careful to test the return value for exact equality to SPI_ERROR_NOATTRIBUTE to detect an error; testing the result for less than or equal to 0 is not correct unless system columns should be rejected.

Arguments

TupleDesc rowdesc
    input row description
const char * colname
    column name

Return Value
Column number (count starts at 1), or SPI_ERROR_NOATTRIBUTE if the named column was not found.
SPI_getvalue

Name
SPI_getvalue — return the string value of the specified column

Synopsis
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)

Description
SPI_getvalue returns the string representation of the value of the specified column.
The result is returned in memory allocated using palloc. (You can use pfree to release the memory
when you don’t need it anymore.)

Arguments
HeapTuple row
   input row to be examined
TupleDesc rowdesc
   input row description
int colnumber
   column number (count starts at 1)

Return Value
Column value, or NULL if the column is null, colnumber is out of range (SPI_result is set
to SPI_ERROR_NOATTRIBUTE), or no output function is available (SPI_result is set to
SPI_ERROR_NOOUTFUNC).
SPI_getbinval

Name

SPI_getbinval — return the binary value of the specified column

Synopsis

Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber, bool * isnull)

Description

SPI_getbinval returns the value of the specified column in the internal form (as type Datum).
This function does not allocate new space for the datum. In the case of a pass-by-reference data type, the return value will be a pointer into the passed row.

Arguments

HeapTuple row
    input row to be examined
TupleDesc rowdesc
    input row description
int colnumber
    column number (count starts at 1)
bool * isnull
    flag for a null value in the column

Return Value

The binary value of the column is returned. The variable pointed to by isnull is set to true if the column is null, else to false.
SPI_result is set to SPI_ERROR_NOATTRIBUTE on error.
**SPI_gettype**

**Name**

SPI_gettype — return the data type name of the specified column

**Synopsis**

```c
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

**Description**

SPI_gettype returns a copy of the data type name of the specified column. (You can use pfree to release the copy of the name when you don’t need it anymore.)

**Arguments**

- **TupleDesc rowdesc**
  - input row description
- **int colnumber**
  - column number (count starts at 1)

**Return Value**

The data type name of the specified column, or NULL on error. SPI_result is set to SPI_ERROR_NOATTRIBUTE on error.
SPI_gettypeid

Name
SPI_gettypeid — return the data type OID of the specified column

Synopsis
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)

Description
SPI_gettypeid returns the OID of the data type of the specified column.

Arguments
TupleDesc rowdesc
   input row description
int colnumber
   column number (count starts at 1)

Return Value
The OID of the data type of the specified column or InvalidOid on error. On error, SPI_result is set to SPI_ERROR_NOATTRIBUTE.
SPI_getrelname

Name
SPI_getrelname — return the name of the specified relation

Synopsis
char * SPI_getrelname(Relation rel)

Description
SPI_getrelname returns a copy of the name of the specified relation. (You can use pfree to release the copy of the name when you don’t need it anymore.)

Arguments
Relation rel
   input relation

Return Value
The name of the specified relation.
SPI_getnspname

Name
SPI_getnspname — return the namespace of the specified relation

Synopsis
char * SPI_getnspname(Relation rel)

Description
SPI_getnspname returns a copy of the name of the namespace that the specified Relation belongs to. This is equivalent to the relation’s schema. You should `pfree` the return value of this function when you are finished with it.

Arguments

Relation rel
input relation

Return Value
The name of the specified relation’s namespace.
42.3. Memory Management

PostgreSQL allocates memory within memory contexts, which provide a convenient method of managing allocations made in many different places that need to live for differing amounts of time. Destroying a context releases all the memory that was allocated in it. Thus, it is not necessary to keep track of individual objects to avoid memory leaks; instead only a relatively small number of contexts have to be managed. `palloc` and related functions allocate memory from the “current” context.

`SPI_connect` creates a new memory context and makes it current. `SPI_finish` restores the previous current memory context and destroys the context created by `SPI_connect`. These actions ensure that transient memory allocations made inside your procedure are reclaimed at procedure exit, avoiding memory leakage.

However, if your procedure needs to return an object in allocated memory (such as a value of a pass-by-reference data type), you cannot allocate that memory using `palloc`, at least not while you are connected to SPI. If you try, the object will be deallocated by `SPI_finish`, and your procedure will not work reliably. To solve this problem, use `SPI_palloc` to allocate memory for your return object.

If `SPI_palloc` is called while the procedure is not connected to SPI, then it acts the same as a normal `palloc`. Before a procedure connects to the SPI manager, the current memory context is the upper executor context, so all allocations made by the procedure via `palloc` or by SPI utility functions are made in this context.

When `SPI_connect` is called, the private context of the procedure, which is created by `SPI_connect`, is made the current context. All allocations made by `palloc`, `realloc`, or SPI utility functions (except for `SPI_copytuple`, `SPI_returntuple`, `SPI_modifytuple`, and `SPI_palloc`) are made in this context. When a procedure disconnects from the SPI manager (via `SPI_finish`) the current context is restored to the upper executor context, and all allocations made in the procedure memory context are freed and cannot be used any more.

All functions described in this section can be used by both connected and unconnected procedures. In an unconnected procedure, they act the same as the underlying ordinary server functions (`palloc`, etc.).

SPI_palloc

Name

SPI_palloc — allocate memory in the upper executor context

Synopsis

```c
void * SPI_palloc(Size size)
```

Description

`SPI_palloc` allocates memory in the upper executor context.
**Arguments**

Size size

size in bytes of storage to allocate

**Return Value**

pointer to new storage space of the specified size
**SPI_repalloc**

**Name**

SPI_repalloc — reallocate memory in the upper executor context

**Synopsis**

```c
void * SPI_repalloc(void * pointer, Size size)
```

**Description**

SPI_repalloc changes the size of a memory segment previously allocated using SPI_palloc. This function is no longer different from plain realloc. It’s kept just for backward compatibility of existing code.

**Arguments**

- `void * pointer`
  - pointer to existing storage to change
- `Size size`
  - size in bytes of storage to allocate

**Return Value**

pointer to new storage space of specified size with the contents copied from the existing area
SPI_pfree

Name

SPI_pfree — free memory in the upper executor context

Synopsis

void SPI_pfree(void * pointer)

Description

SPI_pfree frees memory previously allocated using SPI_palloc or SPI_repalloc.
This function is no longer different from plain pfree. It’s kept just for backward compatibility of existing code.

Arguments

void * pointer

pointer to existing storage to free
SPI_copytuple

Name

SPI_copytuple — make a copy of a row in the upper executor context

Synopsis

HeapTuple SPI_copytuple(HeapTuple row)

Description

SPI_copytuple makes a copy of a row in the upper executor context. This is normally used to return a modified row from a trigger. In a function declared to return a composite type, use SPI_returntuple instead.

Arguments

HeapTuple row
row to be copied

Return Value

copied row; NULL only if tuple is NULL
SPI_returntuple

Name
SPI_returntuple — prepare to return a tuple as a Datum

Synopsis
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)

Description
SPI_returntuple makes a copy of a row in the upper executor context, returning it in the form of a row type Datum. The returned pointer need only be converted to Datum via PointerGetDatum before returning.

Note that this should be used for functions that are declared to return composite types. It is not used for triggers; use SPI_copytuple for returning a modified row in a trigger.

Arguments
HeapTuple row
row to be copied
TupleDesc rowdesc
descriptor for row (pass the same descriptor each time for most effective caching)

Return Value
HeapTupleHeader pointing to copied row; NULL only if row or rowdesc is NULL
SPI_modifytuple

Name
SPI_modifytuple — create a row by replacing selected fields of a given row

Synopsis
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, ncols, colnum, Datum * values, const char * Nulls)

Description
SPI_modifytuple creates a new row by substituting new values for selected columns, copying the original row’s columns at other positions. The input row is not modified.

Arguments
Relation rel
Used only as the source of the row descriptor for the row. (Passing a relation rather than a row descriptor is a misfeature.)

HeapTuple row
row to be modified

int ncols
number of column numbers in the array colnum

int * colnum
array of the numbers of the columns that are to be changed (column numbers start at 1)

Datum * values
new values for the specified columns

const char * Nulls
which new values are null, if any (see SPI_execute_plan for the format)

Return Value
new row with modifications, allocated in the upper executor context; NULL only if row is NULL

On error, SPI_result is set as follows:

SPI_ERROR_ARGUMENT
if rel is NULL, or if row is NULL, or if ncols is less than or equal to 0, or if colnum is NULL, or if values is NULL.
SPI_ERROR_NOATTRIBUTE

if colnum contains an invalid column number (less than or equal to 0 or greater than the number of column in row)
SPI_freetuple

Name
SPI_freetuple — free a row allocated in the upper executor context

Synopsis
void SPI_freetuple(HeapTuple row)

Description
SPI_freetuple frees a row previously allocated in the upper executor context.
This function is no longer different from plain heap_freetuple. It’s kept just for backward compatibility of existing code.

Arguments
HeapTuple row
row to free
**SPI_freetuptable**

**Name**

SPI_freetuptable — free a row set created by SPI_execute or a similar function

**Synopsis**

void SPI_freetuptable(SPITupleTable * tuptable)

**Description**

SPI_freetuptable frees a row set created by a prior SPI command execution function, such as SPI_execute. Therefore, this function is usually called with the global variable SPI_tupletable as argument.

This function is useful if a SPI procedure needs to execute multiple commands and does not want to keep the results of earlier commands around until it ends. Note that any unfreed row sets will be freed anyway at SPI_finish.

**Arguments**

SPITupleTable * tuptable

pointer to row set to free
SPI_freeplan

Name
SPI_freeplan — free a previously saved plan

Synopsis
int SPI_freeplan(SPIPlanPtr plan)

Description
SPI_freeplan releases a command execution plan previously returned by SPI_prepare or saved by SPI_saveplan.

Arguments
SPIPlanPtr plan
pointer to plan to free

Return Value
SPI_ERROR_ARGUMENT if plan is NULL or invalid
### 42.4. Visibility of Data Changes

The following rules govern the visibility of data changes in functions that use SPI (or any other C function):

- During the execution of an SQL command, any data changes made by the command are invisible to the command itself. For example, in:
  
  ```sql
  INSERT INTO a SELECT * FROM a;
  the inserted rows are invisible to the SELECT part.
  ```

- Changes made by a command C are visible to all commands that are started after C, no matter whether they are started inside C (during the execution of C) or after C is done.

- Commands executed via SPI inside a function called by an SQL command (either an ordinary function or a trigger) follow one or the other of the above rules depending on the read/write flag passed to SPI. Commands executed in read-only mode follow the first rule: they cannot see changes of the calling command. Commands executed in read-write mode follow the second rule: they can see all changes made so far.

- All standard procedural languages set the SPI read-write mode depending on the volatility attribute of the function. Commands of `STABLE` and `IMMUTABLE` functions are done in read-only mode, while commands of `VOLATILE` functions are done in read-write mode. While authors of C functions are able to violate this convention, it’s unlikely to be a good idea to do so.

The next section contains an example that illustrates the application of these rules.

### 42.5. Examples

This section contains a very simple example of SPI usage. The procedure `execq` takes an SQL command as its first argument and a row count as its second, executes the command using `SPI_exec` and returns the number of rows that were processed by the command. You can find more complex examples for SPI in the source tree in `src/test/regress/regress.c` and in `contrib/spi`.

```c
#include "executor/spi.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

int execq(text *sql, int cnt);

int execq(text *sql, int cnt)
{
    char *command;
    int ret;
    int proc;

    /* Convert given text object to a C string */
    command = DatumGetCString(DirectFunctionCall1(textout,
        PointerGetDatum(sql)));
```

843
Chapter 42. Server Programming Interface

SPI_connect();

ret = SPI_exec(command, cnt);

proc = SPI_processed;

/*
 * If some rows were fetched, print them via elog(INFO).
 */
if (ret > 0 && SPI_tuptable != NULL)
{
    TupleDesc tupdesc = SPI_tuptable->tupdesc;
    SPITupleTable *tuptable = SPI_tuptable;
    char buf[8192];
    int i, j;

    for (j = 0; j < proc; j++)
    {
        HeapTuple tuple = tuptable->vals[j];
        for (i = 1, buf[0] = 0; i < tupdesc->natts; i++)
            snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                SPI_getvalue(tuple, tupdesc, i),
                (i == tupdesc->natts) ? " " : " |");
        elog(INFO, "EXECQ: %s", buf);
    }
}

SPI_finish();
pfree(command);

return (proc);

(This function uses call convention version 0, to make the example easier to understand. In real applications you should use the new version 1 interface.)

This is how you declare the function after having compiled it into a shared library (details are in Section 34.9.6):

CREATE FUNCTION execq(text, integer) RETURNS integer
AS 'filename'
    LANGUAGE C;

Here is a sample session:

=> SELECT execq('CREATE TABLE a (x integer)', 0);
execq
-------
0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1

=> SELECT execq('SELECT * FROM a', 0);
INFO: EXECQ: 0 -- inserted by execq
INFO: EXECQ: 1 -- returned by execq and inserted by upper INSERT
execq
-------
2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
execq
-------
1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO: EXECQ: 0
INFO: EXECQ: 1
INFO: EXECQ: 2 -- 0 + 2, only one row inserted - as specified
execq
-------
3 -- 10 is the max value only, 3 is the real number of rows
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
x
---
1 -- no rows in a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 2
INSERT 0 2
=> SELECT * FROM a;
x
---
1
2 -- there was one row in a + 1
(2 rows)

-- This demonstrates the data changes visibility rule:

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 2
INFO: EXECQ: 1
INSERT 0 2
=> SELECT * FROM a;
x
---
1
2
2 -- 2 rows * 1 (x in first row)
6 rows
(4 rows) -- 3 rows (2 + 1 just inserted) * 2 (x in second row)

^~~~~~~

rows visible to execq() in different invocations
VI. Reference

The entries in this Reference are meant to provide in reasonable length an authoritative, complete, and formal summary about their respective subjects. More information about the use of PostgreSQL, in narrative, tutorial, or example form, can be found in other parts of this book. See the cross-references listed on each reference page.

The reference entries are also available as traditional “man” pages.
I. SQL Commands

This part contains reference information for the SQL commands supported by PostgreSQL. By “SQL” the language in general is meant; information about the standards conformance and compatibility of each command can be found on the respective reference page.
**ABORT**

**Name**

ABORT — abort the current transaction

**Synopsis**

ABORT [ WORK | TRANSACTION ]

**Description**

ABORT rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the standard SQL command ROLLBACK, and is present only for historical reasons.

**Parameters**

WORK
TRANSACTION

Optional key words. They have no effect.

**Notes**

Use COMMIT to successfully terminate a transaction.

Issuing ABORT when not inside a transaction does no harm, but it will provoke a warning message.

**Examples**

To abort all changes:

ABORT;

**Compatibility**

This command is a PostgreSQL extension present for historical reasons. ROLLBACK is the equivalent standard SQL command.
See Also

BEGIN, COMMIT, ROLLBACK
ALTER AGGREGATE

Name

ALTER AGGREGATE — change the definition of an aggregate function

Synopsis

ALTER AGGREGATE name ( type [ , ... ] ) RENAME TO new_name
ALTER AGGREGATE name ( type [ , ... ] ) OWNER TO new_owner
ALTER AGGREGATE name ( type [ , ... ] ) SET SCHEMA new_schema

Description

ALTER AGGREGATE changes the definition of an aggregate function.

You must own the aggregate function to use ALTER AGGREGATE. To change the schema of an aggregate function, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the aggregate function’s schema. (These restrictions enforce that altering the owner doesn’t do anything you couldn’t do by dropping and recreating the aggregate function. However, a superuser can alter ownership of any aggregate function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing aggregate function.

type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of input data types.

new_name

The new name of the aggregate function.

new_owner

The new owner of the aggregate function.

new_schema

The new schema for the aggregate function.

Examples

To rename the aggregate function myavg for type integer to my_average:

ALTER AGGREGATE myavg(integer) RENAME TO my_average;
To change the owner of the aggregate function `myavg` for type `integer` to `joe`:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

To move the aggregate function `myavg` for type `integer` into schema `myschema`:

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

**Compatibility**

There is no `ALTER AGGREGATE` statement in the SQL standard.

**See Also**

`CREATE AGGREGATE, DROP AGGREGATE`
ALTER CONVERSION

Name
ALTERN CONVERSION — change the definition of a conversion

Synopsis
ALTER CONVERSION name RENAME TO newname
ALTER CONVERSION name OWNER TO newowner

Description
ALTER CONVERSION changes the definition of a conversion.
You must own the conversion to use ALTER CONVERSION. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the conversion’s schema. (These restrictions enforce that altering the owner doesn’t do anything you couldn’t do by dropping and recreating the conversion. However, a superuser can alter ownership of any conversion anyway.)

Parameters
name
The name (optionally schema-qualified) of an existing conversion.
newname
The new name of the conversion.
newowner
The new owner of the conversion.

Examples
To rename the conversion iso_8859_1_to_utf8 to latin1_to_unicode:
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;

To change the owner of the conversion iso_8859_1_to_utf8 to joe:
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
Compatibility

There is no ALTER CONVERSION statement in the SQL standard.

See Also

CREATE CONVERSION, DROP CONVERSION
ALTER DATABASE

Name
ALTER DATABASE — change a database

Synopsis
ALTER DATABASE name [ [ WITH ] option [ ... ] ]

where option can be:

  CONNECTION LIMIT conlimit

ALTER DATABASE name RENAME TO newname
ALTER DATABASE name OWNER TO new_owner
ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name SET configuration_parameter FROM CURRENT
ALTER DATABASE name RESET configuration_parameter
ALTER DATABASE name RESET ALL

Description
ALTER DATABASE changes the attributes of a database.
The first form changes certain per-database settings. (See below for details.) Only the database owner or a superuser can change these settings.
The second form changes the name of the database. Only the database owner or a superuser can rename a database; non-superuser owners must also have the CREATEDB privilege. The current database cannot be renamed. (Connect to a different database if you need to do that.)
The third form changes the owner of the database. To alter the owner, you must own the database and also be a direct or indirect member of the new owning role, and you must have the CREATEDB privilege. (Note that superusers have all these privileges automatically.)
The remaining forms change the session default for a run-time configuration variable for a PostgreSQL database. Whenever a new session is subsequently started in that database, the specified value becomes the session default value. The database-specific default overrides whatever setting is present in postgresql.conf or has been received from the postgres command line. Only the database owner or a superuser can change the session defaults for a database. Certain variables cannot be set this way, or can only be set by a superuser.

Parameters

name
The name of the database whose attributes are to be altered.
ALTER DATABASE

connlimit
How many concurrent connections can be made to this database. -1 means no limit.

newname
The new name of the database.

new_owner
The new owner of the database.

configuration_parameter value
Set this database’s session default for the specified configuration parameter to the given value. If value is DEFAULT or, equivalently, RESET is used, the database-specific setting is removed, so the system-wide default setting will be inherited in new sessions. Use RESET ALL to clear all database-specific settings. SET FROM CURRENT saves the session’s current value of the parameter as the database-specific value.

See SET and Chapter 18 for more information about allowed parameter names and values.

Notes
It is also possible to tie a session default to a specific role rather than to a database; see ALTER ROLE. Role-specific settings override database-specific ones if there is a conflict.

Examples
To disable index scans by default in the database test:

ALTER DATABASE test SET enable_indexscan TO off;

 Compatibility
The ALTER DATABASE statement is a PostgreSQL extension.

 See Also
CREATE DATABASE, DROP DATABASE, SET
ALTER DOMAIN

Name
ALTER DOMAIN — change the definition of a domain

Synopsis
ALTER DOMAIN name
  { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
  { SET | DROP } NOT NULL
ALTER DOMAIN name
  ADD domain_constraint
ALTER DOMAIN name
  DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
  OWNER TO new_owner
ALTER DOMAIN name
  SET SCHEMA new_schema

Description
ALTER DOMAIN changes the definition of an existing domain. There are several sub-forms:

SET/DROP DEFAULT
These forms set or remove the default value for a domain. Note that defaults only apply to subsequent INSERT commands; they do not affect rows already in a table using the domain.

SET/DROP NOT NULL
These forms change whether a domain is marked to allow NULL values or to reject NULL values. You can only SET NOT NULL when the columns using the domain contain no null values.

ADD domain_constraint
This form adds a new constraint to a domain using the same syntax as CREATE DOMAIN. This will only succeed if all columns using the domain satisfy the new constraint.

DROP CONSTRAINT
This form drops constraints on a domain.

OWNER
This form changes the owner of the domain to the specified user.

SET SCHEMA
This form changes the schema of the domain. Any constraints associated with the domain are moved into the new schema as well.

You must own the domain to use ALTER DOMAIN. To change the schema of a domain, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the domain’s schema.
ALTER DOMAIN

(These restrictions enforce that altering the owner doesn’t do anything you couldn’t do by dropping and recreating the domain. However, a superuser can alter ownership of any domain anyway.)

Parameters

name
The name (possibly schema-qualified) of an existing domain to alter.
domain_constraint
New domain constraint for the domain.
constraint_name
Name of an existing constraint to drop.
CASCADE
Automatically drop objects that depend on the constraint.
RESTRICT
Refuse to drop the constraint if there are any dependent objects. This is the default behavior.
new_owner
The user name of the new owner of the domain.
new_schema
The new schema for the domain.

Notes

Currently, ALTER DOMAIN ADD CONSTRAINT and ALTER DOMAIN SET NOT NULL will fail if the named domain or any derived domain is used within a composite-type column of any table in the database. They should eventually be improved to be able to verify the new constraint for such nested columns.

Examples

To add a NOT NULL constraint to a domain:

ALTER DOMAIN zipcode SET NOT NULL;

To remove a NOT NULL constraint from a domain:

ALTER DOMAIN zipcode DROP NOT NULL;

To add a check constraint to a domain:
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);

To remove a check constraint from a domain:
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;

To move the domain into a different schema:
ALTER DOMAIN zipcode SET SCHEMA customers;

Compatibility
ALTER DOMAIN conforms to the SQL standard, except for the OWNER and SET SCHEMA variants, which are PostgreSQL extensions.

See Also
CREATE DOMAIN, DROP DOMAIN
ALTER FUNCTION

Name

ALTER FUNCTION — change the definition of a function

Synopsis

ALTER FUNCTION name { [ [ argmode ] [ argname ] argtype [, ... ] ] }
action [ ... ] [ RESTRICT ]
ALTER FUNCTION name { [ [ argmode ] [ argname ] argtype [, ... ] ] }
RENAME TO new_name
ALTER FUNCTION name { [ [ argmode ] [ argname ] argtype [, ... ] ] }
OWNER TO new_owner
ALTER FUNCTION name { [ [ argmode ] [ argname ] argtype [, ... ] ] }
SET SCHEMA new_schema

where action is one of:

CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
IMMUTABLE | STABLE | VOLATILE
[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
COST execution_cost
ROWS result_rows
SET configuration_parameter { TO | = } { value | DEFAULT }
SET configuration_parameter FROM CURRENT
RESET configuration_parameter
RESET ALL

Description

ALTER FUNCTION changes the definition of a function.

You must own the function to use ALTER FUNCTION. To change a function’s schema, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the function’s schema. (These restrictions enforce that altering the owner doesn’t do anything you couldn’t do by dropping and recreating the function. However, a superuser can alter ownership of any function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: either IN, OUT, or INOUT. If omitted, the default is IN. Note that ALTER FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function’s identity. So it is sufficient to list the IN and INOUT arguments.
ALTER FUNCTION

argname

The name of an argument. Note that ALTER FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function’s identity.

argtype

The data type(s) of the function’s arguments (optionally schema-qualified), if any.

new_name

The new name of the function.

new_owner

The new owner of the function. Note that if the function is marked SECURITY DEFINER, it will subsequently execute as the new owner.

new_schema

The new schema for the function.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT changes the function so that it will be invoked when some or all of its arguments are null. RETURNS NULL ON NULL INPUT or STRICT changes the function so that it is not invoked if any of its arguments are null; instead, a null result is assumed automatically. See CREATE FUNCTION for more information.

IMMUTABLE
STABLE
VOLATILE

Change the volatility of the function to the specified setting. See CREATE FUNCTION for details.

[ EXTERNAL ] SECURITY INVOKER
[ EXTERNAL ] SECURITY DEFINER

Change whether the function is a security definer or not. The key word EXTERNAL is ignored for SQL conformance. See CREATE FUNCTION for more information about this capability.

COST execution_cost

Change the estimated execution cost of the function. See CREATE FUNCTION for more information.

ROWS result_rows

Change the estimated number of rows returned by a set-returning function. See CREATE FUNCTION for more information.

configuration_parameter

Add or change the assignment to be made to a configuration parameter when the function is called. If value is DEFAULT or, equivalently, RESET is used, the function-local setting is removed, so that the function executes with the value present in its environment. Use RESET ALL to clear all function-local settings. SET FROM CURRENT saves the session’s current value of the parameter as the value to be applied when the function is entered.

See SET and Chapter 18 for more information about allowed parameter names and values.
ALTER FUNCTION

RESTRICT

Ignored for conformance with the SQL standard.

Examples

To rename the function sqrt for type integer to square_root:
ALTER FUNCTION sqrt(integer) RENAME TO square_root;

To change the owner of the function sqrt for type integer to joe:
ALTER FUNCTION sqrt(integer) OWNER TO joe;

To change the schema of the function sqrt for type integer to maths:
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;

To adjust the search path that is automatically set for a function:
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;

To disable automatic setting of search_path for a function:
ALTER FUNCTION check_password(text) RESET search_path;

The function will now execute with whatever search path is used by its caller.

Compatibility

This statement is partially compatible with the ALTER FUNCTION statement in the SQL standard. The standard allows more properties of a function to be modified, but does not provide the ability to rename a function, make a function a security definer, attach configuration parameter values to a function, or change the owner, schema, or volatility of a function. The standard also requires the RESTRICT key word, which is optional in PostgreSQL.

See Also

CREATE FUNCTION, DROP FUNCTION
ALTER GROUP

Name
ALTER GROUP — change role name or membership

Synopsis
ALTER GROUP groupname ADD USER username [, ... ]
ALTER GROUP groupname DROP USER username [, ... ]
ALTER GROUP groupname RENAME TO newname

Description
ALTER GROUP changes the attributes of a user group. This is an obsolete command, though still accepted for backwards compatibility, because groups (and users too) have been superseded by the more general concept of roles.

The first two variants add users to a group or remove them from a group. (Any role can play the part of either a “user” or a “group” for this purpose.) These variants are effectively equivalent to granting or revoking membership in the role named as the “group”; so the preferred way to do this is to use GRANT or REVOKE.

The third variant changes the name of the group. This is exactly equivalent to renaming the role with ALTER ROLE.

Parameters

grouppname

The name of the group (role) to modify.

username

Users (roles) that are to be added to or removed from the group. The users must already exist; ALTER GROUP does not create or drop users.

newname

The new name of the group.

Examples
Add users to a group:

ALTER GROUP staff ADD USER karl, john;

Remove a user from a group:

ALTER GROUP workers DROP USER beth;
**Compatibility**

There is no `ALTER GROUP` statement in the SQL standard.

**See Also**

`GRANT`, `REVOKE`, `ALTER ROLE`
**ALTER INDEX**

**Name**

ALTER INDEX — change the definition of an index

**Synopsis**

ALTER INDEX name RENAME TO new_name  
ALTER INDEX name SET TABLESPACE tablespace_name  
ALTER INDEX name SET { storage_parameter = value [, ... ] }  
ALTER INDEX name RESET { storage_parameter [, ... ] }

**Description**

ALTER INDEX changes the definition of an existing index. There are several subforms:

**RENAME**

The RENAME form changes the name of the index. There is no effect on the stored data.

**SET TABLESPACE**

This form changes the index’s tablespace to the specified tablespace and moves the data file(s) associated with the index to the new tablespace. See also CREATE TABLESPACE.

**SET ( storage_parameter = value [, ... ] )**

This form changes one or more index-method-specific storage parameters for the index. See CREATE INDEX for details on the available parameters. Note that the index contents will not be modified immediately by this command; depending on the parameter you might need to rebuild the index with REINDEX to get the desired effects.

**RESET ( storage_parameter [, ... ] )**

This form resets one or more index-method-specific storage parameters to their defaults. As with SET, a REINDEX might be needed to update the index entirely.

**Parameters**

**name**

The name (possibly schema-qualified) of an existing index to alter.

**new_name**

The new name for the index.

**tablespace_name**

The tablespace to which the index will be moved.

**storage_parameter**

The name of an index-method-specific storage parameter.
The new value for an index-method-specific storage parameter. This might be a number or a word depending on the parameter.

**Notes**

These operations are also possible using `ALTER TABLE`. `ALTER INDEX` is in fact just an alias for the forms of `ALTER TABLE` that apply to indexes.

There was formerly an `ALTER INDEX OWNER` variant, but this is now ignored (with a warning). An index cannot have an owner different from its table’s owner. Changing the table’s owner automatically changes the index as well.

Changing any part of a system catalog index is not permitted.

**Examples**

To rename an existing index:

```
ALTER INDEX distributors RENAME TO suppliers;
```

To move an index to a different tablespace:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

To change an index’s fill factor (assuming that the index method supports it):

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

**Compatibility**

`ALTER INDEX` is a PostgreSQL extension.

**See Also**

`CREATE INDEX, REINDEX`
**ALTER LANGUAGE**

**Name**

ALTER LANGUAGE — change the definition of a procedural language

**Synopsis**

ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO newname
ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO new_owner

**Description**

ALTER LANGUAGE changes the definition of a procedural language. The only functionality is to rename the language or assign a new owner. You must be superuser or owner of the language to use ALTER LANGUAGE.

**Parameters**

name

Name of a language

newname

The new name of the language

new_owner

The new owner of the language

**Compatibility**

There is no ALTER LANGUAGE statement in the SQL standard.

**See Also**

CREATE LANGUAGE, DROP LANGUAGE
ALTER OPERATOR

Name
ALTER OPERATOR — change the definition of an operator

Synopsis
ALTER OPERATOR name ( { lefttype | NONE }, { righttype | NONE } ) OWNER TO newowner

Description
ALTER OPERATOR changes the definition of an operator. The only currently available functionality is to change the owner of the operator.

You must own the operator to use ALTER OPERATOR. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator’s schema. (These restrictions enforce that altering the owner doesn’t do anything you couldn’t do by dropping and recreating the operator. However, a superuser can alter ownership of any operator anyway.)

Parameters

name
The name (optionally schema-qualified) of an existing operator.

lefttype
The data type of the operator’s left operand; write NONE if the operator has no left operand.

righttype
The data type of the operator’s right operand; write NONE if the operator has no right operand.

newowner
The new owner of the operator.

Examples
Change the owner of a custom operator a @@ b for type text:

ALTER OPERATOR @@ (text, text) OWNER TO joe;

Compatibility
There is no ALTER OPERATOR statement in the SQL standard.
See Also

CREATE OPERATOR, DROP OPERATOR
ALTER OPERATOR CLASS

Name
ALTER OPERATOR CLASS — change the definition of an operator class

Synopsis
ALTER OPERATOR CLASS name USING index_method RENAME TO newname
ALTER OPERATOR CLASS name USING index_method OWNER TO newowner

Description
ALTER OPERATOR CLASS changes the definition of an operator class.
You must own the operator class to use ALTER OPERATOR CLASS. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator class’s schema. (These restrictions enforce that altering the owner doesn’t do anything you couldn’t do by dropping and recreating the operator class. However, a superuser can alter ownership of any operator class anyway.)

Parameters
name
    The name (optionally schema-qualified) of an existing operator class.
index_method
    The name of the index method this operator class is for.
newname
    The new name of the operator class.
newowner
    The new owner of the operator class.

Compatibility
There is no ALTER OPERATOR CLASS statement in the SQL standard.

See Also
CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY
ALTER OPERATOR FAMILY

Name

ALTER OPERATOR FAMILY — change the definition of an operator family

Synopsis

ALTER OPERATOR FAMILY name USING index_method ADD
{ OPERATOR strategy_number operator_name (op_type, op_type) [ RECHECK ]
 | FUNCTION support_number [ (op_type [, op_type] ) ] funcname (argument_type [, ...] )
 } [, ... ]
ALTER OPERATOR FAMILY name USING index_method DROP
{ OPERATOR strategy_number (op_type [, op_type] )
 | FUNCTION support_number (op_type [, op_type] )
 } [, ... ]
ALTER OPERATOR FAMILY name USING index_method RENAME TO newname
ALTER OPERATOR FAMILY name USING index_method OWNER TO newowner

Description

ALTER OPERATOR FAMILY changes the definition of an operator family. You can add operators and support functions to the family, remove them from the family, or change the family’s name or owner.

When operators and support functions are added to a family with ALTER OPERATOR FAMILY, they are not part of any specific operator class within the family, but are just “loose” within the family. This indicates that these operators and functions are compatible with the family’s semantics, but are not required for correct functioning of any specific index. (Operators and functions that are so required should be declared as part of an operator class, instead; see CREATE OPERATOR CLASS.) PostgreSQL will allow loose members of a family to be dropped from the family at any time, but members of an operator class cannot be dropped without dropping the whole class and any indexes that depend on it. Typically, single-data-type operators and functions are part of operator classes because they are needed to support an index on that specific data type, while cross-data-type operators and functions are made loose members of the family.

You must be a superuser to use ALTER OPERATOR FAMILY. (This restriction is made because an erroneous operator family definition could confuse or even crash the server.)

ALTER OPERATOR FAMILY does not presently check whether the operator family definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user’s responsibility to define a valid operator family.

Refer to Section 34.14 for further information.

Parameters

name

The name (optionally schema-qualified) of an existing operator family.
index_method

The name of the index method this operator family is for.

strategy_number

The index method’s strategy number for an operator associated with the operator family.

operator_name

The name (optionally schema-qualified) of an operator associated with the operator family.

op_type

In an OPERATOR clause, the operand data type(s) of the operator, or NONE to signify a left- or right-unary operator. Unlike the comparable syntax in CREATE OPERATOR CLASS, the operand data types must always be specified.

In an ADD FUNCTION clause, the operand data type(s) the function is intended to support, if different from the input data type(s) of the function. For B-tree and hash indexes it is not necessary to specify op_type since the function’s input data type(s) are always the correct ones to use. For GIN and GiST indexes it is necessary to specify the input data type the function is to be used with.

In a DROP FUNCTION clause, the operand data type(s) the function is intended to support must be specified.

RECHECK

If present, the index is “lossy” for this operator, and so the rows retrieved using the index must be rechecked to verify that they actually satisfy the qualification clause involving this operator.

support_number

The index method’s support procedure number for a function associated with the operator family.

funcname

The name (optionally schema-qualified) of a function that is an index method support procedure for the operator family.

argument_types

The parameter data type(s) of the function.

newname

The new name of the operator family.

newowner

The new owner of the operator family.

The OPERATOR and FUNCTION clauses can appear in any order.

Notes

Notice that the DROP syntax only specifies the “slot” in the operator family, by strategy or support number and input data type(s). The name of the operator or function occupying the slot is not mentioned. Also, for DROP FUNCTION the type(s) to specify are the input data type(s) the function is intended to support; for GIN and GiST indexes this might have nothing to do with the actual input argument types of the function.

Because the index machinery does not check access permissions on functions before using them, including a function or operator in an operator family is tantamount to granting public execute per-
ALTER OPERATOR FAMILY is not an issue for the sorts of functions that are useful in an operator family.

The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

**Examples**

The following example command adds cross-data-type operators and support functions to an operator family that already contains B-tree operator classes for data types int4 and int2.

```sql
ALTER OPERATOR FAMILY integer_ops USING btree ADD

-- int4 vs int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 > (int4, int2) ,
OPERATOR 5 >= (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 > (int2, int4) ,
OPERATOR 5 >= (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;
```

To remove these entries again:

```sql
ALTER OPERATOR FAMILY integer_ops USING btree DROP

-- int4 vs int2
OPERATOR 1 (int4, int2) ,
OPERATOR 2 (int4, int2) ,
OPERATOR 3 (int4, int2) ,
OPERATOR 4 (int4, int2) ,
OPERATOR 5 (int4, int2) ,
FUNCTION 1 (int4, int2) ,

-- int2 vs int4
OPERATOR 1 (int2, int4) ,
OPERATOR 2 (int2, int4) ,
OPERATOR 3 (int2, int4) ,
OPERATOR 4 (int2, int4) ,
OPERATOR 5 (int2, int4) ,
FUNCTION 1 (int2, int4) ;
```

**Compatibility**

There is no ALTER OPERATOR FAMILY statement in the SQL standard.
ALTER OPERATOR FAMILY

See Also

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS
ALTER ROLE

Name
ALTER ROLE — change a database role

Synopsis
ALTER ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

- SUPERUSER | NOSUPERUSER
- CREATEDB | NOCREATEDB
- CREATEROLE | NOCREATEROLE
- CREATEUSER | NOCREATEUSER
- INHERIT | NOINHERIT
- LOGIN | NOLOGIN
- CONNECTION LIMIT conlimit
- [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
- VALID UNTIL 'timestamp'

ALTER ROLE name RENAME TO newname

ALTER ROLE name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE name SET configuration_parameter FROM CURRENT
ALTER ROLE name RESET configuration_parameter
ALTER ROLE name RESET ALL

Description

ALTER ROLE changes the attributes of a PostgreSQL role.

The first variant of this command listed in the synopsis can change many of the role attributes that can be specified in CREATE ROLE. (All the possible attributes are covered, except that there are no options for adding or removing memberships; use GRANT and REVOKE for that.) Attributes not mentioned in the command retain their previous settings. Database superusers can change any of these settings for any role. Roles having CREATEROLE privilege can change any of these settings, but only for non-superuser roles. Ordinary roles can only change their own password.

The second variant changes the name of the role. Database superusers can rename any role. Roles having CREATEROLE privilege can rename non-superuser roles. The current session user cannot be renamed. (Connect as a different user if you need to do that.) Because MD5-encrypted passwords use the role name as cryptographic salt, renaming a role clears its password if the password is MD5-encrypted.

The remaining variants change a role’s session default for a specified configuration variable. Whenever the role subsequently starts a new session, the specified value becomes the session default, overriding whatever setting is present in postgresql.conf or has been received from the postgres command line. (For a role without LOGIN privilege, session defaults have no effect.) Ordinary roles can change their own session defaults. Superusers can change anyone’s session defaults. Roles having CREATEROLE privilege can change defaults for non-superuser roles. Certain variables cannot be set this way, or can only be set if a superuser issues the command.
ALTER ROLE

Parameters

name

The name of the role whose attributes are to be altered.

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
CREATEUSER
NOCREATEUSER
INHERIT
NOINHERIT
LOGIN
NOLOGIN
CONNECTION LIMIT connlimit
PASSWORD password
ENCRYPTED
UNENCRIPTED
VALID UNTIL 'timestamp'

These clauses alter attributes originally set by CREATE ROLE. For more information, see the CREATE ROLE reference page.

newname

The new name of the role.

configuration_parameter value

Set this role’s session default for the specified configuration parameter to the given value. If value is DEFAULT or, equivalently, RESET is used, the role-specific variable setting is removed, so the role will inherit the system-wide default setting in new sessions. Use RESET ALL to clear all role-specific settings. SET FROM CURRENT saves the session’s current value of the parameter as the role-specific value.

See SET and Chapter 18 for more information about allowed parameter names and values.

Notes

Use CREATE ROLE to add new roles, and DROP ROLE to remove a role.

ALTER ROLE cannot change a role’s memberships. Use GRANT and REVOKE to do that.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in cleartext, and it might also be logged in the client’s command history or the server log. psql contains a command \password that can be used to safely change a role’s password.

It is also possible to tie a session default to a specific database rather than to a role; see ALTER DATABASE. Role-specific settings override database-specific ones if there is a conflict.
Examples

Change a role’s password:

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

Change a password expiration date, specifying that the password should expire at midday on 4th May 2015 using the time zone which is one hour ahead of UTC:

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Make a password valid forever:

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

Give a role the ability to create other roles and new databases:

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Give a role a non-default setting of the maintenance_work_mem parameter:

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

Compatibility

The `ALTER ROLE` statement is a PostgreSQL extension.

See Also

`CREATE ROLE`, `DROP ROLE`, `SET`
ALTER SCHEMA

Name
ALTER SCHEMA — change the definition of a schema

Synopsis
ALTER SCHEMA name RENAME TO newname
ALTER SCHEMA name OWNER TO newowner

Description
ALTER SCHEMA changes the definition of a schema.

You must own the schema to use ALTER SCHEMA. To rename a schema you must also have the CREATE privilege for the database. To alter the owner, you must also be a direct or indirect member of the new owning role, and you must have the CREATE privilege for the database. (Note that superusers have all these privileges automatically.)

Parameters

name
The name of an existing schema.

newname
The new name of the schema. The new name cannot begin with pg_, as such names are reserved for system schemas.

newowner
The new owner of the schema.

Compatibility
There is no ALTER SCHEMA statement in the SQL standard.

See Also
CREATE SCHEMA, DROP SCHEMA
ALTER SEQUENCE

Name
ALTER SEQUENCE — change the definition of a sequence generator

Synopsis
ALTER SEQUENCE name [ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
[ RESTART [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
[ OWNED BY { table.column | NONE } ]
ALTER SEQUENCE name RENAME TO new_name
ALTER SEQUENCE name SET SCHEMA new_schema

Description
ALTER SEQUENCE changes the parameters of an existing sequence generator. Any parameters not specifically set in the ALTER SEQUENCE command retain their prior settings.

You must own the sequence to use ALTER SEQUENCE. To change a sequence’s schema, you must also have CREATE privilege on the new schema.

Parameters

name
The name (optionally schema-qualified) of a sequence to be altered.

increment
The clause INCREMENT BY increment is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue
NO MINVALUE
The optional clause MINVALUE minvalue determines the minimum value a sequence can generate. If NO MINVALUE is specified, the defaults of 1 and -2\(^{31}\)-1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current minimum value will be maintained.

maxvalue
NO MAXVALUE
The optional clause MAXVALUE maxvalue determines the maximum value for the sequence. If NO MAXVALUE is specified, the defaults are 2\(^{31}\)-1 and -1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current maximum value will be maintained.

880
ALTER SEQUENCE

start

The optional clause RESTART WITH start changes the current value of the sequence. This is equivalent to calling the setval function with \texttt{is\_called} = false: the specified value will be returned by the next call of \texttt{nextval}.

cache

The clause \textit{CACHE} cache enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache). If unspecified, the old cache value will be maintained.

CYCLE

The optional \textit{CYCLE} key word can be used to enable the sequence to wrap around when the \texttt{maxvalue} or \texttt{minvalue} has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the \texttt{minvalue} or \texttt{maxvalue}, respectively.

\texttt{NO CYCLE}

If the optional \texttt{NO CYCLE} key word is specified, any calls to \texttt{nextval} after the sequence has reached its maximum value will return an error. If neither \texttt{CYCLE} or \texttt{NO CYCLE} are specified, the old cycle behavior will be maintained.

OWNED BY table.column

OWNED BY NONE

The \texttt{OWNED BY} option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. If specified, this association replaces any previously specified association for the sequence. The specified table must have the same owner and be in the same schema as the sequence. Specifying \texttt{OWNED BY NONE} removes any existing association, making the sequence “free-standing”.

new_name

The new name for the sequence.

new_schema

The new schema for the sequence.

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, ALTER SEQUENCE’s effects on the sequence generation parameters are never rolled back; those changes take effect immediately and are not reversible. However, the \texttt{OWNED BY}, \texttt{RENAME}, and \texttt{SET SCHEMA} clauses cause ordinary catalog updates that can be rolled back.

ALTER SEQUENCE will not immediately affect \texttt{nextval} results in backends, other than the current one, that have preallocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence generation parameters. The current backend will be affected immediately.

ALTER SEQUENCE does not affect the \texttt{currval} status for the sequence. (Before PostgreSQL 8.3, it sometimes did.)

Some variants of ALTER TABLE can be used with sequences as well; for example, to rename a sequence it is also possible to use ALTER TABLE RENAME.
Examples

Restart a sequence called `serial`, at 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Compatibility

`ALTER SEQUENCE` conforms to the SQL standard, except for the `OWNED BY`, `RENAME`, and `SET SCHEMA` clauses, which are PostgreSQL extensions.

See Also

`CREATE SEQUENCE`, `DROP SEQUENCE`
ALTER TABLE

Name

ALTER TABLE — change the definition of a table

Synopsis

ALTER TABLE [ ONLY ] name [ * ]

  action [, ... ]

ALTER TABLE [ ONLY ] name [ * ]

  RENAME [ COLUMN ] column TO new_column

ALTER TABLE name

  RENAME TO new_name

ALTER TABLE name

  SET SCHEMA new_schema

where action is one of:

  ADD [ COLUMN ] column type [ column_constraint [ ... ] ]

  DROP [ COLUMN ] column [ RESTRICT | CASCADE ]

  ALTER [ COLUMN ] column TYPE type [ USING expression ]

  ALTER [ COLUMN ] column SET DEFAULT expression

  ALTER [ COLUMN ] column DROP DEFAULT

  ALTER [ COLUMN ] column ( SET | DROP ) NOT NULL

  ALTER [ COLUMN ] column SET STATISTICS integer

  ALTER [ COLUMN ] column SET STORAGE ( PLAIN | EXTERNAL | EXTENDED | MAIN )

  ADD table_constraint

  DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]

  DISABLE TRIGGER [ trigger_name | ALL | USER ]

  ENABLE TRIGGER [ trigger_name | ALL | USER ]

  ENABLE REPLICA TRIGGER trigger_name

  ENABLE ALWAYS TRIGGER trigger_name

  DISABLE RULE rewrite_rule_name

  ENABLE RULE rewrite_rule_name

  ENABLE REPLICA RULE rewrite_rule_name

  ENABLE ALWAYS RULE rewrite_rule_name

  CLUSTER ON index_name

  SET WITHOUT CLUSTER

  SET WITHOUT OIDS

  SET ( storage_parameter = value [, ... ] )

  RESET ( storage_parameter [, ... ] )

  INHERIT parent_table

  NO INHERIT parent_table

  OWNER TO new_owner

  SET TABLESPACE new_tablespace
ALTER TABLE

Description

ALTER TABLE changes the definition of an existing table. There are several subforms:

ADD COLUMN

This form adds a new column to the table, using the same syntax as CREATE TABLE.

DROP COLUMN

This form drops a column from a table. Indexes and table constraints involving the column will be automatically dropped as well. You will need to say CASCADE if anything outside the table depends on the column, for example, foreign key references or views.

ALTER COLUMN TYPE

This form changes the type of a column of a table. Indexes and simple table constraints involving the column will be automatically converted to use the new column type by reparsing the originally supplied expression. The optional USING clause specifies how to compute the new column value from the old; if omitted, the default conversion is the same as an assignment cast from old data type to new. A USING clause must be provided if there is no implicit or assignment cast from old to new type.

SET/DROP DEFAULT

These forms set or remove the default value for a column. The default values only apply to subsequent INSERT commands; they do not cause rows already in the table to change. Defaults can also be created for views, in which case they are inserted into INSERT statements on the view before the view’s ON INSERT rule is applied.

SET/DROP NOT NULL

These forms change whether a column is marked to allow null values or to reject null values. You can only use SET NOT NULL when the column contains no null values.

SET STATISTICS

This form sets the per-column statistics-gathering target for subsequent ANALYZE operations. The target can be set in the range 0 to 1000; alternatively, set it to -1 to revert to using the system default statistics target (default_statistics_target). For more information on the use of statistics by the PostgreSQL query planner, refer to Section 14.2.

SET STORAGE

This form sets the storage mode for a column. This controls whether this column is held inline or in a supplementary table, and whether the data should be compressed or not. PLAIN must be used for fixed-length values such as integer and is inline, uncompressed. MAIN is for inline, compressible data. EXTERNAL is for external, uncompressed data, and EXTENDED is for external, compressed data. EXTENDED is the default for most data types that support non-PLAIN storage. Use of EXTERNAL will make substring operations on text and bytea columns faster, at the penalty of increased storage space. Note that SET STORAGE doesn’t itself change anything in the table, it just sets the strategy to be pursued during future table updates. See Section 53.2 for more information.

ADD table_constraint

This form adds a new constraint to a table using the same syntax as CREATE TABLE.

DROP CONSTRAINT

This form drops the specified constraint on a table.
ALTER TABLE

DISABLE ENABLE [ REPLICA | ALWAYS ] TRIGGER

These forms configure the firing of trigger(s) belonging to the table. A disabled trigger is still known to the system, but is not executed when its triggering event occurs. For a deferred trigger, the enable status is checked when the event occurs, not when the trigger function is actually executed. One can disable or enable a single trigger specified by name, or all triggers on the table, or only user triggers (this option excludes triggers that are used to implement foreign key constraints). Disabling or enabling constraint triggers requires superuser privileges; it should be done with caution since of course the integrity of the constraint cannot be guaranteed if the triggers are not executed. The trigger firing mechanism is also affected by the configuration variable session_replication_role. Simply enabled triggers will fire when the replication role is “origin” (the default) or “local”. Triggers configured ENABLE REPLICA will only fire if the session is in “replica” mode and triggers configured ENABLE ALWAYS will fire regardless of the current replication mode.

DISABLE ENABLE [ REPLICA | ALWAYS ] RULE

These forms configure the firing of rewrite rules belonging to the table. A disabled rule is still known to the system, but is not applied during query rewriting. The semantics are as for disabled/enabled triggers. This configuration is ignored for ON SELECT rules, which are always applied in order to keep views working even if the current session is in a non-default replication role.

CLUSTER

This form selects the default index for future CLUSTER operations. It does not actually re-cluster the table.

SET WITHOUT CLUSTER

This form removes the most recently used CLUSTER index specification from the table. This affects future cluster operations that don’t specify an index.

SET WITHOUT OIDS

This form removes the oid system column from the table. This is exactly equivalent to DROP COLUMN oid RESTRICT, except that it will not complain if there is already no oid column.

Note that there is no variant of ALTER TABLE that allows OIDs to be restored to a table once they have been removed.

SET ( storage_parameter = value [, ... ] )

This form changes one or more storage parameters for the table. See CREATE TABLE for details on the available parameters. Note that the table contents will not be modified immediately by this command; depending on the parameter you might need to rewrite the table to get the desired effects. That can be done with CLUSTER or one of the forms of ALTER TABLE that forces a table rewrite.

Note: While CREATE TABLE allows OIDS to be specified in the WITH (storage_parameter) syntax, ALTER TABLE does not treat OIDS as a storage parameter.

RESET ( storage_parameter [, ... ] )

This form resets one or more storage parameters to their defaults. As with SET, a table rewrite might be needed to update the table entirely.
**ALTER TABLE**

**INHERIT parent_table**

This form adds the target table as a new child of the specified parent table. Subsequently, queries against the parent will include records of the target table. To be added as a child, the target table must already contain all the same columns as the parent (it could have additional columns, too). The columns must have matching data types, and if they have `NOT NULL` constraints in the parent then they must also have `NOT NULL` constraints in the child.

There must also be matching child-table constraints for all `CHECK` constraints of the parent. Currently `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints are not considered, but this might change in the future.

**NO INHERIT parent_table**

This form removes the target table from the list of children of the specified parent table. Queries against the parent table will no longer include records drawn from the target table.

**OWNER**

This form changes the owner of the table, sequence, or view to the specified user.

**SET TABLESPACE**

This form changes the table’s tablespace to the specified tablespace and moves the data file(s) associated with the table to the new tablespace. Indexes on the table, if any, are not moved; but they can be moved separately with additional `SET TABLESPACE` commands. See also `CREATE TABLESPACE`.

**RENAME**

The `RENAME` forms change the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data.

**SET SCHEMA**

This form moves the table into another schema. Associated indexes, constraints, and sequences owned by table columns are moved as well.

All the actions except `RENAME` and `SET SCHEMA` can be combined into a list of multiple alterations to apply in parallel. For example, it is possible to add several columns and/or alter the type of several columns in a single command. This is particularly useful with large tables, since only one pass over the table need be made.

You must own the table to use `ALTER TABLE`. To change the schema of a table, you must also have `CREATE` privilege on the new schema. To add the table as a new child of a parent table, you must own the parent table as well. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the table’s schema. (These restrictions enforce that altering the owner doesn’t do anything you couldn’t do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.)

**Parameters**

**name**

The name (optionally schema-qualified) of an existing table to alter. If `ONLY` is specified before the table name, only that table is altered. If `ONLY` is not specified, the table and all its descendant tables (if any) are altered. Optionally, `*` can be specified after the table name to explicitly indicate that descendant tables are included.
column
    Name of a new or existing column.
new_column
    New name for an existing column.
new_name
    New name for the table.
type
    Data type of the new column, or new data type for an existing column.
table_constraint
    New table constraint for the table.
constraint_name
    Name of an existing constraint to drop.
CASCADE
    Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column).
RESTRICT
    Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.
trigger_name
    Name of a single trigger to disable or enable.
ALL
    Disable or enable all triggers belonging to the table. (This requires superuser privilege if any of the triggers are for foreign key constraints.)
USER
    Disable or enable all triggers belonging to the table except for foreign key constraint triggers.
index_name
    The index name on which the table should be marked for clustering.
storage_parameter
    The name of a table storage parameter.
value
    The new value for a table storage parameter. This might be a number or a word depending on the parameter.
parent_table
    A parent table to associate or de-associate with this table.
new_owner
    The user name of the new owner of the table.
new_tablespace
    The name of the tablespace to which the table will be moved.
ALTER TABLE

new_schema

The name of the schema to which the table will be moved.

Notes

The key word COLUMN is noise and can be omitted.

When a column is added with ADD COLUMN, all existing rows in the table are initialized with the column’s default value (NULL if no DEFAULT clause is specified).

Adding a column with a non-null default or changing the type of an existing column will require the entire table to be rewritten. This might take a significant amount of time for a large table; and it will temporarily require double the disk space.

Adding a CHECK or NOT NULL constraint requires scanning the table to verify that existing rows meet the constraint.

The main reason for providing the option to specify multiple changes in a single ALTER TABLE is that multiple table scans or rewrites can thereby be combined into a single pass over the table.

The DROP COLUMN form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

The fact that ALTER TYPE requires rewriting the whole table is sometimes an advantage, because the rewriting process eliminates any dead space in the table. For example, to reclaim the space occupied by a dropped column immediately, the fastest way is:

```
ALTER TABLE table ALTER COLUMN anycol TYPE anytype;
```

where anycol is any remaining table column and anytype is the same type that column already has. This results in no semantically-visible change in the table, but the command forces rewriting, which gets rid of no-longer-useful data.

The USING option of ALTER TYPE can actually specify any expression involving the old values of the row; that is, it can refer to other columns as well as the one being converted. This allows very general conversions to be done with the ALTER TYPE syntax. Because of this flexibility, the USING expression is not applied to the column’s default value (if any); the result might not be a constant expression as required for a default. This means that when there is no implicit or assignment cast from old to new type, ALTER TYPE might fail to convert the default even though a USING clause is supplied. In such cases, drop the default with DROP DEFAULT, perform the ALTER TYPE, and then use SET DEFAULT to add a suitable new default. Similar considerations apply to indexes and constraints involving the column.

If a table has any descendant tables, it is not permitted to add, rename, or change the type of a column in the parent table without doing the same to the descendants. That is, ALTER TABLE ONLY will be rejected. This ensures that the descendants always have columns matching the parent.

A recursive DROP COLUMN operation will remove a descendant table’s column only if the descendant does not inherit that column from any other parents and never had an independent definition of the column. A nonrecursive DROP COLUMN (i.e., ALTER TABLE ONLY ... DROP COLUMN) never removes any descendant columns, but instead marks them as independently defined rather than inherited.
ALTER TABLE

The TRIGGER, CLUSTER, OWNER, and TABLESPACE actions never recurse to descendant tables; that is, they always act as though ONLY were specified. Adding a constraint can recurse only for CHECK constraints.

Changing any part of a system catalog table is not permitted.
Refer to CREATE TABLE for a further description of valid parameters. Chapter 5 has further information on inheritance.

Examples

To add a column of type varchar to a table:

ALTER TABLE distributors ADD COLUMN address varchar(30);

To drop a column from a table:

ALTER TABLE distributors DROP COLUMN address RESTRICT;

To change the types of two existing columns in one operation:

ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);

To change an integer column containing UNIX timestamps to timestamp with time zone via a USING clause:

ALTER TABLE foo
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';

The same, when the column has a default expression that won’t automatically cast to the new data type:

ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();

To rename an existing column:

ALTER TABLE distributors RENAME COLUMN address TO city;

To rename an existing table:
ALTER TABLE distributors RENAME TO suppliers;

To add a not-null constraint to a column:
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;

To remove a not-null constraint from a column:
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;

To add a check constraint to a table:
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);

To remove a check constraint from a table and all its children:
ALTER TABLE distributors DROP CONSTRAINT zipchk;

To add a foreign key constraint to a table:
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (address) MATCH FULL;

To add a (multicolumn) unique constraint to a table:
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);

To add an automatically named primary key constraint to a table, noting that a table can only ever have one primary key:
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);

To move a table to a different tablespace:
ALTER TABLE distributors SET TABLESPACE fasttablespace;

To move a table to a different schema:
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
Compatibility

The **ADD**, **DROP**, and **SET DEFAULT** forms conform with the SQL standard. The other forms are PostgreSQL extensions of the SQL standard. Also, the ability to specify more than one manipulation in a single **ALTER TABLE** command is an extension.

**ALTER TABLE DROP COLUMN** can be used to drop the only column of a table, leaving a zero-column table. This is an extension of SQL, which disallows zero-column tables.
ALTER TABLESPACE

Name
ALTER TABLESPACE — change the definition of a tablespace

Synopsis
ALTER TABLESPACE name RENAME TO newname
ALTER TABLESPACE name OWNER TO newowner

Description
ALTER TABLESPACE changes the definition of a tablespace.
You must own the tablespace to use ALTER TABLESPACE. To alter the owner, you must also be a direct or indirect member of the new owning role. (Note that superusers have these privileges automatically.)

Parameters

name
The name of an existing tablespace.

newname
The new name of the tablespace. The new name cannot begin with pg_, as such names are reserved for system tablespaces.

newowner
The new owner of the tablespace.

Examples
Rename tablespace index_space to fast_raid:

ALTER TABLESPACE index_space RENAME TO fast_raid;

Change the owner of tablespace index_space:

ALTER TABLESPACE index_space OWNER TO mary;

Compatibility
There is no ALTER TABLESPACE statement in the SQL standard.
See Also

CREATE TABLESPACE, DROP TABLESPACE
ALTER TEXT SEARCH CONFIGURATION

Name
ALTER TEXT SEARCH CONFIGURATION — change the definition of a text search configuration

Synopsis
ALTER TEXT SEARCH CONFIGURATION name
    ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO newname
ALTER TEXT SEARCH CONFIGURATION name OWNER TO newowner

Description
ALTER TEXT SEARCH CONFIGURATION changes the definition of a text search configuration. You can modify its mappings from token types to dictionaries, or change the configuration’s name or owner.

You must be the owner of the configuration to use ALTER TEXT SEARCH CONFIGURATION.

Parameters
name
The name (optionally schema-qualified) of an existing text search configuration.

token_type
The name of a token type that is emitted by the configuration’s parser.

dictionary_name
The name of a text search dictionary to be consulted for the specified token type(s). If multiple dictionaries are listed, they are consulted in the specified order.

old_dictionary
The name of a text search dictionary to be replaced in the mapping.

new_dictionary
The name of a text search dictionary to be substituted for old_dictionary.

newname
The new name of the text search configuration.
ALTER TEXT SEARCH CONFIGURATION

newowner

The new owner of the text search configuration.

The ADD MAPPING FOR form installs a list of dictionaries to be consulted for the specified token type(s); it is an error if there is already a mapping for any of the token types. The ALTER MAPPING FOR form does the same, but first removing any existing mapping for those token types. The ALTER MAPPING REPLACE forms substitute new_dictionary for old_dictionary anywhere the latter appears. This is done for only the specified token types when FOR appears, or for all mappings of the configuration when it doesn’t. The DROP MAPPING form removes all dictionaries for the specified token type(s), causing tokens of those types to be ignored by the text search configuration. It is an error if there is no mapping for the token types, unless IF EXISTS appears.

Examples

The following example replaces the english dictionary with the swedish dictionary anywhere that english is used within my_config.

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

Compatibility

There is no ALTER TEXT SEARCH CONFIGURATION statement in the SQL standard.

See Also

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION
ALTER TEXT SEARCH DICTIONARY

Name
ALTER TEXT SEARCH DICTIONARY — change the definition of a text search dictionary

Synopsis
ALTER TEXT SEARCH DICTIONARY name (option [ = value ] [, ... ])
ALTER TEXT SEARCH DICTIONARY name RENAME TO newname
ALTER TEXT SEARCH DICTIONARY name OWNER TO newowner

Description
ALTER TEXT SEARCH DICTIONARY changes the definition of a text search dictionary. You can change the dictionary’s template-specific options, or change the dictionary’s name or owner.
You must be the owner of the dictionary to use ALTER TEXT SEARCH DICTIONARY.

Parameters
name
The name (optionally schema-qualified) of an existing text search dictionary.

option
The name of a template-specific option to be set for this dictionary.

value
The new value to use for a template-specific option. If the equal sign and value are omitted, then any previous setting for the option is removed from the dictionary, allowing the default to be used.

newname
The new name of the text search dictionary.

newowner
The new owner of the text search dictionary.
Template-specific options can appear in any order.

Examples
The following example command changes the stopword list for a Snowball-based dictionary. Other parameters remain unchanged.

ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
The following example command changes the language option to dutch, and removes the stopword option entirely.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

The following example command “updates” the dictionary’s definition without actually changing anything.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

(The reason this works is that the option removal code doesn’t complain if there is no such option.) This trick is useful when changing configuration files for the dictionary: the `ALTER` will force existing database sessions to re-read the configuration files, which otherwise they would never do if they had read them earlier.

**Compatibility**

There is no `ALTER TEXT SEARCH DICTIONARY` statement in the SQL standard.

**See Also**

`CREATE TEXT SEARCH DICTIONARY`, `DROP TEXT SEARCH DICTIONARY`
ALTER TEXT SEARCH PARSER

Name

ALTER TEXT SEARCH PARSER — change the definition of a text search parser

Synopsis

ALTER TEXT SEARCH PARSER name RENAME TO newname

Description

ALTER TEXT SEARCH PARSER changes the definition of a text search parser. Currently, the only supported functionality is to change the parser’s name.

You must be a superuser to use ALTER TEXT SEARCH PARSER.

Parameters

name

The name (optionally schema-qualified) of an existing text search parser.

newname

The new name of the text search parser.

Compatibility

There is no ALTER TEXT SEARCH PARSER statement in the SQL standard.

See Also

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER
ALTER TEXT SEARCH TEMPLATE

Name

ALTER TEXT SEARCH TEMPLATE — change the definition of a text search template

Synopsis

ALTER TEXT SEARCH TEMPLATE name RENAME TO newname

Description

ALTER TEXT SEARCH TEMPLATE changes the definition of a text search template. Currently, the only supported functionality is to change the template’s name.

You must be a superuser to use ALTER TEXT SEARCH TEMPLATE.

Parameters

name

The name (optionally schema-qualified) of an existing text search template.

newname

The new name of the text search template.

Compatibility

There is no ALTER TEXT SEARCH TEMPLATE statement in the SQL standard.

See Also

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE
ALTER TRIGGER

Name
ALTER TRIGGER — change the definition of a trigger

Synopsis
ALTER TRIGGER name ON table RENAME TO newname

Description
ALTER TRIGGER changes properties of an existing trigger. The RENAME clause changes the name of the given trigger without otherwise changing the trigger definition.

You must own the table on which the trigger acts to be allowed to change its properties.

Parameters
name
   The name of an existing trigger to alter.

   table
   The name of the table on which this trigger acts.

   newname
   The new name for the trigger.

Notes
The ability to temporarily enable or disable a trigger is provided by ALTER TABLE, not by ALTER TRIGGER, because ALTER TRIGGER has no convenient way to express the option of enabling or disabling all of a table’s triggers at once.

Examples
To rename an existing trigger:

   ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;

Compatibility
ALTER TRIGGER is a PostgreSQL extension of the SQL standard.
See Also

ALTER TABLE
ALTER TYPE

**Name**

ALTER TYPE — change the definition of a type

**Synopsis**

ALTER TYPE name OWNER TO new_owner
ALTER TYPE name SET SCHEMA new_schema

**Description**

ALTER TYPE changes the definition of an existing type. The only currently available capabilities are changing the owner and schema of a type.

You must own the type to use ALTER TYPE. To change the schema of a type, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the type’s schema. (These restrictions enforce that altering the owner doesn’t do anything you couldn’t do by dropping and recreating the type. However, a superuser can alter ownership of any type anyway.)

**Parameters**

*name*

The name (possibly schema-qualified) of an existing type to alter.

*new_owner*

The user name of the new owner of the type.

*new_schema*

The new schema for the type.

**Examples**

To change the owner of the user-defined type `email` to `joe`:

```
ALTER TYPE email OWNER TO joe;
```

To change the schema of the user-defined type `email` to `customers`:

```
ALTER TYPE email SET SCHEMA customers;
```
Compatibility

There is no `ALTER TYPE` statement in the SQL standard.
ALTER USER

**Name**

ALTER USER — change a database role

**Synopsis**

```
ALTER USER name [ [ WITH ] option [ ... ] ]
```

where `option` can be:

- SUPERUSER | NOSUPERUSER
- CREATEDB | NOCREATEDB
- CREATEROLE | NOCREATEROLE
- CREATEUSER | NOCREATEUSER
- INHERIT | NOINHERIT
- LOGIN | NOLOGIN
- CONNECTION LIMIT `connlimit`
- [ ENCRYPTED | UNENCRYPTED ] PASSWORD ‘`password’`
- VALID UNTIL ‘`timestamp’`

```
ALTER USER name RENAME TO `newname`
```

```
ALTER USER name SET `configuration_parameter` { TO | = } { value | DEFAULT }
```

```
ALTER USER name SET `configuration_parameter` FROM CURRENT
```

```
ALTER USER name RESET `configuration_parameter`
```

```
ALTER USER name RESET ALL
```

**Description**

ALTER USER is now an alias for ALTER ROLE.

**Compatibility**

The ALTER USER statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

**See Also**

ALTER ROLE
ALTER VIEW

Name
ALTER VIEW — change the definition of a view

Synopsis
ALTER VIEW name RENAME TO newname

Description
ALTER VIEW changes the definition of a view. The only currently available functionality is to rename the view. To execute this command you must be the owner of the view.

Parameters

name
The name (optionally schema-qualified) of an existing view.

newname
The new name of the view.

Notes
Some variants of ALTER TABLE can be used with views as well; for example, to rename a view it is also possible to use ALTER TABLE RENAME. To change the schema or owner of a view, you currently must use ALTER TABLE.

Examples
To rename the view foo to bar:

ALTER VIEW foo RENAME TO bar;

Compatibility
ALTER VIEW is a PostgreSQL extension of the SQL standard.
See Also

CREATE VIEW, DROP VIEW
ANALYZE

Name
ANALYZE — collect statistics about a database

Synopsis
ANALYZE [ VERBOSE ] [ table [ ( column [, ...] ) ] ]

Description
ANALYZE collects statistics about the contents of tables in the database, and stores the results in the pg_statistic system catalog. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.

With no parameter, ANALYZE examines every table in the current database. With a parameter, ANALYZE examines only that table. It is further possible to give a list of column names, in which case only the statistics for those columns are collected.

Parameters

VERBOSE
Enables display of progress messages.

table
The name (possibly schema-qualified) of a specific table to analyze. Defaults to all tables in the current database.

column
The name of a specific column to analyze. Defaults to all columns.

Outputs

When VERBOSE is specified, ANALYZE emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

Notes

In the default PostgreSQL configuration, The Auto-Vacuum Daemon takes care of automatic analyzing of tables when they are first loaded with data, and as they change throughout regular operation. When autovacuum is disabled, it is a good idea to run ANALYZE periodically, or just after making major changes in the contents of a table. Accurate statistics will help the planner to choose the most appropriate query plan, and thereby improve the speed of query processing. A common strategy is to run VACUUM and ANALYZE once a day during a low-usage time of day.
ANALYZE requires only a read lock on the target table, so it can run in parallel with other activity on the table.

The statistics collected by ANALYZE usually include a list of some of the most common values in each column and a histogram showing the approximate data distribution in each column. One or both of these can be omitted if ANALYZE deems them uninteresting (for example, in a unique-key column, there are no common values) or if the column data type does not support the appropriate operators. There is more information about the statistics in Chapter 23.

For large tables, ANALYZE takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time. Note, however, that the statistics are only approximate, and will change slightly each time ANALYZE is run, even if the actual table contents did not change. This might result in small changes in the planner’s estimated costs shown by EXPLAIN. In rare situations, this non-determinism will cause the query optimizer to choose a different query plan between runs of ANALYZE. To avoid this, raise the amount of statistics collected by ANALYZE, as described below.

The extent of analysis can be controlled by adjusting the default_statistics_target configuration variable, or on a column-by-column basis by setting the per-column statistics target with ALTER TABLE ... ALTER COLUMN ... SET STATISTICS (see ALTER TABLE). The target value sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram. The default target value is 10, but this can be adjusted up or down to trade off accuracy of planner estimates against the time taken for ANALYZE and the amount of space occupied in pg_statistic. In particular, setting the statistics target to zero disables collection of statistics for that column. It might be useful to do that for columns that are never used as part of the WHERE, GROUP BY, or ORDER BY clauses of queries, since the planner will have no use for statistics on such columns.

The largest statistics target among the columns being analyzed determines the number of table rows sampled to prepare the statistics. Increasing the target causes a proportional increase in the time and space needed to do ANALYZE.

Compatibility

There is no ANALYZE statement in the SQL standard.

See Also

VACUUM, vacuumdb, Cost-Based Vacuum Delay, The Auto-Vacuum Daemon
BEGIN

Name

BEGIN — start a transaction block

Synopsis

BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ... ] ]

where transaction_mode is one of:

  ISOLATION LEVEL | SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
                   | READ WRITE    | READ ONLY

Description

BEGIN initiates a transaction block, that is, all statements after a BEGIN command will be executed in a single transaction until an explicit COMMIT or ROLLBACK is given. By default (without BEGIN), PostgreSQL executes transactions in “autocommit” mode, that is, each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done).

Statements are executed more quickly in a transaction block, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also useful to ensure consistency when making several related changes: other sessions will be unable to see the intermediate states wherein not all the related updates have been done.

If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if SET TRANSACTION was executed.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Refer to SET TRANSACTION for information on the meaning of the other parameters to this statement.

Notes

START TRANSACTION has the same functionality as BEGIN.

Use COMMIT or ROLLBACK to terminate a transaction block.

Issuing BEGIN when already inside a transaction block will provoke a warning message. The state of the transaction is not affected. To nest transactions within a transaction block, use savepoints (see SAVEPOINT).
For reasons of backwards compatibility, the commas between successive `transaction_modes` can be omitted.

**Examples**

To begin a transaction block:

```
BEGIN;
```

**Compatibility**

`BEGIN` is a PostgreSQL language extension. It is equivalent to the SQL-standard command `START TRANSACTION`, whose reference page contains additional compatibility information.

Incidentally, the `BEGIN` key word is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

**See Also**

`COMMIT`, `ROLLBACK`, `START TRANSACTION`, `SAVEPOINT`
CHECKPOINT

Name
CHECKPOINT — force a transaction log checkpoint

Synopsis
CHECKPOINT

Description
Write-Ahead Logging (WAL) puts a checkpoint in the transaction log every so often. (To adjust the automatic checkpoint interval, see the run-time configuration options checkpoint_segments and checkpoint_timeout.) The CHECKPOINT command forces an immediate checkpoint when the command is issued, without waiting for a scheduled checkpoint.

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to disk. Refer to Chapter 28 for more information about the WAL system.

Only superusers can call CHECKPOINT. The command is not intended for use during normal operation.

Compatibility

The CHECKPOINT command is a PostgreSQL language extension.
CLOSE

Name
CLOSE — close a cursor

Synopsis
CLOSE { name | ALL }

Description
CLOSE frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

Every non-holdable open cursor is implicitly closed when a transaction is terminated by COMMIT or ROLLBACK. A holdable cursor is implicitly closed if the transaction that created it aborts via ROLLBACK. If the creating transaction successfully commits, the holdable cursor remains open until an explicit CLOSE is executed, or the client disconnects.

Parameters
name
The name of an open cursor to close.
ALL
Close all open cursors.

Notes
PostgreSQL does not have an explicit OPEN cursor statement; a cursor is considered open when it is declared. Use the DECLARE statement to declare a cursor.

You can see all available cursors by querying the pg_cursors system view.

Examples
Close the cursor liahona:

CLOSE liahona;
Compatibility

CLOSE is fully conforming with the SQL standard. CLOSE ALL is a PostgreSQL extension.

See Also

DECLARE, FETCH, MOVE
CLUSTER

Name
CLUSTER — cluster a table according to an index

Synopsis
CLUSTER tablename [ USING indexname ]
CLUSTER

Description
CLUSTER instructs PostgreSQL to cluster the table specified by tablename based on the index specified by indexname. The index must already have been defined on tablename.

When a table is clustered, it is physically reordered based on the index information. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order. (If one wishes, one can periodically recluster by issuing the command again. Also, setting the table’s FILLFACTOR storage parameter to less than 100% can aid in preserving cluster ordering during updates, since updated rows are preferentially kept on the same page.)

When a table is clustered, PostgreSQL remembers which index it was clustered by. The form CLUSTER tablename reclusters the table using the same index as before.

CLUSTER without any parameter reclusters all the previously-clustered tables in the current database that the calling user owns, or all such tables if called by a superuser. This form of CLUSTER cannot be executed inside a transaction block.

When a table is being clustered, an ACCESS EXCLUSIVE lock is acquired on it. This prevents any other database operations (both reads and writes) from operating on the table until the CLUSTER is finished.

Parameters

tablename

The name (possibly schema-qualified) of a table.

indexname

The name of an index.

Notes
In cases where you are accessing single rows randomly within a table, the actual order of the data in the table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using CLUSTER. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, CLUSTER will help because once the index identifies the table page for the first row that matches, all other rows
that match are probably already on the same table page, and so you save disk accesses and speed up
the query.

During the cluster operation, a temporary copy of the table is created that contains the table data in
the index order. Temporary copies of each index on the table are created as well. Therefore, you need
free space on disk at least equal to the sum of the table size and the index sizes.

Because \texttt{CLUSTER} remembers the clustering information, one can cluster the tables one wants clus-
tered manually the first time, and setup a timed event similar to \texttt{VACUUM} so that the tables are period-
ically reclustered.

Because the planner records statistics about the ordering of tables, it is advisable to run \texttt{ANALYZE} on
the newly clustered table. Otherwise, the planner might make poor choices of query plans.

There is another way to cluster data. The \texttt{CLUSTER} command reorders the original table by scanning
it using the index you specify. This can be slow on large tables because the rows are fetched from the
table in index order, and if the table is disordered, the entries are on random pages, so there is one disk
page retrieved for every row moved. (PostgreSQL has a cache, but the majority of a big table will not
fit in the cache.) The other way to cluster a table is to use:

\begin{verbatim}
CREATE TABLE newtable AS
    SELECT * FROM table ORDER BY columnlist;
\end{verbatim}

which uses the PostgreSQL sorting code to produce the desired order; this is usually much faster than
an index scan for disordered data. Then you drop the old table, use \texttt{ALTER TABLE ... RENAME} to
rename \texttt{newtable} to the old name, and recreate the table’s indexes. The big disadvantage of this
approach is that it does not preserve OIDs, constraints, foreign key relationships, granted privileges,
and other ancillary properties of the table — all such items must be manually recreated. Another
disadvantage is that this way requires a sort temporary file about the same size as the table itself, so
peak disk usage is about three times the table size instead of twice the table size.

\section*{Examples}

Cluster the table \texttt{employees} on the basis of its index \texttt{employees\_ind}:

\begin{verbatim}
CLUSTER employees USING employees\_ind;
\end{verbatim}

Cluster the \texttt{employees} table using the same index that was used before:

\begin{verbatim}
CLUSTER employees;
\end{verbatim}

Cluster all tables in the database that have previously been clustered:

\begin{verbatim}
CLUSTER;
\end{verbatim}

\section*{Compatibility}

There is no \texttt{CLUSTER} statement in the SQL standard.

The syntax
CLUSTER indexname ON tablename

is also supported for compatibility with pre-8.3 PostgreSQL versions.

See Also

clusterdb
COMMENT

Name

COMMENT — define or change the comment of an object

Synopsis

COMMENT ON
{ TABLE object_name | COLUMN table_name.column_name | AGGREGATE agg_name { agg_type [, ... ] } | CAST { sourcetype AS targettype } | CONSTRAINT constraint_name ON table_name | CONVERSION object_name | DATABASE object_name | DOMAIN object_name | FUNCTION func_name ( [ [ argmode ] [ argname ] argtype [, ... ] ] ) | INDEX object_name | LARGE OBJECT large_object_oid | OPERATOR op ( leftoperand_type, rightoperand_type ) | OPERATOR CLASS object_name USING index_method | OPERATOR FAMILY object_name USING index_method | [ PROCEDURAL ] LANGUAGE object_name | ROLE object_name | RULE rule_name ON table_name | SCHEMA object_name | SEQUENCE object_name | TABLESPACE object_name | TEXT SEARCH CONFIGURATION object_name | TEXT SEARCH DICTIONARY object_name | TEXT SEARCH PARSER object_name | TEXT SEARCH TEMPLATE object_name | TRIGGER trigger_name ON table_name | TYPE object_name | VIEW object_name
} IS 'text'

Description

COMMENT stores a comment about a database object.

To modify a comment, issue a new COMMENT command for the same object. Only one comment string is stored for each object. To remove a comment, write NULL in place of the text string. Comments are automatically dropped when the object is dropped.

Comments can be viewed using psql's \d family of commands. Other user interfaces to retrieve comments can be built atop the same built-in functions that psql uses, namely obj_description, col_description, and shobj_description (see Table 9-48).
### Parameters

- **object_name**: The name of the object to be commented. Names of tables, aggregates, domains, functions, indexes, operators, operator classes, operator families, sequences, text search objects, types, and views can be schema-qualified.

- **table_name.column_name**: An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write `*` in place of the list of input data types.

- **agg_name**: The name of the source data type of the cast.

- **constraint_name**: The name of the target data type of the cast.

- **func_name**: The name of a function argument. Note that `COMMENT ON FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function’s identity.

- **op**: The mode of a function argument: either `IN`, `OUT`, or `INOUT`. If omitted, the default is `IN`. Note that `COMMENT ON FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function’s identity. So it is sufficient to list the `IN` and `INOUT` arguments.

- **rule_name**: The data type(s) of the function’s arguments (optionally schema-qualified), if any.

- **trigger_name**: The OID of the large object.

- **large_object_oid**: PROCEDURAL

  This is a noise word.

- **text**: The new comment, written as a string literal; or `NULL` to drop the comment.

### Notes

There is presently no security mechanism for comments: any user connected to a database can see all the comments for objects in that database (although only superusers can change comments for objects that they don’t own). For shared objects such as databases, roles, and tablespaces comments...
are stored globally and any user connected to any database can see all the comments for shared objects. Therefore, don’t put security-critical information in comments.

Examples

Attach a comment to the table mytable:

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

Remove it again:

```
COMMENT ON TABLE mytable IS NULL;
```

Some more examples:

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, text) IS 'This is a prefix operator on text';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for btrees';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for swedish language';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

Compatibility

There is no COMMENT command in the SQL standard.
COMMIT

Name
COMMIT — commit the current transaction

Synopsis
COMMIT [ WORK | TRANSACTION ]

Description
COMMIT commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters
WORK
TRANSACTION
Optional key words. They have no effect.

Notes
Use ROLLBACK to abort a transaction.
Issuing COMMIT when not inside a transaction does no harm, but it will provoke a warning message.

Examples
To commit the current transaction and make all changes permanent:
COMMIT;

Compatibility
The SQL standard only specifies the two forms COMMIT and COMMIT WORK. Otherwise, this command is fully conforming.

See Also
BEGIN, ROLLBACK
COMMIT PREPARED

Name

COMMIT PREPARED — commit a transaction that was earlier prepared for two-phase commit

Synopsis

COMMIT PREPARED transaction_id

Description

COMMIT PREPARED commits a transaction that is in prepared state.

Parameters

transaction_id

The transaction identifier of the transaction that is to be committed.

Notes

To commit a prepared transaction, you must be either the same user that executed the transaction originally, or a superuser. But you do not have to be in the same session that executed the transaction.

This command cannot be executed inside a transaction block. The prepared transaction is committed immediately.

All currently available prepared transactions are listed in the pg_prepared_xacts system view.

Examples

Commit the transaction identified by the transaction identifier foobar:

COMMIT PREPARED 'foobar';

See Also

PREPARE TRANSACTION, ROLLBACK PREPARED
COPY

Name
COPY — copy data between a file and a table

Synopsis
COPY tablename [ ( column [, ...] ) ]
  FROM { 'filename' | STDIN }
  [ [ WITH ]
    [ BINARY ]
    [ OIDS ]
    [ DELIMITER [ AS ] 'delimiter' ]
    [ NULL [ AS ] 'null string' ]
    [ CSV [ HEADER ]
      [ QUOTE [ AS ] 'quote' ]
      [ ESCAPE [ AS ] 'escape' ]
      [ FORCE NOT NULL column [, ...] ]
  ]
COPY { tablename [ ( column [, ...] ) ] | ( query ) } TO { 'filename' | STDOUT }
  [ [ WITH ]
    [ BINARY ]
    [ OIDS ]
    [ DELIMITER [ AS ] 'delimiter' ]
    [ NULL [ AS ] 'null string' ]
    [ CSV [ HEADER ]
      [ QUOTE [ AS ] 'quote' ]
      [ ESCAPE [ AS ] 'escape' ]
      [ FORCE QUOTE column [, ...] ]
  ]

Description
COPY moves data between PostgreSQL tables and standard file-system files. COPY TO copies the contents of a table to a file, while COPY FROM copies data from a file to a table (appending the data to whatever is in the table already). COPY TO can also copy the results of a SELECT query.

If a list of columns is specified, COPY will only copy the data in the specified columns to or from the file. If there are any columns in the table that are not in the column list, COPY FROM will insert the default values for those columns.

COPY with a file name instructs the PostgreSQL server to directly read from or write to a file. The file must be accessible to the server and the name must be specified from the viewpoint of the server. When STDIN or STDOUT is specified, data is transmitted via the connection between the client and the server.


Parameters

tablename

The name (optionally schema-qualified) of an existing table.

column

An optional list of columns to be copied. If no column list is specified, all columns of the table will be copied.

query

A SELECT or VALUES command whose results are to be copied. Note that parentheses are required around the query.

filename

The absolute path name of the input or output file. Windows users might need to use an `E" string and double backslashes used as path separators.

STDIN

Specifies that input comes from the client application.

STDOUT

Specifies that output goes to the client application.

BINARY

Causes all data to be stored or read in binary format rather than as text. You cannot specify the DELIMITER, NULL, or CSV options in binary mode.

OIDS

Specifies copying the OID for each row. (An error is raised if OIDS is specified for a table that does not have OIDs, or in the case of copying a query.)

delimiter

The single ASCII character that separates columns within each row (line) of the file. The default is a tab character in text mode, a comma in CSV mode.

null string

The string that represents a null value. The default is `\N` (backslash-N) in text mode, and an empty value with no quotes in CSV mode. You might prefer an empty string even in text mode for cases where you don’t want to distinguish nulls from empty strings.

Note: When using COPY FROM, any data item that matches this string will be stored as a null value, so you should make sure that you use the same string as you used with COPY TO.

CSV

Selects Comma Separated Value (CSV) mode.

HEADER

Specifies that the file contains a header line with the names of each column in the file. On output, the first line contains the column names from the table, and on input, the first line is ignored.

quote

Specifies the ASCII quotation character in CSV mode. The default is double-quote.
escape

Specifies the ASCII character that should appear before a "QUOTE" data character value in CSV mode. The default is the "QUOTE" value (usually double-quote).

FORCE QUOTE

In CSV COPY TO mode, forces quoting to be used for all non-NULL values in each specified column. NULL output is never quoted.

FORCE NOT NULL

In CSV COPY FROM mode, process each specified column as though it were quoted and hence not a NULL value. For the default null string in CSV mode (""), this causes missing values to be input as zero-length strings.

Outputs

On successful completion, a COPY command returns a command tag of the form

COPY count

The count is the number of rows copied.

Notes

COPY can only be used with plain tables, not with views. However, you can write COPY (SELECT * FROM viewname) TO ....

The BINARY key word causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the normal text mode, but a binary-format file is less portable across machine architectures and PostgreSQL versions.

You must have select privilege on the table whose values are read by COPY TO, and insert privilege on the table into which values are inserted by COPY FROM.

Files named in a COPY command are read or written directly by the server, not by the client application. Therefore, they must reside on or be accessible to the database server machine, not the client. They must be accessible to and readable or writable by the PostgreSQL user (the user ID the server runs as), not the client. COPY naming a file is only allowed to database superusers, since it allows reading or writing any file that the server has privileges to access.

Do not confuse COPY with the psql instruction \copy. \copy invokes COPY FROM STDIN or COPY TO STDOUT, and then fetches/stores the data in a file accessible to the psql client. Thus, file accessibility and access rights depend on the client rather than the server when \copy is used.

It is recommended that the file name used in COPY always be specified as an absolute path. This is enforced by the server in the case of COPY TO, but for COPY FROM you do have the option of reading from a file specified by a relative path. The path will be interpreted relative to the working directory of the server process (normally the cluster’s data directory), not the client’s working directory.

COPY FROM will invoke any triggers and check constraints on the destination table. However, it will not invoke rules.

COPY input and output is affected by DateStyle. To ensure portability to other PostgreSQL installations that might use non-default DateStyle settings, DateStyle should be set to ISO before using COPY TO.
Input data is interpreted according to the current client encoding, and output data is encoded in the current client encoding, even if the data does not pass through the client but is read from or written to a file.

`COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY TO`, but the target table will already have received earlier rows in a `COPY FROM`. These rows will not be visible or accessible, but they still occupy disk space. This might amount to a considerable amount of wasted disk space if the failure happened well into a large copy operation. You might wish to invoke `VACUUM` to recover the wasted space.

**File Formats**

**Text Format**

When `COPY` is used without the `BINARY` or `CSV` options, the data read or written is a text file with one line per table row. Columns in a row are separated by the delimiter character. The column values themselves are strings generated by the output function, or acceptable to the input function, of each attribute’s data type. The specified null string is used in place of columns that are null. `COPY FROM` will raise an error if any line of the input file contains more or fewer columns than are expected. If `OID` is specified, the OID is read or written as the first column, preceding the user data columns.

End of data can be represented by a single line containing just backslash-period (`\.`). An end-of-data marker is not necessary when reading from a file, since the end of file serves perfectly well; it is needed only when copying data to or from client applications using pre-3.0 client protocol.

Backslash characters (`\`) can be used in the `COPY` data to quote data characters that might otherwise be taken as row or column delimiters. In particular, the following characters must be preceded by a backslash if they appear as part of a column value: backslash itself, newline, carriage return, and the current delimiter character.

The specified null string is sent by `COPY TO` without adding any backslashes; conversely, `COPY FROM` matches the input against the null string before removing backslashes. Therefore, a null string such as `\N` cannot be confused with the actual data value `\N` (which would be represented as `\\N`).

The following special backslash sequences are recognized by `COPY FROM`:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\b</code></td>
<td>Backspace (ASCII 8)</td>
</tr>
<tr>
<td><code>\f</code></td>
<td>Form feed (ASCII 12)</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>Newline (ASCII 10)</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage return (ASCII 13)</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Tab (ASCII 9)</td>
</tr>
<tr>
<td><code>\v</code></td>
<td>Vertical tab (ASCII 11)</td>
</tr>
<tr>
<td><code>\digits</code></td>
<td>Backslash followed by one to three octal digits specifies the character with that numeric code</td>
</tr>
<tr>
<td><code>\xdigits</code></td>
<td>Backslash \ followed by one or two hex digits specifies the character with that numeric code</td>
</tr>
</tbody>
</table>

Presently, `COPY TO` will never emit an octal or hex-digits backslash sequence, but it does use the other sequences listed above for those control characters.

Any other backslashed character that is not mentioned in the above table will be taken to represent
itself. However, beware of adding backslashes unnecessarily, since that might accidentally produce a string matching the end-of-data marker (\.) or the null string (\N by default). These strings will be recognized before any other backslash processing is done.

It is strongly recommended that applications generating COPY data convert data newlines and carriage returns to the \n and \r sequences respectively. At present it is possible to represent a data carriage return by a backslash and carriage return, and to represent a data newline by a backslash and newline. However, these representations might not be accepted in future releases. They are also highly vulnerable to corruption if the COPY file is transferred across different machines (for example, from Unix to Windows or vice versa).

COPY TO will terminate each row with a Unix-style newline ("\n"). Servers running on Microsoft Windows instead output carriage return/newline ("\r\n"), but only for COPY to a server file; for consistency across platforms, COPY TO STDOUT always sends "\n" regardless of server platform. COPY FROM can handle lines ending with newlines, carriage returns, or carriage return/newlines. To reduce the risk of error due to un-backslashed newlines or carriage returns that were meant as data, COPY FROM will complain if the line endings in the input are not all alike.

CSV Format

This format is used for importing and exporting the Comma Separated Value (CSV) file format used by many other programs, such as spreadsheets. Instead of the escaping used by PostgreSQL’s standard text mode, it produces and recognizes the common CSV escaping mechanism.

The values in each record are separated by the DELIMITER character. If the value contains the delimiter character, the QUOTE character, the NULL string, a carriage return, or line feed character, then the whole value is prefixed and suffixed by the QUOTE character, and any occurrence within the value of a QUOTE character or the ESCAPE character is preceded by the escape character. You can also use FORCE QUOTE to force quotes when outputting non-NULL values in specific columns.

The CSV format has no standard way to distinguish a NULL value from an empty string. PostgreSQL’s COPY handles this by quoting. A NULL is output as the NULL string and is not quoted, while a data value matching the NULL string is quoted. Therefore, using the default settings, a NULL is written as an unquoted empty string, while an empty string is written with double quotes ("""). Reading values follows similar rules. You can use FORCE NOT NULL to prevent NULL input comparisons for specific columns.

Because backslash is not a special character in the CSV format, \\., the end-of-data marker, could also appear as a data value. To avoid any misinterpretation, a \\ data value appearing as a lone entry on a line is automatically quoted on output, and on input, if quoted, is not interpreted as the end-of-data marker. If you are loading a file created by another application that has a single unquoted column and might have a value of \\., you might need to quote that value in the input file.

Note: In CSV mode, all characters are significant. A quoted value surrounded by white space, or any characters other than DELIMITER, will include those characters. This can cause errors if you import data from a system that pads CSV lines with white space out to some fixed width. If such a situation arises you might need to preprocess the CSV file to remove the trailing white space, before importing the data into PostgreSQL.

Note: CSV mode will both recognize and produce CSV files with quoted values containing embedded carriage returns and line feeds. Thus the files are not strictly one line per table row like text-mode files.
Note: Many programs produce strange and occasionally perverse CSV files, so the file format is more a convention than a standard. Thus you might encounter some files that cannot be imported using this mechanism, and COPY might produce files that other programs cannot process.

Binary Format

The file format used for COPY BINARY changed in PostgreSQL 7.4. The new format consists of a file header, zero or more tuples containing the row data, and a file trailer. Headers and data are now in network byte order.

File Header

The file header consists of 15 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:

Signature

11-byte sequence PGCOPY\n\377\r\n\0 — note that the zero byte is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by end-of-line-translation filters, dropped zero bytes, dropped high bits, or parity changes.)

Flags field

32-bit integer bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB). Note that this field is stored in network byte order (most significant byte first), as are all the integer fields used in the file format. Bits 16-31 are reserved to denote critical file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0-15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag bit is defined, and the rest must be zero:

Bit 16

if 1, OIDs are included in the data; if 0, not

Header extension area length

32-bit integer, length in bytes of remainder of header, not including self. Currently, this is zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with.

The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.

This design allows for both backwards-compatible header additions (add header extension chunks, or set low-order flag bits) and non-backwards-compatible changes (set high-order flag bits to signal such changes, and add supporting data to the extension area if needed).
**Tuples**

Each tuple begins with a 16-bit integer count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is a 32-bit length word followed by that many bytes of field data. (The length word does not include itself, and can be zero.) As a special case, -1 indicates a NULL field value. No value bytes follow in the NULL case.

There is no alignment padding or any other extra data between fields.

Presently, all data values in a COPY BINARY file are assumed to be in binary format (format code one). It is anticipated that a future extension might add a header field that allows per-column format codes to be specified.

To determine the appropriate binary format for the actual tuple data you should consult the PostgreSQL source, in particular the *send* and *recv* functions for each column’s data type (typically these functions are found in the src/backend/utils/adt/ directory of the source distribution).

If OIDs are included in the file, the OID field immediately follows the field-count word. It is a normal field except that it’s not included in the field-count. In particular it has a length word — this will allow handling of 4-byte vs. 8-byte OIDs without too much pain, and will allow OIDs to be shown as null if that ever proves desirable.

**File Trailer**

The file trailer consists of a 16-bit integer word containing -1. This is easily distinguished from a tuple’s field-count word.

A reader should report an error if a field-count word is neither -1 nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

**Examples**

The following example copies a table to the client using the vertical bar (|) as the field delimiter:

COPY country TO STDOUT WITH DELIMITER ‘|’;

To copy data from a file into the country table:

COPY country FROM ’/usr1/proj/bray/sql/country_data’;

To copy into a file just the countries whose names start with ‘A’:

COPY (SELECT * FROM country WHERE country_name LIKE ’A%’) TO ’/usr1/proj/bray/sql/a_list_countries.copy’;

Here is a sample of data suitable for copying into a table from STDIN:

```
AF  AFGHANISTAN
AL  ALBANIA
DZ  ALGERIA
ZM  ZAMBIA
```
Note that the white space on each line is actually a tab character.

The following is the same data, output in binary format. The data is shown after filtering through the Unix utility `od -c`. The table has three columns: the first has type `char(2)`, the second has type `text`, and the third has type `integer`. All the rows have a null value in the third column.

```
0000000 P G C O P Y \n \n 0 \n 0 0 0 3 0 0 0 0 2 A F 0 0 0 1 3 A
0000040 F G H A N I S T A N 377 377 377 377 0 0 3
0000060 0 0 0 2 A L I 0 0 0 0 7 A L B A N I
0000100 A 377 377 377 377 0 0 3 0 0 0 2 D Z 0 0 0 0
0000120 0 0 7 A L G E R I A 377 377 377 377 0 0 3 0 0 0
0000140 0 0 2 Z M 0 0 0 0 6 Z A M B I A 377 377 377 0 0 0 0
0000160 377 377 0 0 3 0 0 0 0 2 Z W 0 0 0 0 0 0 Z I
0000200 M B A B W E 377 377 377 377 377 377 377
```

**Compatibility**

There is no `COPY` statement in the SQL standard.

The following syntax was used before PostgreSQL version 7.3 and is still supported:

```
COPY [ BINARY ] tablename [ WITH OIDS ]
FROM { 'filename' | STDIN }
[ [USING] DELIMITERS 'delimiter' ]
[ WITH NULL AS 'null string' ]
COPY [ BINARY ] tablename [ WITH OIDS ]
TO { 'filename' | STDOUT }
[ [USING] DELIMITERS 'delimiter' ]
[ WITH NULL AS 'null string' ]
```
CREATE AGGREGATE

Name
CREATE AGGREGATE — define a new aggregate function

Synopsis
CREATE AGGREGATE name ( input_data_type [ , ... ] ) (  
  SFUNC = sfunc,  
  STYPE = state_data_type  
  [ , FINALFUNC = ffunc ]  
  [ , INITCOND = initial_condition ]  
  [ , SORTOP = sort_operator ]  
)

or the old syntax
CREATE AGGREGATE name (  
  BASETYPE = base_type,  
  SFUNC = sfunc,  
  STYPE = state_data_type  
  [ , FINALFUNC = ffunc ]  
  [ , INITCOND = initial_condition ]  
  [ , SORTOP = sort_operator ]  
)

Description
CREATE AGGREGATE defines a new aggregate function. Some basic and commonly-used aggregate functions are included with the distribution; they are documented in Section 9.18. If one defines new types or needs an aggregate function not already provided, then CREATE AGGREGATE can be used to provide the desired features.

If a schema name is given (for example, CREATE AGGREGATE myschema.myagg ...) then the aggregate function is created in the specified schema. Otherwise it is created in the current schema.

An aggregate function is identified by its name and input data type(s). Two aggregates in the same schema can have the same name if they operate on different input types. The name and input data type(s) of an aggregate must also be distinct from the name and input data type(s) of every ordinary function in the same schema.

An aggregate function is made from one or two ordinary functions: a state transition function \texttt{sfunc}, and an optional final calculation function \texttt{ffunc}. These are used as follows:

\texttt{sfunc}( internal-state, next-data-values ) \rightarrow next-internal-state  
\texttt{ffunc}( internal-state ) \rightarrow aggregate-value

PostgreSQL creates a temporary variable of data type \texttt{stype} to hold the current internal state of the aggregate. At each input row, the aggregate argument value(s) are calculated and the state transition...
CREATE AGGREGATE

function is invoked with the current state value and the new argument value(s) to calculate a new internal state value. After all the rows have been processed, the final function is invoked once to calculate the aggregate’s return value. If there is no final function then the ending state value is returned as-is.

An aggregate function can provide an initial condition, that is, an initial value for the internal state value. This is specified and stored in the database as a value of type text, but it must be a valid external representation of a constant of the state value data type. If it is not supplied then the state value starts out null.

If the state transition function is declared “strict”, then it cannot be called with null inputs. With such a transition function, aggregate execution behaves as follows. Rows with any null input values are ignored (the function is not called and the previous state value is retained). If the initial state value is null, then at the first row with all-nonnull input values, the first argument value replaces the state value, and the transition function is invoked at subsequent rows with all-nonnull input values. This is handy for implementing aggregates like max. Note that this behavior is only available when state_data_type is the same as the first input_data_type. When these types are different, you must supply a nonnull initial condition or use a nonstrict transition function.

If the state transition function is not strict, then it will be called unconditionally at each input row, and must deal with null inputs and null transition values for itself. This allows the aggregate author to have full control over the aggregate’s handling of null values.

If the final function is declared “strict”, then it will not be called when the ending state value is null; instead a null result will be returned automatically. (Of course this is just the normal behavior of strict functions.) In any case the final function has the option of returning a null value. For example, the final function for avg returns null when it sees there were zero input rows.

Aggregates that behave like MIN or MAX can sometimes be optimized by looking into an index instead of scanning every input row. If this aggregate can be so optimized, indicate it by specifying a sort operator. The basic requirement is that the aggregate must yield the first element in the sort ordering induced by the operator; in other words:

```
SELECT agg(col) FROM tab;
```

must be equivalent to:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Further assumptions are that the aggregate ignores null inputs, and that it delivers a null result if and only if there were no non-null inputs. Ordinarily, a data type’s < operator is the proper sort operator for MIN, and > is the proper sort operator for MAX. Note that the optimization will never actually take effect unless the specified operator is the “less than” or “greater than” strategy member of a B-tree index operator class.

Parameters

name

The name (optionally schema-qualified) of the aggregate function to create.

input_data_type

An input data type on which this aggregate function operates. To create a zero-argument aggregate function, write * in place of the list of input data types. (An example of such an aggregate is count(*).)
CREATE AGGREGATE

base_type

In the old syntax for CREATE AGGREGATE, the input data type is specified by a basetype parameter rather than being written next to the aggregate name. Note that this syntax allows only one input parameter. To define a zero-argument aggregate function, specify the basetype as "ANY" (not *).

sfunc

The name of the state transition function to be called for each input row. For an N-argument aggregate function, the sfunc must take N+1 arguments, the first being of type state_data_type and the rest matching the declared input data type(s) of the aggregate. The function must return a value of type state_data_type. This function takes the current state value and the current input data value(s), and returns the next state value.

state_data_type

The data type for the aggregate’s state value.

ffunc

The name of the final function called to compute the aggregate’s result after all input rows have been traversed. The function must take a single argument of type state_data_type. The return data type of the aggregate is defined as the return type of this function. If ffunc is not specified, then the ending state value is used as the aggregate’s result, and the return type is state_data_type.

initial_condition

The initial setting for the state value. This must be a string constant in the form accepted for the data type state_data_type. If not specified, the state value starts out null.

sort_operator

The associated sort operator for a MIN- or MAX-like aggregate. This is just an operator name (possibly schema-qualified). The operator is assumed to have the same input data types as the aggregate (which must be a single-argument aggregate).

The parameters of CREATE AGGREGATE can be written in any order, not just the order illustrated above.

Examples

See Section 34.10.

Compatibility

CREATE AGGREGATE is a PostgreSQL language extension. The SQL standard does not provide for user-defined aggregate functions.

See Also

ALTER AGGREGATE, DROP AGGREGATE
CREATE CAST

Name
CREATE CAST — define a new cast

Synopsis

CREATE CAST (sourcetype AS targettype)
   WITH FUNCTION funcname (argtypes)
      [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (sourcetype AS targettype)
   WITHOUT FUNCTION
      [ AS ASSIGNMENT | AS IMPLICIT ]

Description

CREATE CAST defines a new cast. A cast specifies how to perform a conversion between two data types. For example:

```
SELECT CAST(42 AS float8);
```

converts the integer constant 42 to type float8 by invoking a previously specified function, in this case float8(int4). (If no suitable cast has been defined, the conversion fails.)

Two types can be binary compatible, which means that they can be converted into one another “for free” without invoking any function. This requires that corresponding values use the same internal representation. For instance, the types text and varchar are binary compatible.

By default, a cast can be invoked only by an explicit cast request, that is an explicit `CAST(x AS typename)` or `x::typename` construct.

If the cast is marked AS ASSIGNMENT then it can be invoked implicitly when assigning a value to a column of the target data type. For example, supposing that foo.f1 is a column of type text, then:

```
INSERT INTO foo (f1) VALUES (42);
```

will be allowed if the cast from type integer to type text is marked AS ASSIGNMENT, otherwise not. (We generally use the term assignment cast to describe this kind of cast.)

If the cast is marked AS IMPLICIT then it can be invoked implicitly in any context, whether assignment or internally in an expression. (We generally use the term implicit cast to describe this kind of cast.) For example, consider this query:

```
SELECT 2 + 4.0;
```

The parser initially marks the constants as being of type integer and numeric respectively. There is no integer + numeric operator in the system catalogs, but there is a numeric + numeric operator. The query will therefore succeed if a cast from integer to numeric is available and is marked AS IMPLICIT — which in fact it is. The parser will apply the implicit cast and resolve the query as if it had been written

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```
Now, the catalogs also provide a cast from numeric to integer. If that cast were marked AS IMPLICIT — which it is not — then the parser would be faced with choosing between the above interpretation and the alternative of casting the numeric constant to integer and applying the integer + integer operator. Lacking any knowledge of which choice to prefer, it would give up and declare the query ambiguous. The fact that only one of the two casts is implicit is the way in which we teach the parser to prefer resolution of a mixed numeric-and-integer expression as numeric; there is no built-in knowledge about that.

It is wise to be conservative about marking casts as implicit. An overabundance of implicit casting paths can cause PostgreSQL to choose surprising interpretations of commands, or to be unable to resolve commands at all because there are multiple possible interpretations. A good rule of thumb is to make a cast implicitly invokable only for information-preserving transformations between types in the same general type category. For example, the cast from int2 to int4 can reasonably be implicit, but the cast from float8 to int4 should probably be assignment-only. Cross-type-category casts, such as text to int4, are best made explicit-only.

To be able to create a cast, you must own the source or the target data type. To create a binary-compatible cast, you must be superuser. (This restriction is made because an erroneous binary-compatible cast conversion can easily crash the server.)

**Parameters**

**sourcetype**

The name of the source data type of the cast.

**targettype**

The name of the target data type of the cast.

**funcname(argtypes)**

The function used to perform the cast. The function name can be schema-qualified. If it is not, the function will be looked up in the schema search path. The function’s result data type must match the target type of the cast. Its arguments are discussed below.

**WITHOUT FUNCTION**

Indicates that the source type and the target type are binary compatible, so no function is required to perform the cast.

**AS ASSIGNMENT**

Indicates that the cast can be invoked implicitly in assignment contexts.

**AS IMPLICIT**

Indicates that the cast can be invoked implicitly in any context.

Cast implementation functions can have one to three arguments. The first argument type must be identical to the cast’s source type. The second argument, if present, must be type integer; it receives the type modifier associated with the destination type, or -1 if there is none. The third argument, if present, must be type boolean; it receives true if the cast is an explicit cast, false otherwise. (Bizarrely, the SQL spec demands different behaviors for explicit and implicit casts in some cases. This argument is supplied for functions that must implement such casts. It is not recommended that you design your own data types so that this matters.)
Ordinarily a cast must have different source and target data types. However, it is allowed to declare a cast with identical source and target types if it has a cast implementation function with more than one argument. This is used to represent type-specific length coercion functions in the system catalogs. The named function is used to coerce a value of the type to the type modifier value given by its second argument.

When a cast has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

Notes

Use `DROP CAST` to remove user-defined casts.

Remember that if you want to be able to convert types both ways you need to declare casts both ways explicitly.

It is normally not necessary to create casts between user-defined types and the standard string types (`text`, `varchar`, and `char(n)`). PostgreSQL will automatically handle a cast to a string type by invoking the other type’s output function, or conversely handle a cast from a string type by invoking the other type’s input function. These automatically-provided casts are known as I/O conversion casts. I/O conversion casts to string types are treated as assignment casts, while I/O conversion casts from string types are explicit-only. You can override this behavior by declaring your own cast to replace an I/O conversion cast, but usually the only reason to do so is if you want the conversion to be more easily invokable than the standard assignment-only or explicit-only setting. Another possible reason is that you want the conversion to behave differently from the type’s I/O function; but that is sufficiently surprising that you should think twice about whether it’s a good idea. (A small number of the built-in types do indeed have different behaviors for conversions, mostly because of requirements of the SQL standard.)

Prior to PostgreSQL 7.3, every function that had the same name as a data type, returned that data type, and took one argument of a different type was automatically a cast function. This convention has been abandoned in face of the introduction of schemas and to be able to represent binary compatible casts in the system catalogs. The built-in cast functions still follow this naming scheme, but they have to be shown as casts in the system catalog `pg_cast` as well.

While not required, it is recommended that you continue to follow this old convention of naming cast implementation functions after the target data type. Many users are used to being able to cast data types using a function-style notation, that is `typename(x)`. This notation is in fact nothing more nor less than a call of the cast implementation function; it is not specially treated as a cast. If your conversion functions are not named to support this convention then you will have surprised users. Since PostgreSQL allows overloading of the same function name with different argument types, there is no difficulty in having multiple conversion functions from different types that all use the target type’s name.

Note: Actually the preceding paragraph is an oversimplification: there are two cases in which a function-call construct will be treated as a cast request without having matched it to an actual function. If a function call `name(x)` does not exactly match any existing function, but `name` is the name of a data type and `pg_cast` provides a binary-compatible cast to this type from the type of `x`, then the call will be construed as a binary-compatible cast. This exception is made so that binary-compatible casts can be invoked using functional syntax, even though they lack any function. Likewise, if there is no `pg_cast` entry but the cast would be to or from a string type, the call will
be construed as an I/O conversion cast. This exception allows I/O conversion casts to be invoked using functional syntax.

Examples

To create a cast from type $\texttt{bigint}$ to type $\texttt{int4}$ using the function $\texttt{int4} (\texttt{bigint})$:

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint);
```

(This cast is already predefined in the system.)

Compatibility

The `CREATE CAST` command conforms to the SQL standard, except that SQL does not make provisions for binary-compatible types or extra arguments to implementation functions. `$\texttt{AS IMPLICIT}$` is a PostgreSQL extension, too.

See Also

`CREATE FUNCTION`, `CREATE TYPE`, `DROP CAST`
CREATE CONSTRAINT TRIGGER

Name

CREATE CONSTRAINT TRIGGER — define a new constraint trigger

Synopsis

CREATE CONSTRAINT TRIGGER name
    AFTER event [ OR ... ]
    ON table_name
    [ FROM referenced_table_name ]
    { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }
    FOR EACH ROW
    EXECUTE PROCEDURE funcname ( arguments )

Description

CREATE CONSTRAINT TRIGGER creates a constraint trigger. This is the same as a regular trigger except that the timing of the trigger firing can be adjusted using SET CONSTRAINTS. Constraint triggers must be AFTER ROW triggers. They can be fired either at the end of the statement causing the triggering event, or at the end of the containing transaction; in the latter case they are said to be deferred. A pending deferred-trigger firing can also be forced to happen immediately by using SET CONSTRAINTS.

Parameters

name

The name of the constraint trigger. This is also the name to use when modifying the trigger’s behavior using SET CONSTRAINTS. The name cannot be schema-qualified — the trigger inherits the schema of its table.

event

One of INSERT, UPDATE, or DELETE; this specifies the event that will fire the trigger. Multiple events can be specified using OR.

table_name

The (possibly schema-qualified) name of the table in which the triggering events occur.

referenced_table_name

The (possibly schema-qualified) name of another table referenced by the constraint. This option is used for foreign-key constraints and is not recommended for general use.
CREATE CONSTRAINT TRIGGER

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED

The default timing of the trigger. See the CREATE TABLE documentation for details of these constraint options.

funcname

The function to call when the trigger is fired. See CREATE TRIGGER for details.

arguments

Optional argument strings to pass to the trigger function. See CREATE TRIGGER for details.

Compatibility

CREATE CONSTRAINT TRIGGER is a PostgreSQL extension of the SQL standard.

See Also

CREATE TRIGGER, DROP TRIGGER, SET CONSTRAINTS
CREATE CONVERSION

Name
CREATE CONVERSION — define a new encoding conversion

Synopsis
CREATE [ DEFAULT ] CONVERSION name
    FOR source_encoding TO dest_encoding FROM funcname

Description
CREATE CONVERSION defines a new conversion between character set encodings. Also, conversions that are marked DEFAULT can be used for automatic encoding conversion between client and server. For this purpose, two conversions, from encoding A to B and from encoding B to A, must be defined.

To be able to create a conversion, you must have EXECUTE privilege on the function and CREATE privilege on the destination schema.

Parameters

DEFAULT
The DEFAULT clause indicates that this conversion is the default for this particular source to destination encoding. There should be only one default encoding in a schema for the encoding pair.

name
The name of the conversion. The conversion name can be schema-qualified. If it is not, the conversion is defined in the current schema. The conversion name must be unique within a schema.

source_encoding
The source encoding name.

dest_encoding
The destination encoding name.

funcname
The function used to perform the conversion. The function name can be schema-qualified. If it is not, the function will be looked up in the path.

The function must have the following signature:

conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
    cstring,  -- source string (null terminated C string)
    internal, -- destination (fill with a null terminated C string)
    integer,  -- source string length
) RETURNS void;
Notes

Use \texttt{DROP CONVERSION} to remove user-defined conversions.
The privileges required to create a conversion might be changed in a future release.

Examples

To create a conversion from encoding \texttt{UTF8} to \texttt{LATIN1} using \texttt{myfunc}:

\begin{verbatim}
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
\end{verbatim}

Compatibility

\texttt{CREATE CONVERSION} is a PostgreSQL extension. There is no \texttt{CREATE CONVERSION} statement in the SQL standard.

See Also

\texttt{ALTER CONVERSION}, \texttt{CREATE FUNCTION}, \texttt{DROP CONVERSION}
CREATE DATABASE

Name
CREATE DATABASE — create a new database

Synopsis
CREATE DATABASE name
  [ [ WITH ] [ OWNER [=] dbowner ]
  [ TEMPLATE [=] template ]
  [ ENCODING [=] encoding ]
  [ TABLESPACE [=] tablespace ]
  [ CONNECTION LIMIT [=] conlimit ] ]

Description
CREATE DATABASE creates a new PostgreSQL database.
To create a database, you must be a superuser or have the special CREATEDB privilege. See CREATE USER.
By default, the new database will be created by cloning the standard system database template1. A different template can be specified by writing TEMPLATE name. In particular, by writing TEMPLATE template0, you can create a virgin database containing only the standard objects predefined by your version of PostgreSQL. This is useful if you wish to avoid copying any installation-local objects that might have been added to template1.

Parameters
name
The name of a database to create.
dbowner
The role name of the user who will own the new database, or DEFAULT to use the default (namely, the user executing the command). To create a database owned by another role, you must be a direct or indirect member of that role, or be a superuser.
template
The name of the template from which to create the new database, or DEFAULT to use the default template (template1).
encoding
Character set encoding to use in the new database. Specify a string constant (e.g., ‘SQL_ASCII’), or an integer encoding number, or DEFAULT to use the default encoding (namely, the encoding of the template database). The character sets supported by the PostgreSQL server are described in Section 22.2.1. See below for additional restrictions.
CREATE DATABASE

tablespace

The name of the tablespace that will be associated with the new database, or DEFAULT to use the
template database’s tablespace. This tablespace will be the default tablespace used for objects
created in this database. See CREATE TABLESPACE for more information.

connlimit

How many concurrent connections can be made to this database. -1 (the default) means no limit.
Optional parameters can be written in any order, not only the order illustrated above.

Notes

CREATE DATABASE cannot be executed inside a transaction block.
Errors along the line of “could not initialize database directory” are most likely related to insufficient
permissions on the data directory, a full disk, or other file system problems.
Use DROP DATABASE to remove a database.
The program createdb is a wrapper program around this command, provided for convenience.
Although it is possible to copy a database other than template1 by specifying its name as the tem-
plate, this is not (yet) intended as a general-purpose “COPY DATABASE” facility. The principal lim-
itation is that no other sessions can be connected to the template database while it is being copied.
CREATE DATABASE will fail if any other connection exists when it starts; otherwise, new connections
to the template database are locked out until CREATE DATABASE completes. See Section 20.3 for
more information.
Any character set encoding specified for the new database must be compatible with the server’s
LC_CTYPE locale setting. If LC_CTYPE is C (or equivalently POSIX), then all encodings are allowed,
but for other locale settings there is only one encoding that will work properly, and so the apparent
freedom to specify an encoding is illusory if you didn’t initialize the database cluster in C locale.
CREATE DATABASE will allow superusers to specify SQL_ASCII encoding regardless of the locale
setting, but this choice is deprecated and may result in misbehavior of character-string functions if
data that is not encoding-compatible with the locale is stored in the database.
The CONNECTION LIMIT option is only enforced approximately; if two new sessions start at about
the same time when just one connection “slot” remains for the database, it is possible that both will
fail. Also, the limit is not enforced against superusers.

Examples

To create a new database:

CREATE DATABASE lusiadas;

To create a database sales owned by user salesapp with a default tablespace of salesspace:

CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;

To create a database music which supports the ISO-8859-1 character set:
CREATE DATABASE music ENCODING 'LATIN1';

Compatibility

There is no CREATE DATABASE statement in the SQL standard. Databases are equivalent to catalogs, whose creation is implementation-defined.

See Also

ALTER DATABASE, DROP DATABASE
CREATE DOMAIN

Name

CREATE DOMAIN — define a new domain

Synopsis

CREATE DOMAIN name [ AS ] data_type
    [ DEFAULT expression ]
    [ constraint [ ... ] ]

where constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }

Description

CREATE DOMAIN creates a new domain. A domain is essentially a data type with optional constraints (restrictions on the allowed set of values). The user who defines a domain becomes its owner.

If a schema name is given (for example, CREATE DOMAIN myschema.mydomain ...) then the domain is created in the specified schema. Otherwise it is created in the current schema. The domain name must be unique among the types and domains existing in its schema.

Domains are useful for abstracting common constraints on fields into a single location for maintenance. For example, several tables might contain email address columns, all requiring the same CHECK constraint to verify the address syntax. Define a domain rather than setting up each table’s constraint individually.

Parameters

name

The name (optionally schema-qualified) of a domain to be created.

data_type

The underlying data type of the domain. This can include array specifiers.

DEFAULT expression

The DEFAULT clause specifies a default value for columns of the domain data type. The value is any variable-free expression (but subqueries are not allowed). The data type of the default expression must match the data type of the domain. If no default value is specified, then the default value is the null value.

The default expression will be used in any insert operation that does not specify a value for the column. If a default value is defined for a particular column, it overrides any default associated with the domain. In turn, the domain default overrides any default value associated with the underlying data type.
CONSTRAINT constraint_name

An optional name for a constraint. If not specified, the system generates a name.

NOT NULL

Values of this domain are not allowed to be null.

NULL

Values of this domain are allowed to be null. This is the default.

This clause is only intended for compatibility with nonstandard SQL databases. Its use is dis-
couraged in new applications.

CHECK (expression)

CHECK clauses specify integrity constraints or tests which values of the domain must satisfy.

Each constraint must be an expression producing a Boolean result. It should use the key word
VALUE to refer to the value being tested.

Currently, CHECK expressions cannot contain subqueries nor refer to variables other than VALUE.

Examples

This example creates the us_postal_code data type and then uses the type in a table definition. A
regular expression test is used to verify that the value looks like a valid US postal code:

CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
   OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (  
   address_id SERIAL PRIMARY KEY,
   street1 TEXT NOT NULL,
   street2 TEXT,
   street3 TEXT,
   city TEXT NOT NULL,
   postal us_postal_code NOT NULL
);

Compatibility

The command CREATE DOMAIN conforms to the SQL standard.

See Also

ALTER DOMAIN, DROP DOMAIN
CREATE FUNCTION

Name
CREATE FUNCTION — define a new function

Synopsis
CREATE [ OR REPLACE ] FUNCTION
  name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
  [ RETURNS rettype ]
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
  [ WITH ( attribute [, ...] ) ]

Description
CREATE FUNCTION defines a new function. CREATE OR REPLACE FUNCTION will either create a
new function, or replace an existing definition.

If a schema name is included, then the function is created in the specified schema. Otherwise it is
created in the current schema. The name of the new function must not match any existing function
with the same argument types in the same schema. However, functions of different argument types
can share a name (this is called overloading).

To replace the current definition of an existing function, use CREATE OR REPLACE FUNCTION. It is
not possible to change the name or argument types of a function this way (if you tried, you would
actually be creating a new, distinct function). Also, CREATE OR REPLACE FUNCTION will not let you
change the return type of an existing function. To do that, you must drop and recreate the function.
(When using OUT parameters, that means you cannot change the names or types of any OUT parameters
except by dropping the function.)

If you drop and then recreate a function, the new function is not the same entity as the old; you
will have to drop existing rules, views, triggers, etc. that refer to the old function. Use CREATE OR
REPLACE FUNCTION to change a function definition without breaking objects that refer to the func-
tion. Also, ALTER FUNCTION can be used to change most of the auxiliary properties of an existing
function.

The user that creates the function becomes the owner of the function.
CREATE FUNCTION

Parameters

name
The name (optionally schema-qualified) of the function to create.

argmode
The mode of an argument: either IN, OUT, or INOUT. If omitted, the default is IN.

argname
The name of an argument. Some languages (currently only PL/pgSQL) let you use the name in
the function body. For other languages the name of an input argument is just extra documentation.
But the name of an output argument is significant, since it defines the column name in the result row type. (If you omit the name for an output argument, the system will choose a default column name.)

argtype
The data type(s) of the function’s arguments (optionally schema-qualified), if any. The argument
types can be base, composite, or domain types, or can reference the type of a table column.
Depending on the implementation language it might also be allowed to specify “pseudotypes”
such as cstring. Pseudotypes indicate that the actual argument type is either incompletely
specified, or outside the set of ordinary SQL data types.
The type of a column is referenced by writing tablename.columnname%TYPE. Using this feature
can sometimes help make a function independent of changes to the definition of a table.

rettype
The return data type (optionally schema-qualified). The return type can be a base, composite,
or domain type, or can reference the type of a table column. Depending on the implementation
language it might also be allowed to specify “pseudotypes” such as cstring. If the function is
not supposed to return a value, specify void as the return type.
When there are OUT or INOUT parameters, the RETURNS clause can be omitted. If present, it must
agree with the result type implied by the output parameters: RECORD if there are multiple output
parameters, or the same type as the single output parameter.
The SELOPT modifier indicates that the function will return a set of items, rather than a single
item.
The type of a column is referenced by writing tablename.columnname%TYPE.

langname
The name of the language that the function is implemented in. Can be SQL, C, internal, or
the name of a user-defined procedural language. For backward compatibility, the name can be
enclosed by single quotes.

IMMUTABLE
STABLE
VOLATILE

These attributes inform the query optimizer about the behavior of the function. At most one
choice can be specified. If none of these appear, VOLATILE is the default assumption.
IMMUTABLE indicates that the function cannot modify the database and always returns the same
result when given the same argument values; that is, it does not do database lookups or otherwise
use information not directly present in its argument list. If this option is given, any call of the
function with all-constant arguments can be immediately replaced with the function value.
CREATE FUNCTION

STABLE indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for functions whose results depend on database lookups, parameter variables (such as the current time zone), etc. Also note that the current_timestamp family of functions qualify as stable, since their values do not change within a transaction.

VOLATILE indicates that the function value can change even within a single table scan, so no optimizations can be made. Relatively few database functions are volatile in this sense; some examples are random(), currval(), timeofday(). But note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away; an example is setval().

For additional details see Section 34.6.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT (the default) indicates that the function will be called normally when some of its arguments are null. It is then the function author’s responsibility to check for null values if necessary and respond appropriately.

RETURNS NULL ON NULL INPUT or STRICT indicates that the function always returns null whenever any of its arguments are null. If this parameter is specified, the function is not executed when there are null arguments; instead a null result is assumed automatically.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER indicates that the function is to be executed with the privileges of the user that calls it. That is the default. SECURITY DEFINER specifies that the function is to be executed with the privileges of the user that created it.

The key word EXTERNAL is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not only external ones.

execution_cost

A positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. If the cost is not specified, 1 unit is assumed for C-language and internal functions, and 100 units for functions in all other languages. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

result_rows

A positive number giving the estimated number of rows that the planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 rows.

configuration_parameter value

The SET clause causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. SET FROM CURRENT saves the session’s current value of the parameter as the value to be applied when the function is entered.

See SET and Chapter 18 for more information about allowed parameter names and values.
**CREATE FUNCTION**

**definition**

A string constant defining the function; the meaning depends on the language. It can be an internal function name, the path to an object file, an SQL command, or text in a procedural language.

**obj_file, link_symbol**

This form of the `AS` clause is used for dynamically loadable C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string `obj_file` is the name of the file containing the dynamically loadable object, and `link_symbol` is the function’s link symbol, that is, the name of the function in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL function being defined.

**attribute**

The historical way to specify optional pieces of information about the function. The following attributes can appear here:

- **isStrict**
  Equivalent to `STRICT` or `RETURNS NULL ON NULL INPUT`.

- **isCachable**
  `isCachable` is an obsolete equivalent of `IMMUTABLE`; it’s still accepted for backwards-compatibility reasons.

Attribute names are not case-sensitive.

**Notes**

Refer to Section 34.3 for further information on writing functions.

The full SQL type syntax is allowed for input arguments and return value. However, some details of the type specification (e.g., the precision field for type `numeric`) are the responsibility of the underlying function implementation and are silently swallowed (i.e., not recognized or enforced) by the `CREATE FUNCTION` command.

PostgreSQL allows function overloading; that is, the same name can be used for several different functions so long as they have distinct argument types. However, the C names of all functions must be different, so you must give overloaded C functions different C names (for example, use the argument types as part of the C names).

Two functions are considered the same if they have the same names and input argument types, ignoring any `OUT` parameters. Thus for example these declarations conflict:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

When repeated `CREATE FUNCTION` calls refer to the same object file, the file is only loaded once per session. To unload and reload the file (perhaps during development), start a new session.

Use `DROP FUNCTION` to remove user-defined functions.

It is often helpful to use dollar quoting (see Section 4.1.2.2) to write the function definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function definition must be escaped by doubling them.
CREATE FUNCTION

If a SET clause is attached to a function, then the effects of a SET LOCAL command executed inside the function for the same variable are restricted to the function: the configuration parameter’s prior value is still restored at function exit. However, an ordinary SET command (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL command: the effects of such a command will persist after function exit, unless the current transaction is rolled back.

To be able to define a function, the user must have the USAGE privilege on the language.

When CREATE OR REPLACE FUNCTION is used to replace an existing function, the ownership and permissions of the function do not change. All other function properties are assigned the values specified or implied in the command. You must own the function to replace it (this includes being a member of the owning role).

Examples

Here are some trivial examples to help you get started. For more information and examples, see Section 34.3.

CREATE FUNCTION add(integer, integer) RETURNS integer
AS ‘select $1 + $2;’
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;

Increment an integer, making use of an argument name, in PL/pgSQL:

CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
BEGIN
  RETURN i + 1;
END;
$$ LANGUAGE plpgsql;

Return a record containing multiple output parameters:

CREATE FUNCTION dup(in int, out f1 int, out f2 text)
AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
LANGUAGE SQL;

SELECT * FROM dup(42);

You can do the same thing more verbosely with an explicitly named composite type:

CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
LANGUAGE SQL;

SELECT * FROM dup(42);
Writing **SECURITY DEFINER** Functions Safely

Because a **SECURITY DEFINER** function is executed with the privileges of the user that created it, care is needed to ensure that the function cannot be misused. For security, `search_path` should be set to exclude any schemas writable by untrusted users. This prevents malicious users from creating objects that mask objects used by the function. Particularly important in this regard is the temporary-table schema, which is searched first by default, and is normally writable by anyone. A secure arrangement can be had by forcing the temporary schema to be searched last. To do this, write `pg_temp` as the last entry in `search_path`. This function illustrates safe usage:

```sql
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
    SELECT (pwd = $2) INTO passed
    FROM pwds
    WHERE username = $1;

    RETURN passed;
END;
$$ LANGUAGE plpgsql
SECURITY DEFINER
-- Set a secure search_path: trusted schema(s), then 'pg_temp'.
SET search_path = admin, pg_temp;
```

Before PostgreSQL version 8.3, the `SET` option was not available, and so older functions may contain rather complicated logic to save, set, and restore `search_path`. The `SET` option is far easier to use for this purpose.

Another point to keep in mind is that by default, execute privilege is granted to `PUBLIC` for newly created functions (see `GRANT` for more information). Frequently you will wish to restrict use of a security definer function to only some users. To do that, you must revoke the default `PUBLIC` privileges and then grant execute privilege selectively. To avoid having a window where the new function is accessible to all, create it and set the privileges within a single transaction. For example:

```sql
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

**Compatibility**

A `CREATE FUNCTION` command is defined in SQL:1999 and later. The PostgreSQL version is similar but not fully compatible. The attributes are not portable, neither are the different available languages.

For compatibility with some other database systems, `argmode` can be written either before or after `argname`. But only the first way is standard-compliant.

**See Also**

`ALTER FUNCTION`, `DROP FUNCTION`, `GRANT`, `LOAD`, `REVOKE`, `createlang`
CREATE GROUP

Name
CREATE GROUP — define a new database role

Synopsis
CREATE GROUP name [ [ WITH ] option [ ... ] ]

where option can be:

- SUPERUSER | NOSUPERUSER
- CREATEDB | NOCREATEDB
- CREATEROLE | NOCREATEROLE
- CREATEUSER | NOCREATEUSER
- INHERIT | NOINHERIT
- LOGIN | NOLOGIN
- [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
- VALID UNTIL 'timestamp'
- IN ROLE rolename [, ...]
- IN GROUP rolename [, ...]
- ROLE rolename [, ...]
- ADMIN rolename [, ...]
- USER rolename [, ...]
- SYSID uid

Description
CREATE GROUP is now an alias for CREATE ROLE.

Compatibility
There is no CREATE GROUP statement in the SQL standard.

See Also
CREATE ROLE
CREATE INDEX

Name
CREATE INDEX — define a new index

Synopsis
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] name ON table [ USING method ]
( { column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [,]
[ WITH ( storage_parameter = value [, ... ] ) ]
[ TABLESPACE tablespace ]
[ WHERE predicate ]

Description
CREATE INDEX constructs an index name on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolour indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on upper(col) would allow the clause WHERE upper(col) = 'JIM' to use an index.

PostgreSQL provides the index methods B-tree, hash, GiST, and GIN. Users can also define their own index methods, but that is fairly complicated.

When the WHERE clause is present, a partial index is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use WHERE with UNIQUE to enforce uniqueness over a subset of a table. See Section 11.8 for more discussion.

The expression used in the WHERE clause can refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subqueries and aggregate expressions are also forbidden in WHERE. The same restrictions apply to index fields that are expressions.

All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or WHERE clause, remember to mark the function immutable when you create it.
CREATE INDEX

Parameters

**UNIQUE**

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

**CONCURRENTLY**

When this option is used, PostgreSQL will build the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table; whereas a standard index build locks out writes (but not reads) on the table until it’s done. There are several caveats to be aware of when using this option — see Building Indexes Concurrently.

**name**

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

**table**

The name (possibly schema-qualified) of the table to be indexed.

**method**

The name of the index method to be used. Choices are `btree`, `hash`, `gist`, and `gin`. The default method is `btree`.

**column**

The name of a column of the table.

**expression**

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses can be omitted if the expression has the form of a function call.

**opclass**

The name of an operator class. See below for details.

**ASC**

Specifies ascending sort order (which is the default).

**DESC**

Specifies descending sort order.

**NULLS FIRST**

Specifies that nulls sort before non-nulls. This is the default when `DESC` is specified.

**NULLS LAST**

Specifies that nulls sort after non-nulls. This is the default when `DESC` is not specified.

**storage_parameter**

The name of an index-method-specific storage parameter. See below for details.

**tablespace**

The tablespace in which to create the index. If not specified, default_tablespace is consulted, or temp_tablespaces for indexes on temporary tables.
**CREATE INDEX**

`predicate`

The constraint expression for a partial index.

**Index Storage Parameters**

The **WITH** clause can specify **storage parameters** for indexes. Each index method can have its own set of allowed storage parameters. The built-in index methods all accept a single parameter:

**FILLFACTOR**

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index’s efficiency. B-trees use a default fillfactor of 90, but any value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index’s physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

**Building Indexes Concurrently**

Creating an index can interfere with regular operation of a database. Normally PostgreSQL locks the table to be indexed against writes and performs the entire index build with a single scan of the table. Other transactions can still read the table, but if they try to insert, update, or delete rows in the table they will block until the index build is finished. This could have a severe effect if the system is a live production database. Very large tables can take many hours to be indexed, and even for smaller tables, an index build can lock out writers for periods that are unacceptably long for a production system.

PostgreSQL supports building indexes without locking out writes. This method is invoked by specifying the **CONCURRENTLY** option of **CREATE INDEX**. When this option is used, PostgreSQL must perform two scans of the table, and in addition it must wait for all existing transactions that could potentially use the index to terminate. Thus this method requires more total work than a standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is built, this method is useful for adding new indexes in a production environment. Of course, the extra CPU and I/O load imposed by the index creation might slow other operations.

In a concurrent index build, the index is actually entered into the system catalogs in one transaction, then the two table scans occur in a second and third transaction. If a problem arises while scanning the table, such as a uniqueness violation in a unique index, the **CREATE INDEX** command will fail but leave behind an “invalid” index. This index will be ignored for querying purposes because it might be incomplete; however it will still consume update overhead. The `psql \d` command will mark such an index as **INVALID**:

```
postgres=# \d tab
       Table "public.tab"
 Column | Type    | Modifiers
--------+---------+--------------------------
        col | integer |--------------------------
Indexes:
          
"idx" btree (col) INVALID
```
The recommended recovery method in such cases is to drop the index and try again to perform `CREATE INDEX CONCURRENTLY`. (Another possibility is to rebuild the index with `REINDEX`. However, since `REINDEX` does not support concurrent builds, this option is unlikely to seem attractive.)

Another caveat when building a unique index concurrently is that the uniqueness constraint is already being enforced against other transactions when the second table scan begins. This means that constraint violations could be reported in other queries prior to the index becoming available for use, or even in cases where the index build eventually fails. Also, if a failure does occur in the second scan, the “invalid” index continues to enforce its uniqueness constraint afterwards.

Concurrent builds of expression indexes and partial indexes are supported. Errors occurring in the evaluation of these expressions could cause behavior similar to that described above for unique constraint violations.

Regular index builds permit other regular index builds on the same table to occur in parallel, but only one concurrent index build can occur on a table at a time. In both cases, no other types of schema modification on the table are allowed meanwhile. Another difference is that a regular `CREATE INDEX` command can be performed within a transaction block, but `CREATE INDEX CONCURRENTLY` cannot.

**Notes**

See Chapter 11 for information about when indexes can be used, when they are not used, and in which particular situations they can be useful.

Currently, only the B-tree and GiST index methods support multicolumn indexes. Up to 32 fields can be specified by default. (This limit can be altered when building PostgreSQL.) Only B-tree currently supports unique indexes.

An **operator class** can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column’s data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. More information about operator classes is in Section 11.9 and in Section 34.14.

For index methods that support ordered scans (currently, only B-tree), the optional clauses `ASC`, `DESC`, `NULLS FIRST`, and/or `NULLS LAST` can be specified to reverse the normal sort direction of the index. Since an ordered index can be scanned either forward or backward, it is not normally useful to create a single-column `DESC` index — that sort ordering is already available with a regular index. The value of these options is that multicolumn indexes can be created that match the sort ordering requested by a mixed-ordering query, such as `SELECT ... ORDER BY x ASC, y DESC`. The `NULLS` options are useful if you need to support “nulls sort low” behavior, rather than the default “nulls sort high”, in queries that depend on indexes to avoid sorting steps.

Use `DROP INDEX` to remove an index.

Prior releases of PostgreSQL also had an R-tree index method. This method has been removed because it had no significant advantages over the GiST method. If `USING rtree` is specified, `CREATE INDEX` will interpret it as `USING gist`, to simplify conversion of old databases to GiST.
Examples

To create a B-tree index on the column `title` in the table `films`:

```sql
CREATE UNIQUE INDEX title_idx ON films (title);
```

To create an index on the expression `lower(title)`, allowing efficient case-insensitive searches:

```sql
CREATE INDEX lower_title_idx ON films ((lower(title)));
```

Caution

Hash index operations are not presently WAL-logged, so hash indexes might need to be rebuilt with `REINDEX` after a database crash if there were unwritten changes. Also, changes to hash indexes are not replicated over warm standby replication after the initial base backup, so they give wrong answers to queries that subsequently use them. For these reasons, hash index use is presently discouraged.

To create an index with non-default sort ordering of nulls:

```sql
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

To create an index with non-default fill factor:

```sql
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

To create an index on the column `code` in the table `films` and have the index reside in the tablespace `indexspace`:

```sql
CREATE INDEX code_idx ON films(code) TABLESPACE indexspace;
```

To create an index without locking out writes to the table:

```sql
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

Compatibility

`CREATE INDEX` is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

See Also

`ALTER INDEX`, `DROP INDEX`
CREATE LANGUAGE

Name
CREATE LANGUAGE — define a new procedural language

Synopsis
CREATE [ PROCEDURAL ] LANGUAGE name
CREATE [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ VALIDATOR valfunction ]

Description
Using CREATE LANGUAGE, a PostgreSQL user can register a new procedural language with a PostgreSQL database. Subsequently, functions and trigger procedures can be defined in this new language.

CREATE LANGUAGE effectively associates the language name with a call handler that is responsible for executing functions written in the language. Refer to Chapter 37 for more information about language call handlers.

There are two forms of the CREATE LANGUAGE command. In the first form, the user supplies just the name of the desired language, and the PostgreSQL server consults the pg_pltemplate system catalog to determine the correct parameters. In the second form, the user supplies the language parameters along with the language name. The second form can be used to create a language that is not defined in pg_pltemplate, but this approach is considered obsolescent.

When the server finds an entry in the pg_pltemplate catalog for the given language name, it will use the catalog data even if the command includes language parameters. This behavior simplifies loading of old dump files, which are likely to contain out-of-date information about language support functions.

Ordinarily, the user must have the PostgreSQL superuser privilege to register a new language. However, the owner of a database can register a new language within that database if the language is listed in the pg_pltemplate catalog and is marked as allowed to be created by database owners (tmpldbacreate is true). The default is that trusted languages can be created by database owners, but this can be adjusted by superusers by modifying the contents of pg_pltemplate. The creator of a language becomes its owner and can later drop it, rename it, or assign it to a new owner.

Parameters

TRUSTED

TRUSTED specifies that the call handler for the language is safe, that is, it does not offer an unprivileged user any functionality to bypass access restrictions. If this key word is omitted when registering the language, only users with the PostgreSQL superuser privilege can use this language to create new functions.

PROCEDURAL

This is a noise word.
CREATE LANGUAGE

name

The name of the new procedural language. The language name is case insensitive. The name
must be unique among the languages in the database.

For backward compatibility, the name can be enclosed by single quotes.

HANDLER call_handler

call_handler is the name of a previously registered function that will be called to execute the
procedural language functions. The call handler for a procedural language must be written in a
compiled language such as C with version 1 call convention and registered with PostgreSQL as
a function taking no arguments and returning the language_handler type, a placeholder type
that is simply used to identify the function as a call handler.

VALIDATOR valfunction

valfunction is the name of a previously registered function that will be called when a new
function in the language is created, to validate the new function. If no validator function is spec-
ified, then a new function will not be checked when it is created. The validator function must
take one argument of type oid, which will be the OID of the to-be-created function, and will
typically return void.

A validator function would typically inspect the function body for syntactical correctness, but
it can also look at other properties of the function, for example if the language cannot handle
certain argument types. To signal an error, the validator function should use the ereport() function. The return value of the function is ignored.

The TRUSTED option and the support function name(s) are ignored if the server has an entry for the
specified language name in pg_pltemplate.

Notes

The createlang program is a simple wrapper around the CREATE LANGUAGE command. It eases in-
stallation of procedural languages from the shell command line.

Use DROP LANGUAGE, or better yet the droplang program, to drop procedural languages.

The system catalog pg_language (see Section 44.21) records information about the currently in-
stalled languages. Also, createlang has an option to list the installed languages.

To create functions in a procedural language, a user must have the USAGE privilege for the language.
By default, USAGE is granted to PUBLIC (i.e., everyone) for trusted languages. This can be revoked if
desired.

Procedural languages are local to individual databases. However, a language can be installed into the
template1 database, which will cause it to be available automatically in all subsequently-created
databases.

The call handler function and the validator function (if any) must already exist if the server does not
have an entry for the language in pg_pltemplate. But when there is an entry, the functions need not
already exist; they will be automatically defined if not present in the database. (This might result in
CREATE LANGUAGE failing, if the shared library that implements the language is not available in the
installation.)

In PostgreSQL versions before 7.3, it was necessary to declare handler functions as returning the
placeholder type opaque, rather than language_handler. To support loading of old dump files,
CREATE LANGUAGE will accept a function declared as returning opaque, but it will issue a notice
and change the function’s declared return type to language_handler.
**Examples**

The preferred way of creating any of the standard procedural languages is just:

```sql
CREATE LANGUAGE plpgsql;
```

For a language not known in the `pg_pltemplate` catalog, a sequence such as this is needed:

```sql
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

**Compatibility**

`CREATE LANGUAGE` is a **PostgreSQL** extension.

**See Also**

`ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE, createlang, droplang`
CREATE OPERATOR

Name
CREATE OPERATOR — define a new operator

Synopsis
CREATE OPERATOR name {
PROCEDURE = funcname
[, LEFTARG = lefttype ] [, RIGHTARG = righttype ]
[, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
[, RESTRICT = res_proc ] [, JOIN = join_proc ]
[, HASHES ] [, MERGES ]
}

Description
CREATE OPERATOR defines a new operator, name. The user who defines an operator becomes its owner. If a schema name is given then the operator is created in the specified schema. Otherwise it is created in the current schema.

The operator name is a sequence of up to NAMEDATALEN-1 (63 by default) characters from the following list:
+ - * / < > = ~ ! @ # % ^ & | ' ?

There are a few restrictions on your choice of name:

• -- and /* cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
• A multicharacter operator name cannot end in + or -, unless the name also contains at least one of these characters:
  ~ ! @ # % ^ & | ' ?

  For example, @- is an allowed operator name, but -* is not. This restriction allows PostgreSQL to parse SQL-compliant commands without requiring spaces between tokens.

The operator != is mapped to <> on input, so these two names are always equivalent.

At least one of LEFTARG and RIGHTARG must be defined. For binary operators, both must be defined. For right unary operators, only LEFTARG should be defined, while for left unary operators only RIGHTARG should be defined.

The funcname procedure must have been previously defined using CREATE FUNCTION and must be defined to accept the correct number of arguments (either one or two) of the indicated types.

The other clauses specify optional operator optimization clauses. Their meaning is detailed in Section 34.13.
CREATE OPERATOR

Parameters

name
The name of the operator to be defined. See above for allowable characters. The name can be schema-qualified, for example CREATE OPERATOR myschema.+ (...). If not, then the operator is created in the current schema. Two operators in the same schema can have the same name if they operate on different data types. This is called overloading.

funcname
The function used to implement this operator.

lefttype
The data type of the operator’s left operand, if any. This option would be omitted for a left-unary operator.

righttype
The data type of the operator’s right operand, if any. This option would be omitted for a right-unary operator.

com_op
The commutator of this operator.

neg_op
The negator of this operator.

res_proc
The restriction selectivity estimator function for this operator.

join_proc
The join selectivity estimator function for this operator.

HASHES
Indicates this operator can support a hash join.

MERGES
Indicates this operator can support a merge join.

To give a schema-qualified operator name in com_op or the other optional arguments, use the OPERATOR() syntax, for example:

COMMUTATOR = OPERATOR(myschema.===) ,

Notes
Refer to Section 34.12 for further information.

The obsolete options SORT1, SORT2, LTCMP, and GTCMP were formerly used to specify the names of sort operators associated with a merge-joinable operator. This is no longer necessary, since information about associated operators is found by looking at B-tree operator families instead. If one of these options is given, it is ignored except for implicitly setting MERGES true.

Use DROP OPERATOR to delete user-defined operators from a database. Use ALTER OPERATOR to modify operators in a database.
CREATE OPERATOR

Examples

The following command defines a new operator, area-equality, for the data type box:

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES, MERGES
);
```

Compatibility

CREATE OPERATOR is a PostgreSQL extension. There are no provisions for user-defined operators in the SQL standard.

See Also

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR
CREATE OPERATOR CLASS

Name

CREATE OPERATOR CLASS — define a new operator class

Synopsis

CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  { OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ RECHECK ]
   | FUNCTION support_number [ ( op_type [, op_type ] ) ] funcname { argument_type [, ... ] }
   | STORAGE storage_type
 } [, ... ]

Description

CREATE OPERATOR CLASS creates a new operator class. An operator class defines how a particular data type can be used with an index. The operator class specifies that certain operators will fill particular roles or “strategies” for this data type and this index method. The operator class also specifies the support procedures to be used by the index method when the operator class is selected for an index column. All the operators and functions used by an operator class must be defined before the operator class can be created.

If a schema name is given then the operator class is created in the specified schema. Otherwise it is created in the current schema. Two operator classes in the same schema can have the same name only if they are for different index methods.

The user who defines an operator class becomes its owner. Presently, the creating user must be a superuser. (This restriction is made because an erroneous operator class definition could confuse or even crash the server.)

CREATE OPERATOR CLASS does not presently check whether the operator class definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user’s responsibility to define a valid operator class.

Related operator classes can be grouped into operator families. To add a new operator class to an existing family, specify the FAMILY option in CREATE OPERATOR CLASS. Without this option, the new class is placed into a family named the same as the new class (creating that family if it doesn’t already exist).

Refer to Section 34.14 for further information.

Parameters

name

The name of the operator class to be created. The name can be schema-qualified.

DEFAULT

If present, the operator class will become the default operator class for its data type. At most one operator class can be the default for a specific data type and index method.
CREATE OPERATOR CLASS

data_type
   The column data type that this operator class is for.

index_method
   The name of the index method this operator class is for.

family_name
   The name of the existing operator family to add this operator class to. If not specified, a family
   named the same as the operator class is used (creating it, if it doesn’t already exist).

strategy_number
   The index method’s strategy number for an operator associated with the operator class.

operator_name
   The name (optionally schema-qualified) of an operator associated with the operator class.

op_type
   In an OPERATOR clause, the operand data type(s) of the operator, or NONE to signify a left-unary
   or right-unary operator. The operand data types can be omitted in the normal case where they are
   the same as the operator class’s data type.

   In a FUNCTION clause, the operand data type(s) the function is intended to support, if different
   from the input data type(s) of the function (for B-tree and hash indexes) or the class’s data type
   (for GIN and GiST indexes). These defaults are always correct, so there is no point in specifying
   op_type in a FUNCTION clause in CREATE OPERATOR CLASS, but the option is provided for
   consistency with the comparable syntax in ALTER OPERATOR FAMILY.

RECHECK
   If present, the index is “lossy” for this operator, and so the rows retrieved using the index must
   be rechecked to verify that they actually satisfy the qualification clause involving this operator.

support_number
   The index method’s support procedure number for a function associated with the operator class.

funcname
   The name (optionally schema-qualified) of a function that is an index method support procedure
   for the operator class.

argument_types
   The parameter data type(s) of the function.

storage_type
   The data type actually stored in the index. Normally this is the same as the column data type,
   but some index methods (currently GIN and GiST) allow it to be different. The storage clause
   must be omitted unless the index method allows a different type to be used.

   The OPERATOR, FUNCTION, and STORAGE clauses can appear in any order.

Notes

Because the index machinery does not check access permissions on functions before using them, in-
cluding a function or operator in an operator class is tantamount to granting public execute permission
on it. This is usually not an issue for the sorts of functions that are useful in an operator class.
The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

Examples

The following example command defines a GiST index operator class for the data type \texttt{\_int4} (array of \texttt{int4}). See \texttt{contrib/intarray/} for the complete example.

```sql
CREATE OPERATOR CLASS gist\_int\_ops
DEFAULT FOR TYPE \_int4 USING gist AS
   OPERATOR 3 \&\&,
   OPERATOR 6 = RECHECK,
   OPERATOR 7 @>,
   OPERATOR 8 <@,
   OPERATOR 20 @> (_int4, query\_int),
   FUNCTION 1 g\_int\_consistent (internal, \_int4, int4),
   FUNCTION 2 g\_int\_union (bytea, internal),
   FUNCTION 3 g\_int\_compress (internal),
   FUNCTION 4 g\_int\_decompress (internal),
   FUNCTION 5 g\_int\_penalty (internal, internal, internal),
   FUNCTION 6 g\_int\_picksplit (internal, internal),
   FUNCTION 7 g\_int\_same (_int4, \_int4, internal);
```

Compatibility

\texttt{CREATE OPERATOR CLASS} is a PostgreSQL extension. There is no \texttt{CREATE OPERATOR CLASS} statement in the SQL standard.

See Also

\texttt{ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY}
CREATE OPERATOR FAMILY

Name

CREATE OPERATOR FAMILY — define a new operator family

Synopsis

CREATE OPERATOR FAMILY name USING index_method

Description

CREATE OPERATOR FAMILY creates a new operator family. An operator family defines a collection of related operator classes, and perhaps some additional operators and support functions that are compatible with these operator classes but not essential for the functioning of any individual index. (Operators and functions that are essential to indexes should be grouped within the relevant operator class, rather than being “loose” in the operator family. Typically, single-data-type operators are bound to operator classes, while cross-data-type operators can be loose in an operator family containing operator classes for both data types.)

The new operator family is initially empty. It should be populated by issuing subsequent CREATE OPERATOR CLASS commands to add contained operator classes, and optionally ALTER OPERATOR FAMILY commands to add “loose” operators and their corresponding support functions.

If a schema name is given then the operator family is created in the specified schema. Otherwise it is created in the current schema. Two operator families in the same schema can have the same name only if they are for different index methods.

The user who defines an operator family becomes its owner. Presently, the creating user must be a superuser. (This restriction is made because an erroneous operator family definition could confuse or even crash the server.)

Refer to Section 34.14 for further information.

Parameters

name

The name of the operator family to be created. The name can be schema-qualified.

index_method

The name of the index method this operator family is for.

Compatibility

CREATE OPERATOR FAMILY is a PostgreSQL extension. There is no CREATE OPERATOR FAMILY statement in the SQL standard.
See Also

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER
OPERATOR CLASS, DROP OPERATOR CLASS
CREATE ROLE

Name
CREATE ROLE — define a new database role

Synopsis
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

- SUPERUSER | NOSUPERUSER
- CREATEDB | NOCREATEDB
- CREATEROLE | NOCREATEROLE
- CREATEUSER | NOCREATEUSER
- INHERIT | NOINHERIT
- LOGIN | NOLOGIN
- CONNECTION LIMIT connlimit
- [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
- VALID UNTIL 'timestamp'
- IN ROLE rolename [, ...]
- IN GROUP rolename [, ...]
- ROLE rolename [, ...]
- ADMIN rolename [, ...]
- USER rolename [, ...]
- SYSID uid

Description
CREATE ROLE adds a new role to a PostgreSQL database cluster. A role is an entity that can own
database objects and have database privileges; a role can be considered a “user”, a “group”, or both
depending on how it is used. Refer to Chapter 19 and Chapter 21 for information about managing
users and authentication. You must have CREATEROLE privilege or be a database superuser to use this
command.

Note that roles are defined at the database cluster level, and so are valid in all databases in the cluster.

Parameters

name
The name of the new role.

SUPERUSER
NOSUPERUSER

These clauses determine whether the new role is a “superuser”, who can override all access
restrictions within the database. Superuser status is dangerous and should be used only when
really needed. You must yourself be a superuser to create a new superuser. If not specified,
NOSUPERUSER is the default.
CREATE ROLE
CREATEDB
NOCREATEDDB
These clauses define a role’s ability to create databases. If CREATEDB is specified, the role being defined will be allowed to create new databases. Specifying NOCREATEDB will deny a role the ability to create databases. If not specified, NOCREATEDB is the default.
CREATEROLE
NOCREATEROLE
These clauses determine whether a role will be permitted to create new roles (that is, execute CREATE ROLE). A role with CREATEROLE privilege can also alter and drop other roles. If not specified, NOCREATEROLE is the default.
CREATEUSER
NOCREATEUSER
These clauses are an obsolete, but still accepted, spelling of SUPERUSER and NOSUPERUSER. Note that they are not equivalent to CREATEROLE as one might naively expect!
INHERIT
NOINHERIT
These clauses determine whether a role “inherits” the privileges of roles it is a member of. A role with the INHERIT attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. Without INHERIT, membership in another role only grants the ability to SET ROLE to that other role; the privileges of the other role are only available after having done so. If not specified, INHERIT is the default.
LOGIN
NOLOGIN
These clauses determine whether a role is allowed to log in; that is, whether the role can be given as the initial session authorization name during client connection. A role having the LOGIN attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges, but are not users in the usual sense of the word. If not specified, NOLOGIN is the default, except when CREATE ROLE is invoked through its alternative spelling CREATE USER.
CONNECTION LIMIT connlimit
If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.
PASSWORD password
Sets the role’s password. (A password is only of use for roles having the LOGIN attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as PASSWORD NULL.
ENCRYPTED
UNENCRYPTED
These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter password_encryption.) If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether ENCRYPTED or UNENCRYPTED is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.
Note that older clients might lack support for the MD5 authentication mechanism that is needed to work with passwords that are stored encrypted.

**VALID UNTIL 'timestamp'**

The **VALID UNTIL** clause sets a date and time after which the role's password is no longer valid. If this clause is omitted the password will be valid for all time.

**IN ROLE rolename**

The **IN ROLE** clause lists one or more existing roles to which the new role will be immediately added as a new member. (Note that there is no option to add the new role as an administrator; use a separate **GRANT** command to do that.)

**IN GROUP rolename**

**IN GROUP** is an obsolete spelling of **IN ROLE**.

**ROLE rolename**

The **ROLE** clause lists one or more existing roles which are automatically added as members of the new role. (This in effect makes the new role a "group".)

**ADMIN rolename**

The **ADMIN** clause is like **ROLE**, but the named roles are added to the new role with **ADMIN OPTION**, giving them the right to grant membership in this role to others.

**USER rolename**

The **USER** clause is an obsolete spelling of the **ROLE** clause.

**SYSID uid**

The **SYSID** clause is ignored, but is accepted for backwards compatibility.

**Notes**

Use **ALTER ROLE** to change the attributes of a role, and **DROP ROLE** to remove a role. All the attributes specified by **CREATE ROLE** can be modified by later **ALTER ROLE** commands.

The preferred way to add and remove members of roles that are being used as groups is to use **GRANT** and **REVOKE**.

The **VALID UNTIL** clause defines an expiration time for a password only, not for the role **per se**. In particular, the expiration time is not enforced when logging in using a non-password-based authentication method.

The **INHERIT** attribute governs inheritance of grantable privileges (that is, access privileges for database objects and role memberships). It does not apply to the special role attributes set by **CREATE ROLE** and **ALTER ROLE**. For example, being a member of a role with **CREATEDB** privilege does not immediately grant the ability to create databases, even if **INHERIT** is set; it would be necessary to become that role via **SET ROLE** before creating a database.

The **INHERIT** attribute is the default for reasons of backwards compatibility: in prior releases of PostgreSQL, users always had access to all privileges of groups they were members of. However, **NOINHERIT** provides a closer match to the semantics specified in the SQL standard.

Be careful with the **CREATEROLE** privilege. There is no concept of inheritance for the privileges of a **CREATEROLE**-role. That means that even if a role does not have a certain privilege but is allowed to create other roles, it can easily create another role with different privileges than its own (except for creating roles with superuser privileges). For example, if the role "user" has the **CREATEROLE**
privilege but not the `CREATEDB` privilege, nonetheless it can create a new role with the `CREATEDB`
privilege. Therefore, regard roles that have the `CREATEROLE` privilege as almost-superuser-roles.

PostgreSQL includes a program `createuser` that has the same functionality as `CREATE ROLE` (in fact, it
calls this command) but can be run from the command shell.

The `CONNECTION LIMIT` option is only enforced approximately; if two new sessions start at about
the same time when just one connection “slot” remains for the role, it is possible that both will fail.
Also, the limit is never enforced for superusers.

Caution must be exercised when specifying an unencrypted password with this command. The pass-
word will be transmitted to the server in cleartext, and it might also be logged in the client’s command
history or the server log. The command `createuser`, however, transmits the password encrypted. Also,
`psql` contains a command `\password` that can be used to safely change the password later.

### Examples

Create a role that can log in, but don’t give it a password:

```sql
CREATE ROLE jonathan LOGIN;
```

Create a role with a password:

```sql
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

( `CREATE USER` is the same as `CREATE ROLE` except that it implies `LOGIN`.)

Create a role with a password that is valid until the end of 2004. After one second has ticked in 2005, the password is no longer valid.

```sql
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Create a role that can create databases and manage roles:

```sql
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

### Compatibility

The `CREATE ROLE` statement is in the SQL standard, but the standard only requires the syntax

```sql
CREATE ROLE name [ WITH ADMIN rolename ]
```

Multiple initial administrators, and all the other options of `CREATE ROLE`, are PostgreSQL extensions.

The SQL standard defines the concepts of users and roles, but it regards them as distinct concepts and leaves all commands defining users to be specified by each database implementation. In PostgreSQL we have chosen to unify users and roles into a single kind of entity. Roles therefore have many more optional attributes than they do in the standard.

The behavior specified by the SQL standard is most closely approximated by giving users the `NOINHERIT` attribute, while roles are given the `INHERIT` attribute.
See Also

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser
CREATE RULE

Name
CREATE RULE — define a new rewrite rule

Synopsis
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }

Description
CREATE RULE defines a new rule applying to a specified table or view. CREATE OR REPLACE RULE will either create a new rule, or replace an existing rule of the same name for the same table.

The PostgreSQL rule system allows one to define an alternative action to be performed on insertions, updates, or deletions in database tables. Roughly speaking, a rule causes additional commands to be executed when a given command on a given table is executed. Alternatively, an INSTEAD rule can replace a given command by another, or cause a command not to be executed at all. Rules are used to implement table views as well. It is important to realize that a rule is really a command transformation mechanism, or command macro. The transformation happens before the execution of the commands starts. If you actually want an operation that fires independently for each physical row, you probably want to use a trigger, not a rule. More information about the rules system is in Chapter 36.

Presently, ON SELECT rules must be unconditional INSTEAD rules and must have actions that consist of a single SELECT command. Thus, an ON SELECT rule effectively turns the table into a view, whose visible contents are the rows returned by the rule’s SELECT command rather than whatever had been stored in the table (if anything). It is considered better style to write a CREATE VIEW command than to create a real table and define an ON SELECT rule for it.

You can create the illusion of an updatable view by defining ON INSERT, ON UPDATE, and ON DELETE rules (or any subset of those that’s sufficient for your purposes) to replace update actions on the view with appropriate updates on other tables. If you want to support INSERT RETURNING and so on, then be sure to put a suitable RETURNING clause into each of these rules.

There is a catch if you try to use conditional rules for view updates: there must be an unconditional INSTEAD rule for each action you wish to allow on the view. If the rule is conditional, or is not INSTEAD, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table of the view in some cases. If you want to handle all the useful cases in conditional rules, add an unconditional DO INSTEAD NOTHING rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules non-INSTEAD; in the cases where they are applied, they add to the default INSTEAD NOTHING action. (This method does not currently work to support RETURNING queries, however.)
CREATE RULE

Parameters

name

The name of a rule to create. This must be distinct from the name of any other rule for the same table. Multiple rules on the same table and same event type are applied in alphabetical name order.

event

The event is one of SELECT, INSERT, UPDATE, or DELETE.

table

The name (optionally schema-qualified) of the table or view the rule applies to.

condition

Any SQL conditional expression (returning boolean). The condition expression cannot refer to any tables except NEW and OLD, and cannot contain aggregate functions.

INSTEAD

INSTEAD indicates that the commands should be executed instead of the original command.

ALSO

ALSO indicates that the commands should be executed in addition to the original command.

If neither ALSO nor INSTEAD is specified, ALSO is the default.

command

The command or commands that make up the rule action. Valid commands are SELECT, INSERT, UPDATE, DELETE, or NOTIFY.

Within condition and command, the special table names NEW and OLD can be used to refer to values in the referenced table. NEW is valid in ON INSERT and ON UPDATE rules to refer to the new row being inserted or updated. OLD is valid in ON UPDATE and ON DELETE rules to refer to the existing row being updated or deleted.

Notes

You must be the owner of a table to create or change rules for it.

In a rule for INSERT, UPDATE, or DELETE on a view, you can add a RETURNING clause that emits the view’s columns. This clause will be used to compute the outputs if the rule is triggered by an INSERT RETURNING, UPDATE RETURNING, or DELETE RETURNING command respectively. When the rule is triggered by a command without RETURNING, the rule’s RETURNING clause will be ignored. The current implementation allows only unconditional INSTEAD rules to contain RETURNING; furthermore there can be at most one RETURNING clause among all the rules for the same event. (This ensures that there is only one candidate RETURNING clause to be used to compute the results.) RETURNING queries on the view will be rejected if there is no RETURNING clause in any available rule.

It is very important to take care to avoid circular rules. For example, though each of the following two rule definitions are accepted by PostgreSQL, the SELECT command would cause PostgreSQL to report an error because of recursive expansion of a rule:

CREATE RULE ".RETURN" AS
   ON SELECT TO t1
   DO INSTEAD
      SELECT * FROM t2;
CREATE RULE "_RETURN" AS
  ON SELECT TO t2
  DO INSTEAD
    SELECT * FROM t1;

SELECT * FROM t1;

Presently, if a rule action contains a NOTIFY command, the NOTIFY command will be executed unconditionally, that is, the NOTIFY will be issued even if there are not any rows that the rule should apply to. For example, in:

CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;

one NOTIFY event will be sent during the UPDATE, whether or not there are any rows that match the condition id = 42. This is an implementation restriction that might be fixed in future releases.

Compatibility

CREATE RULE is a PostgreSQL language extension, as is the entire query rewrite system.
CREATE SCHEMA

Name
CREATE SCHEMA — define a new schema

Synopsis
CREATE SCHEMA schemaname [ AUTHORIZATION username ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION username [ schema_element [ ... ] ]

Description
CREATE SCHEMA enters a new schema into the current database. The schema name must be distinct from the name of any existing schema in the current database.

A schema is essentially a namespace: it contains named objects (tables, data types, functions, and operators) whose names can duplicate those of other objects existing in other schemas. Named objects are accessed either by “qualifying” their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). A CREATE command specifying an unqualified object name creates the object in the current schema (the one at the front of the search path, which can be determined with the function current_schema).

Optionally, CREATE SCHEMA can include subcommands to create objects within the new schema. The subcommands are treated essentially the same as separate commands issued after creating the schema, except that if the AUTHORIZATION clause is used, all the created objects will be owned by that user.

Parameters

schemaname
The name of a schema to be created. If this is omitted, the username is used as the schema name. The name cannot begin with pg_, as such names are reserved for system schemas.

username
The role name of the user who will own the new schema. If omitted, defaults to the user executing the command. To create a schema owned by another role, you must be a direct or indirect member of that role, or be a superuser.

schema_element
An SQL statement defining an object to be created within the schema. Currently, only CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE SEQUENCE, CREATE TRIGGER and GRANT are accepted as clauses within CREATE SCHEMA. Other kinds of objects may be created in separate commands after the schema is created.

Notes
To create a schema, the invoking user must have the CREATE privilege for the current database. (Of course, superusers bypass this check.)
CREATE SCHEMA

Examples

Create a schema:

CREATE SCHEMA myschema;

Create a schema for user joe; the schema will also be named joe:

CREATE SCHEMA AUTHORIZATION joe;

Create a schema and create a table and view within it:

CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;

Notice that the individual subcommands do not end with semicolons.

The following is an equivalent way of accomplishing the same result:

CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;

Compatibility

The SQL standard allows a DEFAULT CHARACTER SET clause in CREATE SCHEMA, as well as more subcommand types than are presently accepted by PostgreSQL.

The SQL standard specifies that the subcommands in CREATE SCHEMA can appear in any order. The present PostgreSQL implementation does not handle all cases of forward references in subcommands; it might sometimes be necessary to reorder the subcommands in order to avoid forward references.

According to the SQL standard, the owner of a schema always owns all objects within it. PostgreSQL allows schemas to contain objects owned by users other than the schema owner. This can happen only if the schema owner grants the CREATE privilege on his schema to someone else, or a superuser chooses to create objects in it.

See Also

ALTER SCHEMA, DROP SCHEMA
CREATE SEQUENCE

Name
CREATE SEQUENCE — define a new sequence generator

Synopsis
CREATE [ TEMPORARY | TEMP | SEQUENCE name [ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
[ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
[ OWNED BY { table.column | NONE } ]

Description
CREATE SEQUENCE creates a new sequence number generator. This involves creating and initializing a new special single-row table with the name name. The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name cannot be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, you use the functions nextval, currval, and setval to operate on the sequence. These functions are documented in Section 9.15.

Although you cannot update a sequence directly, you can use a query like:
SELECT * FROM name;

to examine the parameters and current state of a sequence. In particular, the last_value field of the sequence shows the last value allocated by any session. (Of course, this value might be obsolete by the time it’s printed, if other sessions are actively doing nextval calls.)

Parameters

TEMPORARY or TEMP
If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.

name
The name (optionally schema-qualified) of the sequence to be created.

increment
The optional clause INCREMENT BY increment specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.
CREATE SEQUENCE

minvalue
NO MINVALUE

The optional clause MINVALUE minvalue determines the minimum value a sequence can generate. If this clause is not supplied or NO MINVALUE is specified, then defaults will be used. The defaults are 1 and \(-2^{61}-1\) for ascending and descending sequences, respectively.

maxvalue
NO MAXVALUE

The optional clause MAXVALUE maxvalue determines the maximum value for the sequence. If this clause is not supplied or NO MAXVALUE is specified, then default values will be used. The defaults are \(2^{61}-1\) and -1 for ascending and descending sequences, respectively.

start

The optional clause START WITH start allows the sequence to begin anywhere. The default starting value is minvalue for ascending sequences and maxvalue for descending ones.

cache

The optional clause CACHE cache specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache), and this is also the default.

CYCLE
NO CYCLE

The CYCLE option allows the sequence to wrap around when the maxvalue or minvalue has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the minvalue or maxvalue, respectively.

If NO CYCLE is specified, any calls to nextval after the sequence has reached its maximum value will return an error. If neither CYCLE or NO CYCLE are specified, NO CYCLE is the default.

OWNED BY table.column
OWNED BY NONE

The OWNED BY option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. The specified table must have the same owner and be in the same schema as the sequence. OWNED BY NONE, the default, specifies that there is no such association.

Notes

Use DROP SEQUENCE to remove a sequence.

Sequences are based on bigint arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some older platforms, there might be no compiler support for eight-byte integers, in which case sequences use regular integer arithmetic (range -2147483648 to +2147483647).

Unexpected results might be obtained if a cache setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Each session will allocate and cache successive sequence values during one access to the sequence object and increase the sequence object’s last_value accordingly. Then, the next cache-1 uses of nextval within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in “holes” in the sequence.
Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values might be generated out of sequence when all the sessions are considered. For example, with a cache setting of 10, session A might reserve values 1..10 and return nextval=1, then session B might reserve values 11..20 and return nextval=11 before session A has generated nextval=2. Thus, with a cache setting of one it is safe to assume that nextval values are generated sequentially; with a cache setting greater than one you should only assume that the nextval values are all distinct, not that they are generated purely sequentially. Also, last_value will reflect the latest value reserved by any session, whether or not it has yet been returned by nextval.

Another consideration is that a setval executed on such a sequence will not be noticed by other sessions until they have used up any preallocated values they have cached.

**Examples**

Create an ascending sequence called serial, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence:

```
SELECT nextval('serial');
```

```
nextval
--------
101
```

Select the next number from this sequence:

```
SELECT nextval('serial');
```

```
nextval
--------
102
```

Use this sequence in an INSERT command:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Update the sequence value after a COPY FROM:

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```
Compatibility

CREATE SEQUENCE conforms to the SQL standard, with the following exceptions:

- The standard’s AS <data type> expression is not supported.
- Obtaining the next value is done using the nextval() function instead of the standard’s NEXT VALUE FOR expression.
- The OWNED BY clause is a PostgreSQL extension.

See Also

ALTER SEQUENCE, DROP SEQUENCE
CREATE TABLE

Name
CREATE TABLE — define a new table

Synopsis
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name ( [ column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ] ] ]
| table_constraint
| LIKE parent_table [ { INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES } ]
| [ , ... ]
)
| INHERITS ( parent_table [ , ... ] )
| WITH ( storage_parameter [= value] [ , ... ] ) | WITH OIDS | WITHOUT OIDS
| ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP }
| TABLESPACE tablespace

where column_constraint is:
[ CONSTRAINT constraint_name ]
| NOT NULL |
| NULL |
| UNIQUE index_parameters |
| PRIMARY KEY index_parameters |
| CHECK ( expression ) |
| REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
| [ ON DELETE action ] [ ON UPDATE action ]
| DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint is:
[ CONSTRAINT constraint_name ]
| UNIQUE ( column_name [, ... ] ) index_parameters |
| PRIMARY KEY ( column_name [, ... ] ) index_parameters |
| CHECK ( expression ) |
| FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
| MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ]
| DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

index_parameters in UNIQUE and PRIMARY KEY constraints are:
[ WITH ( storage_parameter [= value] [ , ... ] ) ]
[ USING INDEX TABLESPACE tablespace ]

Description
CREATE TABLE will create a new, initially empty table in the current database. The table will be owned by the user issuing the command. If a schema name is given (for example, CREATE TABLE myschema.mytable ...) then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables
exist in a special schema, so a schema name cannot be given when creating a temporary table. The name of the table must be distinct from the name of any other table, sequence, index, or view in the same schema.

`CREATE TABLE` also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

The optional constraint clauses specify constraints (tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience for use when the constraint only affects one column.

### Parameters

**TEMPORARY** or **TEMP**

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see **ON COMMIT** below). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

Optionally, **GLOBAL** or **LOCAL** can be written before **TEMPORARY** or **TEMP**. This makes no difference in PostgreSQL, but see **Compatibility**.

**table_name**

The name (optionally schema-qualified) of the table to be created.

**column_name**

The name of a column to be created in the new table.

**data_type**

The data type of the column. This can include array specifiers. For more information on the data types supported by PostgreSQL, refer to Chapter 8.

**DEFAULT** *default_expr*

The **DEFAULT** clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

**INHERITS** ( **parent_table** [, ... ] )

The optional **INHERITS** clause specifies a list of tables from which the new table automatically inherits all columns.
Use of **INHERITS** creates a persistent relationship between the new child table and its parent table(s). Schema modifications to the parent(s) normally propagate to children as well, and by default the data of the child table is included in scans of the parent(s).

If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match in each of the parent tables. If there is no conflict, then the duplicate columns are merged to form a single column in the new table. If the column name list of the new table contains a column name that is also inherited, the data type must likewise match the inherited column(s), and the column definitions are merged into one. However, inherited and new column declarations of the same name need not specify identical constraints: all constraints provided from any declaration are merged together and all are applied to the new table. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations of the column. Otherwise, any parents that specify default values for the column must all specify the same default, or an error will be reported.

**LIKE**<br>

```
LIKE parent_table [ { INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES } ]
```

The **LIKE** clause specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints.

Unlike **INHERITS**, the new table and original table are completely decoupled after creation is complete. Changes to the original table will not be applied to the new table, and it is not possible to include data of the new table in scans of the original table.

Default expressions for the copied column definitions will only be copied if **INCLUDING DEFAULTS** is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having null defaults.

Not-null constraints are always copied to the new table. **CHECK** constraints will only be copied if **INCLUDING CONSTRAINTS** is specified; other types of constraints will never be copied. Also, no distinction is made between column constraints and table constraints — when constraints are requested, all check constraints are copied.

Any indexes on the original table will not be created on the new table, unless the **INCLUDING INDEXES** clause is specified.

Note also that unlike **INHERITS**, copied columns and constraints are not merged with similarly named columns and constraints. If the same name is specified explicitly or in another **LIKE** clause, an error is signalled.

**CONSTRAINT**<br>

```
CONSTRAINT constraint_name
```

An optional name for a column or table constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like `col must be positive` can be used to communicate helpful constraint information to client applications. (Double-quotes are needed to specify constraint names that contain spaces.) If a constraint name is not specified, the system generates a name.

**NOT NULL**

The column is not allowed to contain null values.

**NULL**

The column is allowed to contain null values. This is the default.

This clause is only provided for compatibility with non-standard SQL databases. Its use is discouraged in new applications.
CREATE TABLE

UNIQUE (column constraint)
UNIQUE ( column_name [ , ... ] ) (table constraint)

The UNIQUE constraint specifies that a group of one or more columns of a table can contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

For the purpose of a unique constraint, null values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)

PRIMARY KEY (column constraint)
PRIMARY KEY ( column_name [ , ... ] ) (table constraint)

The primary key constraint specifies that a column or columns of a table can contain only unique (non-duplicate), nonnull values. Technically, PRIMARY KEY is merely a combination of UNIQUE and NOT NULL, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables can rely on this set of columns as a unique identifier for rows.

Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table.

CHECK ( expression )

The CHECK clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or UNKNOWN succeed. Should any row of an insert or update operation produce a FALSE result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column’s value only, while an expression appearing in a table constraint can reference multiple columns.

Currently, CHECK expressions cannot contain subqueries nor refer to variables other than columns of the current row.

REFERENCES reftable [ ( refcolumn ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ] (column constraint)
FOREIGN KEY ( column [ , ... ] ) REFERENCES reftable [ ( refcolumn [ , ... ] ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ] (table constraint)

These clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If refcolumn is omitted, the primary key of the reftable is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table. Note that foreign key constraints cannot be defined between temporary tables and permanent tables.

A value inserted into the referencing column(s) is matched against the values of the referenced table and referenced columns using the given match type. There are three match types: MATCH FULL, MATCH PARTIAL, and MATCH SIMPLE, which is also the default. MATCH FULL will not allow one column of a multicolumn foreign key to be null unless all foreign key columns are null. MATCH SIMPLE allows some foreign key columns to be null while other parts of the foreign key are not null. MATCH PARTIAL is not yet implemented.
In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table’s columns. The ON DELETE clause specifies the action to perform when a referenced row in the referenced table is being deleted. Likewise, the ON UPDATE clause specifies the action to perform when a referenced column in the referenced table is being updated to a new value. If the row is updated, but the referenced column is not actually changed, no action is done. Referential actions other than the NO ACTION check cannot be deferred, even if the constraint is declared deferrable. There are the following possible actions for each clause:

NO ACTION

Produce an error indicating that the deletion or update would create a foreign key constraint violation. If the constraint is deferred, this error will be produced at constraint check time if there still exist any referencing rows. This is the default action.

RESTRICT

Produce an error indicating that the deletion or update would create a foreign key constraint violation. This is the same as NO ACTION except that the check is not deferrable.

CASCADE

Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column, respectively.

SET NULL

Set the referencing column(s) to null.

SET DEFAULT

Set the referencing column(s) to their default values.

If the referenced column(s) are changed frequently, it might be wise to add an index to the foreign key column so that referential actions associated with the foreign key column can be performed more efficiently.

DEFERRABLE

NOT DEFERRABLE

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction (using the SET CONSTRAINTS command). NOT DEFERRABLE is the default. Only foreign key constraints currently accept this clause. All other constraint types are not deferrable.

INITIALLY IMMEDIATE

INITIALLY DEFERRED

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is INITIAL IMMEDIATE, it is checked after each statement. This is the default. If the constraint is INITIALLY DEFERRED, it is checked only at the end of the transaction. The constraint check time can be altered with the SET CONSTRAINTS command.

WITH ( storage_parameter [= value] [, ... ] )

This clause specifies optional storage parameters for a table or index; see Storage Parameters for more information. The WITH clause for a table can also include OIDS=TRUE (or just OIDS) to specify that rows of the new table should have OIDs (object identifiers) assigned to them, or OIDS=FALSE to specify that the rows should not have OIDs. If OIDS is not specified, the default setting depends upon the default_with_oids configuration parameter. (If the new table
CREATE TABLE

inherits from any tables that have OIDs, then OIDS=TRUE is forced even if the command says OIDS=FALSE.)

If OIDS=FALSE is specified or implied, the new table does not store OIDs and no OID will be assigned for a row inserted into it. This is generally considered worthwhile, since it will reduce OID consumption and thereby postpone the wraparound of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which makes them considerably less useful. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row (on most machines), slightly improving performance.

To remove OIDs from a table after it has been created, use ALTER TABLE.

WITH OIDS
WITHOUT OIDS

These are obsolescent syntaxes equivalent to WITH {OIDS} and WITH {OIDS=FALSE}, respectively. If you wish to give both an OIDS setting and storage parameters, you must use the WITH { ... } syntax; see above.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using ON COMMIT. The three options are:

PRESERVE ROWS

No special action is taken at the ends of transactions. This is the default behavior.

DELETE ROWS

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic TRUNCATE is done at each commit.

DROP

The temporary table will be dropped at the end of the current transaction block.

TABLESPACE tablespace

The tablespace is the name of the tablespace in which the new table is to be created. If not specified, default_tablespace is consulted, or temp_tablespaces if the table is temporary.

USING INDEX TABLESPACE tablespace

This clause allows selection of the tablespace in which the index associated with a UNIQUE or PRIMARY KEY constraint will be created. If not specified, default_tablespace is consulted, or temp_tablespaces if the table is temporary.

Storage Parameters

The WITH clause can specify storage parameters for tables, and for indexes associated with a UNIQUE or PRIMARY KEY constraint. Storage parameters for indexes are documented in CREATE INDEX. The only storage parameter currently available for tables is:

FILLFACTOR

The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, INSERT operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives UPDATE a chance to place the updated copy of a row on the same page as the
original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate.

**Notes**

Using OIDs in new applications is not recommended: where possible, using a `SERIAL` or other sequence generator as the table’s primary key is preferred. However, if your application does make use of OIDs to identify specific rows of a table, it is recommended to create a unique constraint on the `oid` column of that table, to ensure that OIDs in the table will indeed uniquely identify rows even after counter wraparound. Avoid assuming that OIDs are unique across tables; if you need a database-wide unique identifier, use the combination of `tableoid` and row OID for the purpose.

**Tip:** The use of `OIDS=FALSE` is not recommended for tables with no primary key, since without either an OID or a unique data key, it is difficult to identify specific rows.

PostgreSQL automatically creates an index for each unique constraint and primary key constraint to enforce uniqueness. Thus, it is not necessary to create an index explicitly for primary key columns. (See `CREATE INDEX` for more information.)

Unique constraints and primary keys are not inherited in the current implementation. This makes the combination of inheritance and unique constraints rather dysfunctional.

A table cannot have more than 1600 columns. (In practice, the effective limit is usually lower because of tuple-length constraints.)

**Examples**

Create table `films` and table `distributors`:

```sql
CREATE TABLE films (  
  code varchar(5) CONSTRAINT firstkey PRIMARY KEY,  
  title varchar(40) NOT NULL,  
  did integer NOT NULL,  
  date_prod date,  
  kind varchar(10),  
  len interval hour to minute  
);

CREATE TABLE distributors (  
  did integer PRIMARY KEY DEFAULT nextval('serial'),  
  name varchar(40) NOT NULL CHECK (name <> '')  
);
```

Create a table with a 2-dimensional array:

```sql
CREATE TABLE array_int (  
  vector int[][]  
);
```
Define a unique table constraint for the table `films`. Unique table constraints can be defined on one or more columns of the table:

```sql
CREATE TABLE films (  
  code char(5),  
  title varchar(40),  
  did integer,  
  date Prod date,  
  kind varchar(10),  
  len interval hour to minute,  
  CONSTRAINT production UNIQUE(date Prod)
);  
```

Define a check column constraint:

```sql
CREATE TABLE distributors (  
  did integer CHECK (did > 100),  
  name varchar(40)
);  
```

Define a check table constraint:

```sql
CREATE TABLE distributors (  
  did integer,  
  name varchar(40)  
  CONSTRAINT con1 CHECK (did > 100 AND name <> "")
);  
```

Define a primary key table constraint for the table `films`:

```sql
CREATE TABLE films (  
  code char(5),  
  title varchar(40),  
  did integer,  
  date Prod date,  
  kind varchar(10),  
  len interval hour to minute,  
  CONSTRAINT code_title PRIMARY KEY(code, title)
);  
```

Define a primary key constraint for table `distributors`. The following two examples are equivalent, the first using the table constraint syntax, the second the column constraint syntax:

```sql
CREATE TABLE distributors (  
  did integer,  
  name varchar(40),  
  PRIMARY KEY(did)
);  
```

```sql
CREATE TABLE distributors (  
  did integer,  
  name varchar(40),
  PRIMARY KEY(did)
);  
```
CREATE TABLE distributors (  
    name varchar(40) DEFAULT 'Luso Films',  
    did integer DEFAULT nextval('distributors_serial'),  
    modtime timestamp DEFAULT current_timestamp  
);  

Define two NOT NULL column constraints on the table distributors, one of which is explicitly given a name:  
CREATE TABLE distributors (  
    did integer CONSTRAINT no_null NOT NULL,  
    name varchar(40) NOT NULL  
);  

Define a unique constraint for the name column:  
CREATE TABLE distributors (  
    did integer,  
    name varchar(40) UNIQUE  
);  

The same, specified as a table constraint:  
CREATE TABLE distributors (  
    did integer,  
    name varchar(40),  
    UNIQUE(name)  
);  

Create the same table, specifying 70% fill factor for both the table and its unique index:  
CREATE TABLE distributors (  
    did integer,  
    name varchar(40),  
    UNIQUE(name) WITH (fillfactor=70)  
)  
WITH (fillfactor=70);  

Create table cinemas in tablespace diskvol1:  
CREATE TABLE cinemas (  
    id serial,
CREATE TABLE
  name text,
  location text
) TABLESPACE diskvol1;

Compatibility

The CREATE TABLE command conforms to the SQL standard, with exceptions listed below.

Temporary Tables

Although the syntax of CREATE TEMPORARY TABLE resembles that of the SQL standard, the effect is not the same. In the standard, temporary tables are defined just once and automatically exist (starting with empty contents) in every session that needs them. PostgreSQL instead requires each session to issue its own CREATE TEMPORARY TABLE command for each temporary table to be used. This allows different sessions to use the same temporary table name for different purposes, whereas the standard’s approach constrains all instances of a given temporary table name to have the same table structure.

The standard’s definition of the behavior of temporary tables is widely ignored. PostgreSQL’s behavior on this point is similar to that of several other SQL databases.

The standard’s distinction between global and local temporary tables is not in PostgreSQL, since that distinction depends on the concept of modules, which PostgreSQL does not have. For compatibility’s sake, PostgreSQL will accept the GLOBAL and LOCAL keywords in a temporary table declaration, but they have no effect.

The ON COMMIT clause for temporary tables also resembles the SQL standard, but has some differences. If the ON COMMIT clause is omitted, SQL specifies that the default behavior is ON COMMIT DELETE ROWS. However, the default behavior in PostgreSQL is ON COMMIT PRESERVE ROWS. The ON COMMIT DROP option does not exist in SQL.

Column Check Constraints

The SQL standard says that CHECK column constraints can only refer to the column they apply to; only CHECK table constraints can refer to multiple columns. PostgreSQL does not enforce this restriction; it treats column and table check constraints alike.

NULL “Constraint”

The NULL “constraint” (actually a non-constraint) is a PostgreSQL extension to the SQL standard that is included for compatibility with some other database systems (and for symmetry with the NOT NULL constraint). Since it is the default for any column, its presence is simply noise.

Inheritance

Multiple inheritance via the INHERITS clause is a PostgreSQL language extension. SQL:1999 and later define single inheritance using a different syntax and different semantics. SQL:1999-style inheritance is not yet supported by PostgreSQL.
Zero-column tables

PostgreSQL allows a table of no columns to be created (for example, `CREATE TABLE foo();`). This is an extension from the SQL standard, which does not allow zero-column tables. Zero-column tables are not in themselves very useful, but disallowing them creates odd special cases for `ALTER TABLE DROP COLUMN`, so it seems cleaner to ignore this spec restriction.

WITH clause

The `WITH` clause is a PostgreSQL extension; neither storage parameters nor OIDs are in the standard.

Tablespaces

The PostgreSQL concept of tablespaces is not part of the standard. Hence, the clauses `TABLESPACE` and `USING INDEX TABLESPACE` are extensions.

See Also

`ALTER TABLE`, `DROP TABLE`, `CREATE TABLESPACE`
CREATE TABLE AS

Name
CREATE TABLE AS — define a new table from the results of a query

Synopsis
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
    [ (column_name [, ...] ) ]
    [ WITH (storage_parameter [= value] [, ... ]) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace ]
AS query

Description
CREATE TABLE AS creates a table and fills it with data computed by a SELECT command. The table columns have the names and data types associated with the output columns of the SELECT (except that you can override the column names by giving an explicit list of new column names).

CREATE TABLE AS bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates its defining SELECT statement whenever it is queried.

Parameters
GLOBAL or LOCAL
Ignored for compatibility. Refer to CREATE TABLE for details.

TEMPORARY or TEMP
If specified, the table is created as a temporary table. Refer to CREATE TABLE for details.

table_name
The name (optionally schema-qualified) of the table to be created.

column_name
The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query. If the table is created from an EXECUTE command, a column name list cannot be specified.

WITH (storage_parameter [= value] [, ... ])
This clause specifies optional storage parameters for the new table; see Storage Parameters for more information. The WITH clause can also include OIDS=TRUE (or just OIDS) to specify that rows of the new table should have OIDs (object identifiers) assigned to them, or OIDS=FALSE to specify that the rows should not have OIDs. See CREATE TABLE for more information.
CREATE TABLE AS

WITH OIDS
WITHOUT OIDS

These are obsolescent syntaxes equivalent to WITH (OIDS) and WITH (OIDS=FALSE), respectively. If you wish to give both an OIDS setting and storage parameters, you must use the WITH (...) syntax; see above.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using ON COMMIT. The three options are:

PRETERVE ROWS

No special action is taken at the ends of transactions. This is the default behavior.

DELETE ROWS

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic TRUNCATE is done at each commit.

DROP

The temporary table will be dropped at the end of the current transaction block.

TABLESPACE tablespace

The tablespace is the name of the tablespace in which the new table is to be created. If not specified, default_tablespace is consulted, or temp_tablespaces if the table is temporary.

query

A SELECT or VALUES command, or an EXECUTE command that runs a prepared SELECT or VALUES query.

Notes

This command is functionally similar to SELECT INTO, but it is preferred since it is less likely to be confused with other uses of the SELECT INTO syntax. Furthermore, CREATE TABLE AS offers a superset of the functionality offered by SELECT INTO.

Prior to PostgreSQL 8.0, CREATE TABLE AS always included OIDs in the table it created. As of PostgreSQL 8.0, the CREATE TABLE AS command allows the user to explicitly specify whether OIDs should be included. If the presence of OIDs is not explicitly specified, the default_with_oids configuration variable is used. As of PostgreSQL 8.1, this variable is false by default, so the default behavior is not identical to pre-8.0 releases. Applications that require OIDs in the table created by CREATE TABLE AS should explicitly specify WITH (OIDS) to ensure proper behavior.

Examples

Create a new table films_recent consisting of only recent entries from the table films:

CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
CREATE TABLE AS

Create a new temporary table `films_recent`, consisting of only recent entries from the table `films`, using a prepared statement. The new table has OIDs and will be dropped at commit:

```sql
PREPARE recentfilms(date) AS
    SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
    EXECUTE recentfilms('2002-01-01');
```

Compatibility

`CREATE TABLE AS` conforms to the SQL standard, with the following exceptions:

- The standard requires parentheses around the subquery clause; in PostgreSQL, these parentheses are optional.
- The standard defines a `WITH [ NO ] DATA` clause; this is not currently implemented by PostgreSQL. The behavior provided by PostgreSQL is equivalent to the standard’s `WITH DATA` case. `WITH NO DATA` can be simulated by appending `LIMIT 0` to the query.
- PostgreSQL handles temporary tables in a way rather different from the standard; see `CREATE TABLE` for details.
- The `WITH` clause is a PostgreSQL extension; neither storage parameters nor OIDs are in the standard.
- The PostgreSQL concept of tablespaces is not part of the standard. Hence, the clause `TABLESPACE` is an extension.

See Also

`CREATE TABLE`, `EXECUTE`, `SELECT`, `SELECT INTO`, `VALUES`
**CREATE TABLESPACE**

**Name**

CREATE TABLESPACE — define a new tablespace

**Synopsis**

CREATE TABLESPACE `tablespacename` [ OWNER `username` ] LOCATION 'directory'

**Description**

CREATE TABLESPACE registers a new cluster-wide tablespace. The tablespace name must be distinct from the name of any existing tablespace in the database cluster.

A tablespace allows superusers to define an alternative location on the file system where the data files containing database objects (such as tables and indexes) can reside.

A user with appropriate privileges can pass `tablespacename` to CREATE DATABASE, CREATE TABLE, CREATE INDEX or ADD CONSTRAINT to have the data files for these objects stored within the specified tablespace.

**Parameters**

`tablespacename`

The name of a tablespace to be created. The name cannot begin with `pg_`, as such names are reserved for system tablespaces.

`username`

The name of the user who will own the tablespace. If omitted, defaults to the user executing the command. Only superusers can create tablespaces, but they can assign ownership of tablespaces to non-superusers.

`directory`

The directory that will be used for the tablespace. The directory must be empty and must be owned by the PostgreSQL system user. The directory must be specified by an absolute path name.

**Notes**

Tablespaces are only supported on systems that support symbolic links.

CREATE TABLESPACE cannot be executed inside a transaction block.
**Examples**

Create a tablespace `dbspace` at `/data/dbs`:

```sql
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

Create a tablespace `indexspace` at `/data/indexes` owned by user `genevieve`:

```sql
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

**Compatibility**

`CREATE TABLESPACE` is a PostgreSQL extension.

**See Also**

`CREATE DATABASE`, `CREATE TABLE`, `CREATE INDEX`, `DROP TABLESPACE`, `ALTER TABLESPACE`
CREATE TEXT SEARCH CONFIGURATION

Name

CREATE TEXT SEARCH CONFIGURATION — define a new text search configuration

Synopsis

CREATE TEXT SEARCH CONFIGURATION name (  
    PARSER = parser_name |  
    COPY = source_config  
)

Description

CREATE TEXT SEARCH CONFIGURATION creates a new text search configuration. A text search configuration specifies a text search parser that can divide a string into tokens, plus dictionaries that can be used to determine which tokens are of interest for searching.

If only the parser is specified, then the new text search configuration initially has no mappings from token types to dictionaries, and therefore will ignore all words. Subsequent ALTER TEXT SEARCH CONFIGURATION commands must be used to create mappings to make the configuration useful. Alternatively, an existing text search configuration can be copied.

If a schema name is given then the text search configuration is created in the specified schema. Otherwise it is created in the current schema.

The user who defines a text search configuration becomes its owner.

Refer to Chapter 12 for further information.

Parameters

name

The name of the text search configuration to be created. The name can be schema-qualified.

parser_name

The name of the text search parser to use for this configuration.

source_config

The name of an existing text search configuration to copy.

Notes

The PARSER and COPY options are mutually exclusive, because when an existing configuration is copied, its parser selection is copied too.
CREATE TEXT SEARCH CONFIGURATION

Compatibility

There is no CREATE TEXT SEARCH CONFIGURATION statement in the SQL standard.

See Also

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION
CREATE TEXT SEARCH DICTIONARY

Name
CREATE TEXT SEARCH DICTIONARY — define a new text search dictionary

Synopsis
CREATE TEXT SEARCH DICTIONARY name { 
  TEMPLATE = template 
  [, option = value [, ... ]] 
}

Description
CREATE TEXT SEARCH DICTIONARY creates a new text search dictionary. A text search dictionary specifies a way of recognizing interesting or uninteresting words for searching. A dictionary depends on a text search template, which specifies the functions that actually perform the work. Typically the dictionary provides some options that control the detailed behavior of the template’s functions.

If a schema name is given then the text search dictionary is created in the specified schema. Otherwise it is created in the current schema.

The user who defines a text search dictionary becomes its owner.

Refer to Chapter 12 for further information.

Parameters

name
  The name of the text search dictionary to be created. The name can be schema-qualified.

template
  The name of the text search template that will define the basic behavior of this dictionary.

option
  The name of a template-specific option to be set for this dictionary.

value
  The value to use for a template-specific option. If the value is not a simple identifier or number, it must be quoted (but you can always quote it, if you wish).

The options can appear in any order.

Examples

The following example command creates a Snowball-based dictionary with a nonstandard list of stop words.

CREATE TEXT SEARCH DICTIONARY my_russian {
template = snowball,
language = russian,
stopwords = myrussian
);

Compatibility

There is no CREATE TEXT SEARCH DICTIONARY statement in the SQL standard.

See Also

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY
CREATE TEXT SEARCH PARSER

Name

CREATE TEXT SEARCH PARSER — define a new text search parser

Synopsis

CREATE TEXT SEARCH PARSER name ( 
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)

Description

CREATE TEXT SEARCH PARSER creates a new text search parser. A text search parser defines a method for splitting a text string into tokens and assigning types (categories) to the tokens. A parser is not particularly useful by itself, but must be bound into a text search configuration along with some text search dictionaries to be used for searching.

If a schema name is given then the text search parser is created in the specified schema. Otherwise it is created in the current schema.

You must be a superuser to use CREATE TEXT SEARCH PARSER. (This restriction is made because an erroneous text search parser definition could confuse or even crash the server.)

Refer to Chapter 12 for further information.

Parameters

name

The name of the text search parser to be created. The name can be schema-qualified.

start_function

The name of the start function for the parser.

calltoken_function

The name of the get-next-token function for the parser.

call_function

The name of the end function for the parser.

lextypes_function

The name of the lextypes function for the parser (a function that returns information about the set of token types it produces).
**headline_function**

The name of the headline function for the parser (a function that summarizes a set of tokens).

The function names can be schema-qualified if necessary. Argument types are not given, since the argument list for each type of function is predetermined. All except the headline function are required.

The arguments can appear in any order, not only the one shown above.

**Compatibility**

There is no `CREATE TEXT SEARCH PARSER` statement in the SQL standard.

**See Also**

`ALTER TEXT SEARCH PARSER`, `DROP TEXT SEARCH PARSER`
CREATE TEXT SEARCH TEMPLATE

Name

CREATE TEXT SEARCH TEMPLATE — define a new text search template

Synopsis

CREATE TEXT SEARCH TEMPLATE name {
   [ INIT = init_function , ]
   LEXIZE = lexize_function
}

Description

CREATE TEXT SEARCH TEMPLATE creates a new text search template. Text search templates define the functions that implement text search dictionaries. A template is not useful by itself, but must be instantiated as a dictionary to be used. The dictionary typically specifies parameters to be given to the template functions.

If a schema name is given then the text search template is created in the specified schema. Otherwise it is created in the current schema.

You must be a superuser to use CREATE TEXT SEARCH TEMPLATE. This restriction is made because an erroneous text search template definition could confuse or even crash the server. The reason for separating templates from dictionaries is that a template encapsulates the “unsafe” aspects of defining a dictionary. The parameters that can be set when defining a dictionary are safe for unprivileged users to set, and so creating a dictionary need not be a privileged operation.

Refer to Chapter 12 for further information.

Parameters

name

The name of the text search template to be created. The name can be schema-qualified.

init_function

The name of the init function for the template.

lexize_function

The name of the lexize function for the template.

The function names can be schema-qualified if necessary. Argument types are not given, since the argument list for each type of function is predetermined. The lexize function is required, but the init function is optional.

The arguments can appear in any order, not only the one shown above.
Compatibility

There is no `CREATE TEXT SEARCH TEMPLATE` statement in the SQL standard.

See Also

`ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE`
CREATE TRIGGER

Name

CREATE TRIGGER — define a new trigger

Synopsis

CREATE TRIGGER name ( BEFORE | AFTER ) { event [ OR ... ] }
   ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
   EXECUTE PROCEDURE funcname ( arguments )

Description

CREATE TRIGGER creates a new trigger. The trigger will be associated with the specified table and will execute the specified function funcname when certain events occur.

The trigger can be specified to fire either before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE, or DELETE is attempted) or after the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed). If the trigger fires before the event, the trigger can skip the operation for the current row, or change the row being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the last insertion, update, or deletion, are “visible” to the trigger.

A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. For example, a DELETE that affects 10 rows will cause any ON DELETE triggers on the target relation to be called 10 separate times, once for each deleted row. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies (in particular, an operation that modifies zero rows will still result in the execution of any applicable FOR EACH STATEMENT triggers).

If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.

SELECT does not modify any rows so you cannot create SELECT triggers. Rules and views are more appropriate in such cases.

Refer to Chapter 35 for more information about triggers.

Parameters

name

The name to give the new trigger. This must be distinct from the name of any other trigger for the same table.

BEFORE
AFTER

Determines whether the function is called before or after the event.
CREATE TRIGGER

event

One of INSERT, UPDATE, or DELETE; this specifies the event that will fire the trigger. Multiple events can be specified using OR.

table

The name (optionally schema-qualified) of the table the trigger is for.

FOR EACH ROW

FOR EACH STATEMENT

This specifies whether the trigger procedure should be fired once for every row affected by the trigger event, or just once per SQL statement. If neither is specified, FOR EACH STATEMENT is the default.

funcname

A user-supplied function that is declared as taking no arguments and returning type trigger, which is executed when the trigger fires.

arguments

An optional comma-separated list of arguments to be provided to the function when the trigger is executed. The arguments are literal string constants. Simple names and numeric constants can be written here, too, but they will all be converted to strings. Please check the description of the implementation language of the trigger function about how the trigger arguments are accessible within the function; it might be different from normal function arguments.

Notes

To create a trigger on a table, the user must have the TRIGGER privilege on the table. The user must also have EXECUTE privilege on the trigger function.

In PostgreSQL versions before 7.3, it was necessary to declare trigger functions as returning the placeholder type opaque, rather than trigger. To support loading of old dump files, CREATE TRIGGER will accept a function declared as returning opaque, but it will issue a notice and change the function’s declared return type to trigger.

Use DROP TRIGGER to remove a trigger.

Examples

Section 35.4 contains a complete example.

Compatibility

The CREATE TRIGGER statement in PostgreSQL implements a subset of the SQL standard. The following functionality is currently missing:

- SQL allows triggers to fire on updates to specific columns (e.g., AFTER UPDATE OF col1, col2).
- SQL allows you to define aliases for the “old” and “new” rows or tables for use in the definition of the triggered action (e.g., CREATE TRIGGER ... ON tablename REFERENCING OLD ROW
CREATE TRIGGER

AS somename NEW ROW AS othername ...). Since PostgreSQL allows trigger procedures to be written in any number of user-defined languages, access to the data is handled in a language-specific way.

- PostgreSQL only allows the execution of a user-defined function for the triggered action. The standard allows the execution of a number of other SQL commands, such as 
CREATE TABLE
as the triggered action. This limitation is not hard to work around by creating a user-defined function that executes the desired commands.

SQL specifies that multiple triggers should be fired in time-of-creation order. PostgreSQL uses name order, which was judged to be more convenient.

SQL specifies that BEFORE DELETE triggers on cascaded deletes fire after the cascaded DELETE completes. The PostgreSQL behavior is for BEFORE DELETE to always fire before the delete action, even a cascading one. This is considered more consistent. There is also unpredictable behavior when BEFORE triggers modify rows that are later to be modified by referential actions. This can lead to constraint violations or stored data that does not honor the referential constraint.

The ability to specify multiple actions for a single trigger using OR is a PostgreSQL extension of the SQL standard.

See Also

CREATE FUNCTION, ALTER TRIGGER, DROP TRIGGER
CREATE TYPE

Name
CREATE TYPE — define a new data type

Synopsis
CREATE TYPE name AS
  ( attribute_name data_type [, ... ] )

CREATE TYPE name AS ENUM
  ( 'label' [, ... ] )

CREATE TYPE name (
  INPUT = input_function,
  OUTPUT = output_function
  [ , RECEIVE = receive_function ]
  [ , SEND = send_function ]
  [ , TYPMOD_IN = type_mod_input_function ]
  [ , TYPMOD_OUT = type_mod_output_function ]
  [ , ANALYZE = analyze_function ]
  [ , INTERNALLENGTH = { internallength | VARIABLE } ]
  [ , PASSEDBYVALUE ]
  [ , ALIGNMENT = alignment ]
  [ , STORAGE = storage ]
  [ , DEFAULT = default ]
  [ , ELEMENT = element ]
  [ , DELIMITER = delimiter ]
)

CREATE TYPE name

Description
CREATE TYPE registers a new data type for use in the current database. The user who defines a type becomes its owner.

If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. (Because tables have associated data types, the type name must also be distinct from the name of any existing table in the same schema.)

Composite Types
The first form of CREATE TYPE creates a composite type. The composite type is specified by a list of attribute names and data types. This is essentially the same as the row type of a table, but using CREATE TYPE avoids the need to create an actual table when all that is wanted is to define a type. A stand-alone composite type is useful as the argument or return type of a function.
Enumerated Types

The second form of CREATE TYPE creates an enumerated (enum) type, as described in Section 8.7. Enum types take a list of one or more quoted labels, each of which must be less than \texttt{NAMEDATALEN} bytes long (64 in a standard PostgreSQL build).

Base Types

The third form of CREATE TYPE creates a new base type (scalar type). The parameters can appear in any order, not only that illustrated above, and most are optional. You must register two or more functions (using CREATE FUNCTION) before defining the type. The support functions \texttt{input\_function} and \texttt{output\_function} are required, while the functions \texttt{receive\_function}, \texttt{send\_function}, \texttt{type\_modifier\_input\_function}, \texttt{type\_modifier\_output\_function} and \texttt{analyze\_function} are optional. Generally these functions have to be coded in C or another low-level language.

The \texttt{input\_function} converts the type’s external textual representation to the internal representation used by the operators and functions defined for the type. \texttt{output\_function} performs the reverse transformation. The input function can be declared as taking one argument of type \texttt{cstring}, or as taking three arguments of types \texttt{cstring}, \texttt{oid}, \texttt{integer}. The first argument is the input text as a C string, the second argument is the type’s own OID (except for array types, which instead receive their element type’s OID), and the third is the \texttt{typmod} of the destination column, if known (-1 will be passed if not). The input function must return a value of the data type itself. Usually, an input function should be declared STRICT; if it is not, it will be called with a NULL first parameter when reading a NULL input value. The function must still return NULL in this case, unless it raises an error. (This case is mainly meant to support domain input functions, which might need to reject NULL inputs.) The output function must be declared as taking one argument of the new data type. The output function must return type \texttt{cstring}. Output functions are not invoked for NULL values.

The optional \texttt{receive\_function} converts the type’s external binary representation to the internal representation. If this function is not supplied, the type cannot participate in binary input. The binary representation should be chosen to be cheap to convert to internal form, while being reasonably portable. (For example, the standard integer data types use network byte order as the external binary representation, while the internal representation is in the machine’s native byte order.) The receive function should perform adequate checking to ensure that the value is valid. The receive function can be declared as taking one argument of type \texttt{internal}, or as taking three arguments of types \texttt{internal}, \texttt{oid}, \texttt{integer}. The first argument is a pointer to a \texttt{StringInfo} buffer holding the received byte string; the optional arguments are the same as for the text input function. The receive function must return a value of the data type itself. Usually, a receive function should be declared STRICT; if it is not, it will be called with a NULL first parameter when reading a NULL input value. The function must still return NULL in this case, unless it raises an error. (This case is mainly meant to support domain receive functions, which might need to reject NULL inputs.) Similarly, the optional \texttt{send\_function} converts from the internal representation to the external binary representation. If this function is not supplied, the type cannot participate in binary output. The send function must be declared as taking one argument of the new data type. The send function must return type \texttt{bytea}. Send functions are not invoked for NULL values.

You should at this point be wondering how the input and output functions can be declared to have results or arguments of the new type, when they have to be created before the new type can be created. The answer is that the type should first be defined as a \texttt{shell type}, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command \texttt{CREATE TYPE name,} with no additional parameters. Then the I/O functions can be defined referencing the shell type. Finally, \texttt{CREATE TYPE} with a full definition replaces the shell entry with a complete, valid type.
CREATE TYPE

The optional type_modifier_input_function and type_modifier_output_function are needed if the type supports modifiers, that is optional constraints attached to a type declaration, such as char(5) or numeric(30,2). PostgreSQL allows user-defined types to take one or more simple constants or identifiers as modifiers. However, this information must be capable of being packed into a single non-negative integer value for storage in the system catalogs. The type_modifier_input_function is passed the declared modifier(s) in the form of a cstring array. It must check the values for validity (throwing an error if they are wrong), and if they are correct, return a single non-negative integer value that will be stored as the column “typmod”. Type modifiers will be rejected if the type does not have a type_modifier_input_function. The type_modifier_output_function converts the internal integer typmod value back to the correct form for user display. It must return a cstring value that is the exact string to append to the type name; for example numeric’s function might return (30,2). It is allowed to omit the type_modifier_output_function, in which case the default display format is just the stored typmod integer value enclosed in parentheses.

The optional analyze_function performs type-specific statistics collection for columns of the data type. By default, ANALYZE will attempt to gather statistics using the type’s “equals” and “less-than” operators, if there is a default b-tree operator class for the type. For non-scalar types this behavior is likely to be unsuitable, so it can be overridden by specifying a custom analysis function. The analysis function must be declared to take a single argument of type internal, and return a boolean result. The detailed API for analysis functions appears in src/include/commands/vacuum.h.

While the details of the new type’s internal representation are only known to the I/O functions and other functions you create to work with the type, there are several properties of the internal representation that must be declared to PostgreSQL. Foremost of these is internallength. Base data types can be fixed-length, in which case internallength is a positive integer, or variable length, indicated by setting internallength to VARIABLE. (Internally, this is represented by setting typlen to -1.) The internal representation of all variable-length types must start with a 4-byte integer giving the total length of this value of the type.

The optional flag PASSEDBYVALUE indicates that values of this data type are passed by value, rather than by reference. You cannot pass by value types whose internal representation is larger than the size of the Datum type (4 bytes on most machines, 8 bytes on a few).

The alignment parameter specifies the storage alignment required for the data type. The allowed values equate to alignment on 1, 2, 4, or 8 byte boundaries. Note that variable-length types must have an alignment of at least 4, since they necessarily contain an int4 as their first component.

The storage parameter allows selection of storage strategies for variable-length data types. (Only plain is allowed for fixed-length types.) plain specifies that data of the type will always be stored in-line and not compressed. extended specifies that the system will first try to compress a long data value, and will move the value out of the main table row if it’s still too long. external allows the value to be moved out of the main table, but the system will not try to compress it. main allows compression, but discourages moving the value out of the main table. (Data items with this storage strategy might still be moved out of the main table if there is no other way to make a row fit, but they will be kept in the main table preferentially over extended and external items.)

A default value can be specified, in case a user wants columns of the data type to default to something other than the null value. Specify the default with the DEFAULT keyword. (Such a default can be overridden by an explicit DEFAULT clause attached to a particular column.)

To indicate that a type is an array, specify the type of the array elements using the ELEMENT key word. For example, to define an array of 4-byte integers (int4), specify ELEMENT = int4. More details about array types appear below.
CREATE TYPE

To indicate the delimiter to be used between values in the external representation of arrays of this
type, _delimiter_ can be set to a specific character. The default delimiter is the comma (,). Note that
the delimiter is associated with the array element type, not the array type itself.

**Array Types**

Whenever a user-defined type is created, PostgreSQL automatically creates an associated array type,
whose name consists of the base type’s name prepended with an underscore, and truncated if necessary
to keep it less than `_NAMEDATALEN_` bytes long. (If the name so generated collides with an existing type
name, the process is repeated until a non-colliding name is found.) This implicitly-created array type
is variable length and uses the built-in input and output functions _array_in_ and _array_out_. The
array type tracks any changes in its element type’s owner or schema, and is dropped if the element
type is.

You might reasonably ask why there is an _ELEMENT_ option, if the system makes the correct array
type automatically. The only case where it’s useful to use _ELEMENT_ is when you are making a fixed-
length type that happens to be internally an array of a number of identical things, and you want to
allow these things to be accessed directly by subcripting, in addition to whatever operations you
plan to provide for the type as a whole. For example, type _point_ is represented as just two floating-
point numbers, which it allows to be accessed as _point[0]_ and _point[1]_. Note that this facility
only works for fixed-length types whose internal form is exactly a sequence of identical fixed-length
fields. A subscriptable variable-length type must have the generalized internal representation used
by _array_in_ and _array_out_. For historical reasons (i.e., this is clearly wrong but it’s far too late
to change it), subscripting of fixed-length array types starts from zero, rather than from one as for
variable-length arrays.

**Parameters**

- **name**
  The name (optionally schema-qualified) of a type to be created.

- **attribute_name**
  The name of an attribute (column) for the composite type.

- **data_type**
  The name of an existing data type to become a column of the composite type.

- **label**
  A string literal representing the textual label associated with one value of an enum type.

- **input_function**
  The name of a function that converts data from the type’s external textual form to its internal
  form.

- **output_function**
  The name of a function that converts data from the type’s internal form to its external textual
  form.
**CREATE TYPE**

receive_function

The name of a function that converts data from the type’s external binary form to its internal form.

send_function

The name of a function that converts data from the type’s internal form to its external binary form.

type_modifier_input_function

The name of a function that converts an array of modifier(s) for the type into internal form.

type_modifier_output_function

The name of a function that converts the internal form of the type’s modifier(s) to external textual form.

analyze_function

The name of a function that performs statistical analysis for the data type.

internal_length

A numeric constant that specifies the length in bytes of the new type’s internal representation. The default assumption is that it is variable-length.

alignment

The storage alignment requirement of the data type. If specified, it must be char, int2, int4, or double; the default is int4.

storage

The storage strategy for the data type. If specified, must be plain, external, extended, or main; the default is plain.

default

The default value for the data type. If this is omitted, the default is null.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character to be used between values in arrays made of this type.

**Notes**

Because there are no restrictions on use of a data type once it’s been created, creating a base type is tantamount to granting public execute permission on the functions mentioned in the type definition. (The creator of the type is therefore required to own these functions.) This is usually not an issue for the sorts of functions that are useful in a type definition. But you might want to think twice before designing a type in a way that would require “secret” information to be used while converting it to or from external form.

Before PostgreSQL version 8.3, the name of a generated array type was always exactly the element type’s name with one underscore character (_) prepended. (Type names were therefore restricted in length to one less character than other names.) While this is still usually the case, the array type name may vary from this in case of maximum-length names or collisions with user type names that begin...
with underscore. Writing code that depends on this convention is therefore deprecated. Instead, use `pg_type.typarray` to locate the array type associated with a given type.

It may be advisable to avoid using type and table names that begin with underscore. While the server will change generated array type names to avoid collisions with user-given names, there is still risk of confusion, particularly with old client software that may assume that type names beginning with underscores always represent arrays.

Before PostgreSQL version 8.2, the syntax `CREATE TYPE name` did not exist. The way to create a new base type was to create its input function first. In this approach, PostgreSQL will first see the name of the new data type as the return type of the input function. The shell type is implicitly created in this situation, and then it can be referenced in the definitions of the remaining I/O functions. This approach still works, but is deprecated and might be disallowed in some future release. Also, to avoid accidentally cluttering the catalogs with shell types as a result of simple typos in function definitions, a shell type will only be made this way when the input function is written in C.

In PostgreSQL versions before 7.3, it was customary to avoid creating a shell type at all, by replacing the functions’ forward references to the type name with the placeholder pseudotype `opaque`. The `cstring` arguments and results also had to be declared as `opaque` before 7.3. To support loading of old dump files, `CREATE TYPE` will accept I/O functions declared using `opaque`, but it will issue a notice and change the function declarations to use the correct types.

**Examples**

This example creates a composite type and uses it in a function definition:

```sql
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

This example creates an enumerated type and uses it in a table definition:

```sql
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (id serial,
    description text,
    status bug_status);
```

This example creates the base data type `box` and then uses the type in a table definition:

```sql
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
```
CREATE TYPE

CREATE TABLE myboxes (  
id integer,  
description box
);

If the internal structure of box were an array of four float4 elements, we might instead use:

CREATE TYPE box (  
    INTERNALLENGTH = 16,  
    INPUT = my_box_in_function,  
    OUTPUT = my_box_out_function,  
    ELEMENT = float4
);

which would allow a box value’s component numbers to be accessed by subscripting. Otherwise the type behaves the same as before.

This example creates a large object type and uses it in a table definition:

CREATE TYPE bigobj (  
    INPUT = lo_filein, OUTPUT = lo_fileout,  
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (  
id integer,  
obj bigobj
);

More examples, including suitable input and output functions, are in Section 34.11.

Compatibility

This CREATE TYPE command is a PostgreSQL extension. There is a CREATE TYPE statement in the SQL standard that is rather different in detail.

See Also

CREATE FUNCTION, DROP TYPE, ALTER TYPE, CREATE DOMAIN
CREATE USER

Name

CREATE USER — define a new database role

Synopsis

CREATE USER name [ [ WITH ] option [ ... ] ]

where option can be:

- SUPERUSER | NOSUPERUSER
- CREATEDB | NOCREATEDB
- CREATEROLE | NOCREATEROLE
- CREATEUSER | NOCREATEUSER
- INHERIT | NOINHERIT
- LOGIN | NOLOGIN
- CONNECTION LIMIT conlimit
- [ ENCRIPTED | UNENCRYPTED ] PASSWORD 'password'
- VALID UNTIL 'timestamp'
- IN ROLE rolenum [, ...]
- IN GROUP rolenum [, ...]
- ROLE rolenum [, ...]
- ADMIN rolenum [, ...]
- USER rolenum [, ...]
- SYSID uid

Description

CREATE USER is now an alias for CREATE ROLE. The only difference is that when the command is spelled CREATE USER, LOGIN is assumed by default, whereas NOLOGIN is assumed when the command is spelled CREATE ROLE.

Compatibility

The CREATE USER statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

CREATE ROLE
CREATE VIEW

Name
CREATE VIEW — define a new view

Synopsis
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name [ ( column_name [, ...] ) ]
   AS query

Description
CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, the query
is run every time the view is referenced in a query.

CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced.
You can only replace a view with a new query that generates the identical set of columns (i.e., same
column names and data types).

If a schema name is given (for example, CREATE VIEW myschema.myview ...) then the view is
created in the specified schema. Otherwise it is created in the current schema. Temporary views exist
in a special schema, so a schema name cannot be given when creating a temporary view. The name
of the view must be distinct from the name of any other view, table, sequence, or index in the same
schema.

Parameters
TEMPORARY or TEMP

If specified, the view is created as a temporary view. Temporary views are automatically dropped
at the end of the current session. Existing permanent relations with the same name are not visible
to the current session while the temporary view exists, unless they are referenced with schema-
qualified names.

If any of the tables referenced by the view are temporary, the view is created as a temporary view
(whether TEMPORARY is specified or not).

name

The name (optionally schema-qualified) of a view to be created.

column_name

An optional list of names to be used for columns of the view. If not given, the column names are
deduced from the query.

query

A SELECT or VALUES command which will provide the columns and rows of the view.
CREATE VIEW

Notes

Currently, views are read only: the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables. For more information see CREATE RULE.

Use the DROP VIEW statement to drop views.

Be careful that the names and types of the view’s columns will be assigned the way you want. For example:

CREATE VIEW vista AS SELECT ‘Hello World’;

is bad form in two ways: the column name defaults to ?column?, and the column data type defaults to unknown. If you want a string literal in a view’s result, use something like:

CREATE VIEW vista AS SELECT text ‘Hello World’ AS hello;

Access to tables referenced in the view is determined by permissions of the view owner. In some cases, this can be used to provide secure but restricted access to the underlying tables. However, not all views are secure against tampering; see Section 36.4 for details. Functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call all functions used by the view.

When CREATE OR REPLACE VIEW is used on an existing view, only the view’s defining SELECT rule is changed. Other view properties, including ownership, permissions, and non-SELECT rules, remain unchanged. You must own the view to replace it (this includes being a member of the owning role).

Examples

Create a view consisting of all comedy films:

CREATE VIEW comedies AS
  SELECT *
  FROM films
  WHERE kind = ‘Comedy’;

Compatibility

The SQL standard specifies some additional capabilities for the CREATE VIEW statement:

CREATE VIEW name [ ( column_name [ , ... ] ) ]
  AS query
  [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]

The optional clauses for the full SQL command are:
CREATE VIEW

CHECK OPTION

This option has to do with updatable views. All INSERT and UPDATE commands on the view will be checked to ensure data satisfy the view-defining condition (that is, the new data would be visible through the view). If they do not, the update will be rejected.

LOCAL

Check for integrity on this view.

CASCADED

Check for integrity on this view and on any dependent view. CASCADED is assumed if neither CASCADED nor LOCAL is specified.

CREATE OR REPLACE VIEW is a PostgreSQL language extension. So is the concept of a temporary view.

See Also

ALTER VIEW, DROP VIEW
DEALLOCATE

Name
DEALLOCATE — deallocate a prepared statement

Synopsis
DEALLOCATE [ PREPARE ] { name | ALL }

Description
DEALLOCATE is used to deallocate a previously prepared SQL statement. If you do not explicitly deallocate a prepared statement, it is deallocated when the session ends.

For more information on prepared statements, see PREPARE.

Parameters
PREPARE
This key word is ignored.

name
The name of the prepared statement to deallocate.

ALL
Deallocation all prepared statements.

Compatibility
The SQL standard includes a DEALLOCATE statement, but it is only for use in embedded SQL.

See Also
EXECUTE, PREPARE
DECLARE

Name

DECLARE — define a cursor

Synopsis

DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
CURSOR [ { WITH | WITHOUT } HOLD ] FOR query

Description

DECLARE allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. After the cursor is created, rows are fetched from it using FETCH.

Parameters

name

The name of the cursor to be created.

BINARY

Causes the cursor to return data in binary rather than in text format.

INSENSITIVE

Indicates that data retrieved from the cursor should be unaffected by updates to the table(s) underlying the cursor that occur after the cursor is created. In PostgreSQL, this is the default behavior; so this key word has no effect and is only accepted for compatibility with the SQL standard.

SCROLL

NO SCROLL

SCROLL specifies that the cursor can be used to retrieve rows in a nonsequential fashion (e.g., backward). Depending upon the complexity of the query’s execution plan, specifying SCROLL might impose a performance penalty on the query’s execution time. NO SCROLL specifies that the cursor cannot be used to retrieve rows in a nonsequential fashion. The default is to allow scrolling in some cases; this is not the same as specifying SCROLL. See Notes for details.

WITH HOLD

WITHOUT HOLD

WITH HOLD specifies that the cursor can continue to be used after the transaction that created it successfully commits. WITHOUT HOLD specifies that the cursor cannot be used outside of the transaction that created it. If neither WITHOUT HOLD nor WITH HOLD is specified, WITHOUT HOLD is the default.

query

A SELECT or VALUES command which will provide the rows to be returned by the cursor.

The key words BINARY, INSENSITIVE, and SCROLL can appear in any order.
Notes

Normal cursors return data in text format, the same as a `SELECT` would produce. The `BINARY` option specifies that the cursor should return data in binary format. This reduces conversion effort for both the server and client, at the cost of more programmer effort to deal with platform-dependent binary data formats. As an example, if a query returns a value of one from an integer column, you would get a string of `1` with a default cursor, whereas with a binary cursor you would get a 4-byte field containing the internal representation of the value (in big-endian byte order).

Binary cursors should be used carefully. Many applications, including `psql`, are not prepared to handle binary cursors and expect data to come back in the text format.

Note: When the client application uses the “extended query” protocol to issue a `FETCH` command, the Bind protocol message specifies whether data is to be retrieved in text or binary format. This choice overrides the way that the cursor is defined. The concept of a binary cursor as such is thus obsolete when using extended query protocol — any cursor can be treated as either text or binary.

Unless `WITH HOLD` is specified, the cursor created by this command can only be used within the current transaction. Thus, `DECLARE` without `WITH HOLD` is useless outside a transaction block: the cursor would survive only to the completion of the statement. Therefore PostgreSQL reports an error if such a command is used outside a transaction block. Use `BEGIN`, `COMMIT` and `ROLLBACK` to define a transaction block.

If `WITH HOLD` is specified and the transaction that created the cursor successfully commits, the cursor can continue to be accessed by subsequent transactions in the same session. (But if the creating transaction is aborted, the cursor is removed.) A cursor created with `WITH HOLD` is closed when an explicit `CLOSE` command is issued on it, or the session ends. In the current implementation, the rows represented by a held cursor are copied into a temporary file or memory area so that they remain available for subsequent transactions.

`WITH HOLD` may not be specified when the query includes `FOR UPDATE` or `FOR SHARE`.

The `SCROLL` option should be specified when defining a cursor that will be used to fetch backwards. This is required by the SQL standard. However, for compatibility with earlier versions, PostgreSQL will allow backward fetches without `SCROLL`, if the cursor’s query plan is simple enough that no extra overhead is needed to support it. However, application developers are advised not to rely on using backward fetches from a cursor that has not been created with `SCROLL`. If `NO SCROLL` is specified, then backward fetches are disallowed in any case.

If the cursor’s query includes `FOR UPDATE` or `FOR SHARE`, then returned rows are locked at the time they are first fetched, in the same way as for a regular `SELECT` command with these options. In addition, the returned rows will be the most up-to-date versions; therefore these options provide the equivalent of what the SQL standard calls a “sensitive cursor”. It is often wise to use `FOR UPDATE` if the cursor is intended to be used with `UPDATE ... WHERE CURRENT OF` or `DELETE ... WHERE CURRENT OF`, since this will prevent other sessions from changing the rows between the time they are fetched and the time they are updated. Without `FOR UPDATE`, a subsequent `WHERE CURRENT OF` command will have no effect if the row was changed meanwhile.

`SCROLL` may not be specified when the query includes `FOR UPDATE` or `FOR SHARE`.

The SQL standard only makes provisions for cursors in embedded SQL. The PostgreSQL server does not implement an `OPEN` statement for cursors; a cursor is considered to be open when it is declared. However, ECPG, the embedded SQL preprocessor for PostgreSQL, supports the standard SQL cursor conventions, including those involving `DECLARE` and `OPEN` statements.
DECLARE

You can see all available cursors by querying the `pg_cursors` system view.

Examples

To declare a cursor:

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

See `FETCH` for more examples of cursor usage.

Compatibility

The SQL standard says that it is implementation-dependent whether cursors are sensitive to concurrent updates of the underlying data by default. In PostgreSQL, cursors are insensitive by default, and can be made sensitive by specifying `FOR UPDATE`. Other products may work differently.

The SQL standard allows cursors only in embedded SQL and in modules. PostgreSQL permits cursors to be used interactively.

Binary cursors are a PostgreSQL extension.

See Also

`CLOSE, FETCH, MOVE`
DELETE

**Name**

DELETE — delete rows of a table

**Synopsis**

```
DELETE FROM [ ONLY ] table [ * ] [ [ AS ] alias ]
[ USING usinglist ]
[ WHERE condition | WHERE CURRENT OF cursor_name ]
[ RETURNING * | output_expression [ AS output_name ] [, ... ] ]
```

**Description**

DELETE deletes rows that satisfy the WHERE clause from the specified table. If the WHERE clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

**Tip:** TRUNCATE is a PostgreSQL extension that provides a faster mechanism to remove all rows from a table.

There are two ways to delete rows in a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the USING clause. Which technique is more appropriate depends on the specific circumstances.

The optional RETURNING clause causes DELETE to compute and return value(s) based on each row actually deleted. Any expression using the table’s columns, and/or columns of other tables mentioned in USING, can be computed. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have the DELETE privilege on the table to delete from it, as well as the SELECT privilege for any table in the USING clause or whose values are read in the condition.

**Parameters**

**table**

The name (optionally schema-qualified) of the table to delete rows from. If ONLY is specified before the table name, matching rows are deleted from the named table only. If ONLY is not specified, matching rows are also deleted from any tables inheriting from the named table. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

**alias**

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given DELETE FROM foo AS f, the remainder of the DELETE statement must refer to this table as f not foo.
**DELETE**

*usinglist*

A list of table expressions, allowing columns from other tables to appear in the `WHERE` condition. This is similar to the list of tables that can be specified in the `FROM` Clause of a `SELECT` statement; for example, an alias for the table name can be specified. Do not repeat the target table in the `usinglist`, unless you wish to set up a self-join.

*condition*

An expression that returns a value of type `boolean`. Only rows for which this expression returns `true` will be deleted.

*cursor_name*

The name of the cursor to use in a `WHERE CURRENT OF` condition. The row to be deleted is the one most recently fetched from this cursor. The cursor must be a simple (non-join, non-aggregate) query on the `DELETE`'s target table. Note that `WHERE CURRENT OF` cannot be specified together with a Boolean condition.

*output_expression*

An expression to be computed and returned by the `DELETE` command after each row is deleted. The expression can use any column names of the `table` or table(s) listed in `USING`. Write `*` to return all columns.

*output_name*

A name to use for a returned column.

**Outputs**

On successful completion, a `DELETE` command returns a command tag of the form

```
DELETE count
```

The `count` is the number of rows deleted. If `count` is 0, no rows matched the `condition` (this is not considered an error).

If the `DELETE` command contains a `RETURNING` clause, the result will be similar to that of a `SELECT` statement containing the columns and values defined in the `RETURNING` list, computed over the row(s) deleted by the command.

**Notes**

PostgreSQL lets you reference columns of other tables in the `WHERE` condition by specifying the other tables in the `USING` clause. For example, to delete all films produced by a given producer, one can do:

```
DELETE FROM films USING producers
WHERE producer_id = producers.id AND producers.name = 'foo';
```

What is essentially happening here is a join between `films` and `producers`, with all successfully joined `films` rows being marked for deletion. This syntax is not standard. A more standard way to do it is:

```
DELETE FROM films
WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

In some cases the join style is easier to write or faster to execute than the sub-select style.
Examples

Delete all films but musicals:

DELETE FROM films WHERE kind <> 'Musical';

Clear the table films:

DELETE FROM films;

Delete completed tasks, returning full details of the deleted rows:

DELETE FROM tasks WHERE status = 'DONE' RETURNING *;

Delete the row of tasks on which the cursor c_tasks is currently positioned:

DELETE FROM tasks WHERE CURRENT OF c_tasks;

Compatibility

This command conforms to the SQL standard, except that the USING and RETURNING clauses are PostgreSQL extensions.
**DISCARD**

**Name**

DISCARD — discard session state

**Synopsis**

DISCARD { ALL | PLANS | TEMPORARY | TEMP }

**Description**

DISCARD releases internal resources associated with a database session. These resources are normally released at the end of the session.

DISCARD TEMP drops all temporary tables created in the current session. DISCARD PLANS releases all internally cached query plans. DISCARD ALL resets a session to its original state, discarding temporary resources and resetting session-local configuration changes.

**Parameters**

TEMPORARY or TEMP

Drops all temporary tables created in the current session.

PLANS

Releases all cached query plans.

ALL

Releases all temporary resources associated with the current session and resets the session to its initial state. Currently, this has the same effect as executing the following sequence of statements:

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
```

**Notes**

DISCARD ALL cannot be executed inside a transaction block.

**Compatibility**

DISCARD is a PostgreSQL extension.
DROP AGGREGATE

Name

DROP AGGREGATE — remove an aggregate function

Synopsis

DROP AGGREGATE [ IF EXISTS ] name ( type [ , ... ] ) [ CASCADE | RESTRICT ]

Description

DROP AGGREGATE removes an existing aggregate function. To execute this command the current user must be the owner of the aggregate function.

Parameters

IF EXISTS

Do not throw an error if the aggregate does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing aggregate function.

type

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of input data types.

CASCADE

Automatically drop objects that depend on the aggregate function.

RESTRICT

Refuse to drop the aggregate function if any objects depend on it. This is the default.

Examples

To remove the aggregate function myavg for type integer:

DROP AGGREGATE myavg(integer);

Compatibility

There is no DROP AGGREGATE statement in the SQL standard.
See Also

ALTER AGGREGATE, CREATE AGGREGATE
DROP CAST

Name
DROP CAST — remove a cast

Synopsis
DROP CAST [ IF EXISTS ] (sourcetype AS targettype) [ CASCADE | RESTRICT ]

Description
DROP CAST removes a previously defined cast.
To be able to drop a cast, you must own the source or the target data type. These are the same privileges that are required to create a cast.

Parameters
IF EXISTS
Do not throw an error if the cast does not exist. A notice is issued in this case.
sourcetype
The name of the source data type of the cast.
targettype
The name of the target data type of the cast.
CASCADE
RESTRICT
These key words do not have any effect, since there are no dependencies on casts.

Examples
To drop the cast from type text to type int:
DROP CAST (text AS int);

Compatibility
The DROP CAST command conforms to the SQL standard.
See Also

CREATE CAST
DROP CONVERSION

Name
DROP CONVERSION — remove a conversion

Synopsis
DROP CONVERSION [ IF EXISTS ] name [ CASCADE | RESTRICT ]

Description
DROP CONVERSION removes a previously defined conversion. To be able to drop a conversion, you must own the conversion.

Parameters
IF EXISTS
Do not throw an error if the conversion does not exist. A notice is issued in this case.
name
The name of the conversion. The conversion name can be schema-qualified.
CASCADE
RESTRICT
These key words do not have any effect, since there are no dependencies on conversions.

Examples
To drop the conversion named myname:

DROP CONVERSION myname;

Compatibility
There is no DROP CONVERSION statement in the SQL standard.

See Also
ALTER CONVERSION, CREATE CONVERSION
DROP DATABASE

Name
DROP DATABASE — remove a database

Synopsis
DROP DATABASE [ IF EXISTS ] name

Description
DROP DATABASE drops a database. It removes the catalog entries for the database and deletes the directory containing the data. It can only be executed by the database owner. Also, it cannot be executed while you or anyone else are connected to the target database. (Connect to postgres or any other database to issue this command.)

DROP DATABASE cannot be undone. Use it with care!

Parameters

IF EXISTS
Do not throw an error if the database does not exist. A notice is issued in this case.

name
The name of the database to remove.

Notes
DROP DATABASE cannot be executed inside a transaction block.

This command cannot be executed while connected to the target database. Thus, it might be more convenient to use the program dropdb instead, which is a wrapper around this command.

Compatibility
There is no DROP DATABASE statement in the SQL standard.

See Also
CREATE DATABASE
DROP DOMAIN

Name
DROP DOMAIN — remove a domain

Synopsis
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

Description
DROP DOMAIN removes a domain. Only the owner of a domain can remove it.

Parameters
IF EXISTS
   Do not throw an error if the domain does not exist. A notice is issued in this case.
name
   The name (optionally schema-qualified) of an existing domain.
CASCADE
   Automatically drop objects that depend on the domain (such as table columns).
RESTRICT
   Refuse to drop the domain if any objects depend on it. This is the default.

Examples
To remove the domain box:
DROP DOMAIN box;

Compatibility
This command conforms to the SQL standard, except for the IF EXISTS option, which is a PostgreSQL extension.

See Also
CREATE DOMAIN, ALTER DOMAIN
DROP FUNCTION

Name

DROP FUNCTION — remove a function

Synopsis

DROP FUNCTION [ IF EXISTS ] name ( [ [ argmode ] [ argname argtype [, ...] ] ]
[ CASCADE | RESTRICT ]

Description

DROP FUNCTION removes the definition of an existing function. To execute this command the user
must be the owner of the function. The argument types to the function must be specified, since several
different functions can exist with the same name and different argument lists.

Parameters

IF EXISTS
   Do not throw an error if the function does not exist. A notice is issued in this case.
name
   The name (optionally schema-qualified) of an existing function.
argmode
   The mode of an argument: either IN, OUT, or INOUT. If omitted, the default is IN. Note that
   DROP FUNCTION does not actually pay any attention to OUT arguments, since only the input
   arguments are needed to determine the function’s identity. So it is sufficient to list the IN and
   INOUT arguments.
argname
   The name of an argument. Note that DROP FUNCTION does not actually pay any attention to
   argument names, since only the argument data types are needed to determine the function’s
   identity.
arctype
   The data type(s) of the function’s arguments (optionally schema-qualified), if any.
CASCADE
   Automatically drop objects that depend on the function (such as operators or triggers).
RESTRICT
   Refuse to drop the function if any objects depend on it. This is the default.
Examples
This command removes the square root function:

DROP FUNCTION sqrt(integer);

Compatibility
A DROP FUNCTION statement is defined in the SQL standard, but it is not compatible with this command.

See Also
CREATE FUNCTION, ALTER FUNCTION
DROP GROUP

Name

DROP GROUP — remove a database role

Synopsis

DROP GROUP [ IF EXISTS ] name [ , ... ]

Description

DROP GROUP is now an alias for DROP ROLE.

Compatibility

There is no DROP GROUP statement in the SQL standard.

See Also

DROP ROLE
DROP INDEX

Name
DROP INDEX — remove an index

Synopsis
DROP INDEX [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

Description
DROP INDEX drops an existing index from the database system. To execute this command you must be the owner of the index.

Parameters
IF EXISTS
Do not throw an error if the index does not exist. A notice is issued in this case.
name
The name (optionally schema-qualified) of an index to remove.
CASCADE
Automatically drop objects that depend on the index.
RESTRICT
Refuse to drop the index if any objects depend on it. This is the default.

Examples
This command will remove the index title_idx:

DROP INDEX title_idx;

Compatibility
DROP INDEX is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

See Also
CREATE INDEX
DROP LANGUAGE

Name
DROP LANGUAGE — remove a procedural language

Synopsis
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]

Description
DROP LANGUAGE removes the definition of a previously registered procedural language. You must be a superuser or the owner of the language to use DROP LANGUAGE.

Parameters

IF EXISTS
Do not throw an error if the language does not exist. A notice is issued in this case.

name
The name of an existing procedural language. For backward compatibility, the name can be enclosed by single quotes.

CASCADE
Automatically drop objects that depend on the language (such as functions in the language).

RESTRICT
Refuse to drop the language if any objects depend on it. This is the default.

Examples
This command removes the procedural language plsample:

DROP LANGUAGE plsample;

Compatibility
There is no DROP LANGUAGE statement in the SQL standard.

See Also
ALTER LANGUAGE, CREATE LANGUAGE, droplang
DROP OPERATOR

Name
DROP OPERATOR — remove an operator

Synopsis
DROP OPERATOR [ IF EXISTS ] name ( { lefttype | NONE } , { righttype | NONE } ) [ CASCADE | RESTRICT ]

Description
DROP OPERATOR drops an existing operator from the database system. To execute this command you must be the owner of the operator.

Parameters
IF EXISTS
Do not throw an error if the operator does not exist. A notice is issued in this case.
name
The name (optionally schema-qualified) of an existing operator.
lefttype
The data type of the operator’s left operand; write NONE if the operator has no left operand.
righttype
The data type of the operator’s right operand; write NONE if the operator has no right operand.
CASCADE
Automatically drop objects that depend on the operator.
RESTRICT
Refuse to drop the operator if any objects depend on it. This is the default.

Examples
Remove the power operator \(^{\text{a}}\)\(^{\text{b}}\) for type integer:
DROP OPERATOR ^ (integer, integer);

Remove the left unary bitwise complement operator \(^{\neg}\) for type bit:
DROP OPERATOR ~ (none, bit);
DROP OPERATOR

Remove the right unary factorial operator \( x! \) for type \( \text{bigint} \):

DROP OPERATOR ! (bigint, none);

Compatibility

There is no DROP OPERATOR statement in the SQL standard.

See Also

CREATE OPERATOR, ALTER OPERATOR
DROP OPERATOR CLASS

Name

DROP OPERATOR CLASS — remove an operator class

Synopsis

DROP OPERATOR CLASS [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]

Description

DROP OPERATOR CLASS drops an existing operator class. To execute this command you must be the owner of the operator class.

DROP OPERATOR CLASS does not drop any of the operators or functions referenced by the class. If there are any indexes depending on the operator class, you will need to specify CASCADE for the drop to complete.

Parameters

IF EXISTS

Do not throw an error if the operator class does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing operator class.

index_method

The name of the index access method the operator class is for.

CASCADE

Automatically drop objects that depend on the operator class.

RESTRICT

Refuse to drop the operator class if any objects depend on it. This is the default.

Notes

DROP OPERATOR CLASS will not drop the operator family containing the class, even if there is nothing else left in the family (in particular, in the case where the family was implicitly created by CREATE OPERATOR CLASS). An empty operator family is harmless, but for the sake of tidiness you might wish to remove the family with DROP OPERATOR FAMILY; or perhaps better, use DROP OPERATOR FAMILY in the first place.
Examples

Remove the B-tree operator class `widget_ops`:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

This command will not succeed if there are any existing indexes that use the operator class. Add `CASCADE` to drop such indexes along with the operator class.

Compatibility

There is no `DROP OPERATOR CLASS` statement in the SQL standard.

See Also

`ALTER OPERATOR CLASS`, `CREATE OPERATOR CLASS`, `DROP OPERATOR FAMILY`
DROP OPERATOR FAMILY

**Name**

DROP OPERATOR FAMILY — remove an operator family

**Synopsis**

```
DROP OPERATOR FAMILY [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

**Description**

DROP OPERATOR FAMILY drops an existing operator family. To execute this command you must be the owner of the operator family.

DROP OPERATOR FAMILY includes dropping any operator classes contained in the family, but it does not drop any of the operators or functions referenced by the family. If there are any indexes depending on operator classes within the family, you will need to specify CASCADE for the drop to complete.

**Parameters**

**IF EXISTS**

Do not throw an error if the operator family does not exist. A notice is issued in this case.

**name**

The name (optionally schema-qualified) of an existing operator family.

**index_method**

The name of the index access method the operator family is for.

**CASCADE**

Automatically drop objects that depend on the operator family.

**RESTRICT**

Refuse to drop the operator family if any objects depend on it. This is the default.

**Examples**

Remove the B-tree operator family `float_ops`:

```
DROP OPERATOR FAMILY float_ops USING btree;
```

This command will not succeed if there are any existing indexes that use operator classes within the family. Add CASCADE to drop such indexes along with the operator family.
Compatibility

There is no \texttt{DROP OPERATOR FAMILY} statement in the SQL standard.

See Also

\texttt{ALTER OPERATOR FAMILY}, \texttt{CREATE OPERATOR FAMILY}, \texttt{ALTER OPERATOR CLASS}, \texttt{CREATE OPERATOR CLASS}, \texttt{DROP OPERATOR CLASS}
**DROP OWNED**

**Name**

DROP OWNED — remove database objects owned by a database role

**Synopsis**

DROP OWNED BY name [, ...] [ CASCADE | RESTRICT ]

**Description**

DROP OWNED drops all the objects within the current database that are owned by one of the specified roles. Any privileges granted to the given roles on objects in the current database and on shared objects (databases, tablespaces) will also be revoked.

**Parameters**

- **name**
  
  The name of a role whose objects will be dropped, and whose privileges will be revoked.

- **CASCADE**
  
  Automatically drop objects that depend on the affected objects.

- **RESTRICT**
  
  Refuse to drop the objects owned by a role if any other database objects depend on one of the affected objects. This is the default.

**Notes**

DROP OWNED is often used to prepare for the removal of one or more roles. Because DROP OWNED only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

Using the CASCADE option might make the command recurse to objects owned by other users.

The REASSIGN OWNED command is an alternative that reassigns the ownership of all the database objects owned by one or more roles.

Databases and tablespaces owned by the role(s) will not be removed.

**Compatibility**

The DROP OWNED statement is a PostgreSQL extension.
See Also

REASSIGN OWNED, DROP ROLE
DROP ROLE

**Name**

DROP ROLE — remove a database role

**Synopsis**

DROP ROLE [ IF EXISTS ] name [ , ... ]

**Description**

DROP ROLE removes the specified role(s). To drop a superuser role, you must be a superuser yourself; to drop non-superuser roles, you must have CREATEROLE privilege.

A role cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted. The REASSIGN OWNED and DROP OWNED commands can be useful for this purpose.

However, it is not necessary to remove role memberships involving the role; DROP ROLE automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

**Parameters**

**IF EXISTS**

Do not throw an error if the role does not exist. A notice is issued in this case.

**name**

The name of the role to remove.

**Notes**

PostgreSQL includes a program dropuser that has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

**Examples**

To drop a role:

```
DROP ROLE jonathan;
```
Compatibility

The SQL standard defines `DROP ROLE`, but it allows only one role to be dropped at a time, and it specifies different privilege requirements than PostgreSQL uses.

See Also

`CREATE ROLE`, `ALTER ROLE`, `SET ROLE`
DROP RULE

Name
DROP RULE — remove a rewrite rule

Synopsis
DROP RULE [ IF EXISTS ] name ON relation [ CASCADE | RESTRICT ]

Description
DROP RULE drops a rewrite rule.

Parameters

IF EXISTS
Do not throw an error if the rule does not exist. A notice is issued in this case.

name
The name of the rule to drop.

relation
The name (optionally schema-qualified) of the table or view that the rule applies to.

CASCADE
Automatically drop objects that depend on the rule.

RESTRICT
Refuse to drop the rule if any objects depend on it. This is the default.

Examples

To drop the rewrite rule newrule:

DROP RULE newrule ON mytable;

Compatibility

There is no DROP RULE statement in the SQL standard.
See Also

CREATE RULE
**DROP SCHEMA**

**Name**

DROP SCHEMA — remove a schema

**Synopsis**

DROP SCHEMA [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

**Description**

DROP SCHEMA removes schemas from the database.

A schema can only be dropped by its owner or a superuser. Note that the owner can drop the schema (and thereby all contained objects) even if he does not own some of the objects within the schema.

**Parameters**

**IF EXISTS**

Do not throw an error if the schema does not exist. A notice is issued in this case.

name

The name of a schema.

**CASCADE**

Automatically drop objects (tables, functions, etc.) that are contained in the schema.

**RESTRICT**

Refuse to drop the schema if it contains any objects. This is the default.

**Examples**

To remove schema mystuff from the database, along with everything it contains:

DROP SCHEMA mystuff CASCADE;

**Compatibility**

DROP SCHEMA is fully conforming with the SQL standard, except that the standard only allows one schema to be dropped per command, and apart from the IF EXISTS option, which is a PostgreSQL extension.
See Also

ALTER SCHEMA, CREATE SCHEMA
DROP SEQUENCE

Name
DROP SEQUENCE — remove a sequence

Synopsis
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

Description
DROP SEQUENCE removes sequence number generators. A sequence can only be dropped by its owner or a superuser.

Parameters

IF EXISTS
Do not throw an error if the sequence does not exist. A notice is issued in this case.

name
The name (optionally schema-qualified) of a sequence.

CASCADE
Automatically drop objects that depend on the sequence.

RESTRICT
Refuse to drop the sequence if any objects depend on it. This is the default.

Examples
To remove the sequence serial:

DROP SEQUENCE serial;

Compatibility
DROP SEQUENCE conforms to the SQL standard, except that the standard only allows one sequence to be dropped per command, and apart from the IF EXISTS option, which is a PostgreSQL extension.

See Also
CREATE SEQUENCE, ALTER SEQUENCE
DROP TABLE

Name
DROP TABLE — remove a table

Synopsis
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

Description
DROP TABLE removes tables from the database. Only its owner can drop a table. To empty a table of rows without destroying the table, use DELETE or TRUNCATE.

DROP TABLE always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a view or a foreign-key constraint of another table, CASCADE must be specified. (CASCADE will remove a dependent view entirely, but in the foreign-key case it will only remove the foreign-key constraint, not the other table entirely.)

Parameters

IF EXISTS
Do not throw an error if the table does not exist. A notice is issued in this case.

name
The name (optionally schema-qualified) of the table to drop.

CASCADE
Automatically drop objects that depend on the table (such as views).

RESTRICT
Refuse to drop the table if any objects depend on it. This is the default.

Examples
To destroy two tables, films and distributors:

DROP TABLE films, distributors;

Compatibility
This command conforms to the SQL standard, except that the standard only allows one table to be dropped per command, and apart from the IF EXISTS option, which is a PostgreSQL extension.
See Also

ALTER TABLE, CREATE TABLE
DROP TABLESPACE

Name
DROP TABLESPACE — remove a tablespace

Synopsis
DROP TABLESPACE [ IF EXISTS ] tablesapce

Description
DROP TABLESPACE removes a tablespace from the system.
A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all
database objects before it can be dropped. It is possible that objects in other databases might still
reside in the tablespace even if no objects in the current database are using the tablespace. Also, if the
tablespace is listed in the temp_tablespaces setting of any active session, the DROP might fail due to
temporary files residing in the tablespace.

Parameters
IF EXISTS
Do not throw an error if the tablespace does not exist. A notice is issued in this case.

tablesapce
The name of a tablespace.

Notes
DROP TABLESPACE cannot be executed inside a transaction block.

Examples
To remove tablesapce mystuff from the system:

DROP TABLESPACE mystuff;

Compatibility
DROP TABLESPACE is a PostgreSQL extension.
See Also

CREATE TABLESPACE, ALTER TABLESPACE
DROP TEXT SEARCH CONFIGURATION

Name

DROP TEXT SEARCH CONFIGURATION — remove a text search configuration

Synopsis

DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]

Description

DROP TEXT SEARCH CONFIGURATION drops an existing text search configuration. To execute this command you must be the owner of the configuration.

Parameters

IF EXISTS

Do not throw an error if the text search configuration does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing text search configuration.

CASCADE

Automatically drop objects that depend on the text search configuration.

RESTRICT

Refuse to drop the text search configuration if any objects depend on it. This is the default.

Examples

Remove the text search configuration my_english:

DROP TEXT SEARCH CONFIGURATION my_english;

This command will not succeed if there are any existing indexes that reference the configuration in to_tsvector calls. Add CASCADE to drop such indexes along with the text search configuration.

Compatibility

There is no DROP TEXT SEARCH CONFIGURATION statement in the SQL standard.
See Also

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION
DROP TEXT SEARCH DICTIONARY

Name
DROP TEXT SEARCH DICTIONARY — remove a text search dictionary

Synopsis
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]

Description
DROP TEXT SEARCH DICTIONARY drops an existing text search dictionary. To execute this command you must be the owner of the dictionary.

Parameters
IF EXISTS
   Do not throw an error if the text search dictionary does not exist. A notice is issued in this case.
name
   The name (optionally schema-qualified) of an existing text search dictionary.
CASCADE
   Automatically drop objects that depend on the text search dictionary.
RESTRICT
   Refuse to drop the text search dictionary if any objects depend on it. This is the default.

Examples
Remove the text search dictionary english:
DROP TEXT SEARCH DICTIONARY english;

This command will not succeed if there are any existing text search configurations that use the dictionary. Add CASCADE to drop such configurations along with the dictionary.

Compatibility
There is no DROP TEXT SEARCH DICTIONARY statement in the SQL standard.

See Also
ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY
DROP TEXT SEARCH PARSER

Name
DROP TEXT SEARCH PARSER — remove a text search parser

Synopsis
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]

Description
DROP TEXT SEARCH PARSER drops an existing text search parser. You must be a superuser to use this command.

Parameters
IF EXISTS
Do not throw an error if the text search parser does not exist. A notice is issued in this case.

name
The name (optionally schema-qualified) of an existing text search parser.

CASCADE
Automatically drop objects that depend on the text search parser.

RESTRICT
Refuse to drop the text search parser if any objects depend on it. This is the default.

Examples
Remove the text search parser my_parser:

DROP TEXT SEARCH PARSER my_parser;

This command will not succeed if there are any existing text search configurations that use the parser. Add CASCADE to drop such configurations along with the parser.

Compatibility
There is no DROP TEXT SEARCH PARSER statement in the SQL standard.

See Also
ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER
DROP TEXT SEARCH TEMPLATE

Name
DROP TEXT SEARCH TEMPLATE — remove a text search template

Synopsis
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]

Description
DROP TEXT SEARCH TEMPLATE drops an existing text search template. You must be a superuser to use this command.

Parameters

IF EXISTS
Do not throw an error if the text search template does not exist. A notice is issued in this case.

name
The name (optionally schema-qualified) of an existing text search template.

CASCADE
Automatically drop objects that depend on the text search template.

RESTRICT
Refuse to drop the text search template if any objects depend on it. This is the default.

Examples
Remove the text search template thesaurus:

DROP TEXT SEARCH TEMPLATE thesaurus;

This command will not succeed if there are any existing text search dictionaries that use the template. Add CASCADE to drop such dictionaries along with the template.

Compatibility
There is no DROP TEXT SEARCH TEMPLATE statement in the SQL standard.

See Also
ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE
**DROP TRIGGER**

**Name**

DROP TRIGGER — remove a trigger

**Synopsis**

DROP TRIGGER [ IF EXISTS ] name ON table [ CASCADE | RESTRICT ]

**Description**

DROP TRIGGER removes an existing trigger definition. To execute this command, the current user must be the owner of the table for which the trigger is defined.

**Parameters**

**IF EXISTS**

Do not throw an error if the trigger does not exist. A notice is issued in this case.

*name*

The name of the trigger to remove.

*table*

The name (optionally schema-qualified) of the table for which the trigger is defined.

**CASCADE**

Automatically drop objects that depend on the trigger.

**RESTRICT**

Refuse to drop the trigger if any objects depend on it. This is the default.

**Examples**

Destroy the trigger if_dist_exists on the table films:

```
DROP TRIGGER if_dist_exists ON films;
```

**Compatibility**

The DROP TRIGGER statement in PostgreSQL is incompatible with the SQL standard. In the SQL standard, trigger names are not local to tables, so the command is simply DROP TRIGGER *name*. 
See Also

CREATE TRIGGER
DROP TYPE

Name
DROP TYPE — remove a data type

Synopsis
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

Description
DROP TYPE removes a user-defined data type. Only the owner of a type can remove it.

Parameters

IF EXISTS
Do not throw an error if the type does not exist. A notice is issued in this case.

name
The name (optionally schema-qualified) of the data type to remove.

CASCADE
Automatically drop objects that depend on the type (such as table columns, functions, operators).

RESTRICT
Refuse to drop the type if any objects depend on it. This is the default.

Examples
To remove the data type box:

DROP TYPE box;

Compatibility
This command is similar to the corresponding command in the SQL standard, apart from the IF EXISTS option, which is a PostgreSQL extension. But note that the CREATE TYPE command and the data type extension mechanisms in PostgreSQL differ from the SQL standard.

See Also
CREATE TYPE, ALTER TYPE
DROP USER

Name
DROP USER — remove a database role

Synopsis
DROP USER [ IF EXISTS ] name [, ...]

Description
DROP USER is now an alias for DROP ROLE.

Compatibility
The DROP USER statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also
DROP ROLE
DROP VIEW

Name
DROP VIEW — remove a view

Synopsis
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

Description
DROP VIEW drops an existing view. To execute this command you must be the owner of the view.

Parameters

IF EXISTS
Do not throw an error if the view does not exist. A notice is issued in this case.

name
The name (optionally schema-qualified) of the view to remove.

CASCADE
Automatically drop objects that depend on the view (such as other views).

RESTRICT
Refuse to drop the view if any objects depend on it. This is the default.

Examples
This command will remove the view called kinds:

DROP VIEW kinds;

Compatibility
This command conforms to the SQL standard, except that the standard only allows one view to be dropped per command, and apart from the IF EXISTS option, which is a PostgreSQL extension.

See Also
ALTER VIEW, CREATE VIEW
**Name**

END — commit the current transaction

**Synopsis**

END [ WORK | TRANSACTION ]

**Description**

END commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs. This command is a PostgreSQL extension that is equivalent to `COMMIT`.

**Parameters**

WORK  
TRANSACTION  

Optional key words. They have no effect.

**Notes**

Use `ROLLBACK` to abort a transaction.  
Issuing END when not inside a transaction does no harm, but it will provoke a warning message.

**Examples**

To commit the current transaction and make all changes permanent:

END;

**Compatibility**

END is a PostgreSQL extension that provides functionality equivalent to `COMMIT`, which is specified in the SQL standard.

**See Also**

BEGIN, COMMIT, ROLLBACK
EXECUTE

Name

EXECUTE — execute a prepared statement

Synopsis

EXECUTE name [ ( parameter [, ..., ] ) ]

Description

EXECUTE is used to execute a previously prepared statement. Since prepared statements only exist for the duration of a session, the prepared statement must have been created by a PREPARE statement executed earlier in the current session.

If the PREPARE statement that created the statement specified some parameters, a compatible set of parameters must be passed to the EXECUTE statement, or else an error is raised. Note that (unlike functions) prepared statements are not overloaded based on the type or number of their parameters; the name of a prepared statement must be unique within a database session.

For more information on the creation and usage of prepared statements, see PREPARE.

Parameters

name

The name of the prepared statement to execute.

parameter

The actual value of a parameter to the prepared statement. This must be an expression yielding a value that is compatible with the data type of this parameter, as was determined when the prepared statement was created.

Outputs

The command tag returned by EXECUTE is that of the prepared statement, and not EXECUTE.

Examples

Examples are given in the Examples section of the PREPARE documentation.

Compatibility

The SQL standard includes an EXECUTE statement, but it is only for use in embedded SQL. This version of the EXECUTE statement also uses a somewhat different syntax.
See Also

DEALLOCATE, PREPARE
**EXPLAIN**

**Name**

EXPLAIN — show the execution plan of a statement

**Synopsis**

EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

**Description**

This command displays the execution plan that the PostgreSQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table.

The most critical part of the display is the estimated statement execution cost, which is the planner’s guess at how long it will take to run the statement (measured in units of disk page fetches). Actually two numbers are shown: the start-up time before the first row can be returned, and the total time to return all the rows. For most queries the total time is what matters, but in contexts such as a subquery in EXISTS, the planner will choose the smallest start-up time instead of the smallest total time (since the executor will stop after getting one row, anyway). Also, if you limit the number of rows to return with a LIMIT clause, the planner makes an appropriate interpolation between the endpoint costs to estimate which plan is really the cheapest.

The ANALYZE option causes the statement to be actually executed, not only planned. The total elapsed time expended within each plan node (in milliseconds) and total number of rows it actually returned are added to the display. This is useful for seeing whether the planner’s estimates are close to reality.

**Important:** Keep in mind that the statement is actually executed when the ANALYZE option is used. Although EXPLAIN will discard any output that a SELECT would return, other side effects of the statement will happen as usual. If you wish to use EXPLAIN ANALYZE on an INSERT, UPDATE, DELETE, or EXECUTE statement without letting the command affect your data, use this approach:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

**Parameters**

ANALYZE

Carry out the command and show the actual run times.
EXPLAIN

VERBOSE

Show the full internal representation of the plan tree, rather than just a summary. Usually this option is only useful for specialized debugging purposes. The VERBOSE output is either pretty-printed or not, depending on the setting of the explain_pretty_print configuration parameter.

statement

Any SELECT, INSERT, UPDATE, DELETE, VALUES, EXECUTE, or DECLARE statement, whose execution plan you wish to see.

Notes

There is only sparse documentation on the optimizer’s use of cost information in PostgreSQL. Refer to Section 14.1 for more information.

In order to allow the PostgreSQL query planner to make reasonably informed decisions when optimizing queries, the ANALYZE statement should be run to record statistics about the distribution of data within the table. If you have not done this (or if the statistical distribution of the data in the table has changed significantly since the last time ANALYZE was run), the estimated costs are unlikely to conform to the real properties of the query, and consequently an inferior query plan might be chosen.

Genetic query optimization (GEQO) randomly tests execution plans. Therefore, when the number of join relations exceeds geqo_threshold causing genetic query optimization to be used, the execution plan is likely to change each time the statement is executed.

In order to measure the run-time cost of each node in the execution plan, the current implementation of EXPLAIN ANALYZE can add considerable profiling overhead to query execution. As a result, running EXPLAIN ANALYZE on a query can sometimes take significantly longer than executing the query normally. The amount of overhead depends on the nature of the query.

Examples

To show the plan for a simple query on a table with a single integer column and 10000 rows:

EXPLAIN SELECT * FROM foo;

QUERY PLAN

---------------------------------------------------------
Seq Scan on foo (cost=0.00..155.00 rows=10000 width=4)
(1 row)

If there is an index and we use a query with an indexable WHERE condition, EXPLAIN might show a different plan:

EXPLAIN SELECT * FROM foo WHERE i = 4;

QUERY PLAN

--------------------------------------------------------------
Index Scan using fi on foo (cost=0.00..5.98 rows=1 width=4)
  Index Cond: (i = 4)
(2 rows)
Here is an example of a query plan for a query using an aggregate function:

```sql
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;
```

**QUERY PLAN**

```
Aggregate (cost=23.93..23.93 rows=1 width=4)
  ->  Index Scan using fi on foo (cost=0.00..23.92 rows=6 width=4)
      Index Cond: (i < 10)
```

Here is an example of using `EXPLAIN EXECUTE` to display the execution plan for a prepared query:

```sql
PREPARE query(int, int) AS SELECT sum(bar) FROM test
  WHERE id > $1 AND id < $2
  GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);
```

**QUERY PLAN**

```
HashAggregate (cost=39.53..39.53 rows=1 width=8) (actual time=0.661..0.672 rows=7 loops=1)
  ->  Index Scan using test_pkey on test (cost=0.00..32.97 rows=1311 width=8)
      Index Cond: ((id > $1) AND (id < $2))
Total runtime: 0.851 ms
```

Of course, the specific numbers shown here depend on the actual contents of the tables involved. Also note that the numbers, and even the selected query strategy, might vary between PostgreSQL releases due to planner improvements. In addition, the `ANALYZE` command uses random sampling to estimate data statistics; therefore, it is possible for cost estimates to change after a fresh run of `ANALYZE`, even if the actual distribution of data in the table has not changed.

**Compatibility**

There is no `EXPLAIN` statement defined in the SQL standard.

**See Also**

`ANALYZE`
FETCH

Name
FETCH — retrieve rows from a query using a cursor

Synopsis
FETCH [ direction { FROM | IN } ] cursorname

where direction can be empty or one of:

NEXT
PRIOR
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
BACKWARD
BACKWARD count
BACKWARD ALL

Description
FETCH retrieves rows using a previously-created cursor.

A cursor has an associated position, which is used by FETCH. The cursor position can be before the first row of the query result, on any particular row of the result, or after the last row of the result. When created, a cursor is positioned before the first row. After fetching some rows, the cursor is positioned on the row most recently retrieved. If FETCH runs off the end of the available rows then the cursor is left positioned after the last row, or before the first row if fetching backward. FETCH ALL or FETCH BACKWARD ALL will always leave the cursor positioned after the last row or before the first row.

The forms NEXT, PRIOR, FIRST, LAST, ABSOLUTE, RELATIVE fetch a single row after moving the cursor appropriately. If there is no such row, an empty result is returned, and the cursor is left positioned before the first row or after the last row as appropriate.

The forms using FORWARD and BACKWARD retrieve the indicated number of rows moving in the forward or backward direction, leaving the cursor positioned on the last-returned row (or after/before all rows, if the count exceeds the number of rows available).

RELATIVE 0, FORWARD 0, and BACKWARD 0 all request fetching the current row without moving the cursor, that is, re-fetching the most recently fetched row. This will succeed unless the cursor is positioned before the first row or after the last row; in which case, no row is returned.
Parameters

direction

direction defines the fetch direction and number of rows to fetch. It can be one of the following:

NEXT
Fetch the next row. This is the default if direction is omitted.

PRIOR
Fetch the prior row.

FIRST
Fetch the first row of the query (same as ABSOLUTE 1).

LAST
Fetch the last row of the query (same as ABSOLUTE -1).

ABSOLUTE count
Fetch the \text{count}^{th} row of the query, or the abs(\text{count})^{th} row from the end if \text{count} is negative. Position before first row or after last row if \text{count} is out of range; in particular, ABSOLUTE 0 positions before the first row.

RELATIVE count
Fetch the \text{count}^{th} succeeding row, or the abs(\text{count})^{th} prior row if \text{count} is negative. RELATIVE 0 re-fetches the current row, if any.

count
Fetch the next \text{count} rows (same as FORWARD \text{count}).

ALL
Fetch all remaining rows (same as FORWARD ALL).

FORWARD
Fetch the next row (same as NEXT).

FORWARD count
Fetch the next \text{count} rows. FORWARD 0 re-fetches the current row.

FORWARD ALL
Fetch all remaining rows.

BACKWARD
Fetch the prior row (same as PRIOR).

BACKWARD count
Fetch the prior \text{count} rows (scanning backwards). BACKWARD 0 re-fetches the current row.

BACKWARD ALL
Fetch all prior rows (scanning backwards).
count

`count` is a possibly-signed integer constant, determining the location or number of rows to fetch. For `FORWARD` and `BACKWARD` cases, specifying a negative `count` is equivalent to changing the sense of `FORWARD` and `BACKWARD`.

cursorname

An open cursor’s name.

Outputs

On successful completion, a `FETCH` command returns a command tag of the form

```
FETCH count
```

The `count` is the number of rows fetched (possibly zero). Note that in psql, the command tag will not actually be displayed, since psql displays the fetched rows instead.

Notes

The cursor should be declared with the `SCROLL` option if one intends to use any variants of `FETCH` other than `FETCH NEXT` or `FETCH FORWARD` with a positive count. For simple queries PostgreSQL will allow backwards fetch from cursors not declared with `SCROLL`, but this behavior is best not relied on. If the cursor is declared with `NO SCROLL`, no backward fetches are allowed.

`ABSOLUTE` fetches are not any faster than navigating to the desired row with a relative move: the underlying implementation must traverse all the intermediate rows anyway. Negative absolute fetches are even worse: the query must be read to the end to find the last row, and then traversed backward from there. However, rewinding to the start of the query (as with `FETCH ABSOLUTE 0`) is fast.

`DECLARE` is used to define a cursor. Use `MOVE` to change cursor position without retrieving data.

Examples

The following example traverses a table using a cursor:

```
BEGIN WORK;

-- Set up a cursor:
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- Fetch the first 5 rows in the cursor liahona:
FETCH FORWARD 5 FROM liahona;

code | title | did | date_prod | kind | len
-------+-------+-----+------------+----------+-------
BL101 | The Third Man | 101 | 1949-12-23 | Drama | 01:44
BL102 | The African Queen | 101 | 1951-08-11 | Romantic | 01:43
JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08
P_302 | Becket | 103 | 1964-02-03 | Drama | 02:28

-- Fetch the previous row:
```
FETCH PRIOR FROM liahona;

<table>
<thead>
<tr>
<th>code</th>
<th>title</th>
<th>did</th>
<th>date_prod</th>
<th>kind</th>
<th>len</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_301</td>
<td>Vertigo</td>
<td>103</td>
<td>1958-11-14</td>
<td>Action</td>
<td>02:08</td>
</tr>
</tbody>
</table>

-- Close the cursor and end the transaction:
CLOSE liahona;
COMMIT WORK;

Compatibility

The SQL standard defines FETCH for use in embedded SQL only. The variant of FETCH described here returns the data as if it were a SELECT result rather than placing it in host variables. Other than this point, FETCH is fully upward-compatible with the SQL standard.

The FETCH forms involving FORWARD and BACKWARD, as well as the forms FETCH count and FETCH ALL, in which FORWARD is implicit, are PostgreSQL extensions.

The SQL standard allows only FROM preceding the cursor name; the option to use IN is an extension.

See Also

CLOSE, DECLARE, MOVE
**GRANT**

**Name**

GRANT — define access privileges

**Synopsis**

GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER } 
[,...] | ALL [ PRIVILEGES ] )
ON [ TABLE ] tablename [,...]
TO { [ GROUP ] rolename | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE } 
[,...] | ALL [ PRIVILEGES ] )
ON SEQUENCE sequencename [,...]
TO { [ GROUP ] rolename | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [,,...] | ALL [ PRIVILEGES ] )
ON DATABASE dbname [,,...]
TO { [ GROUP ] rolename | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] )
ON FUNCTION funcname ( [ [ argmode ] [ argname ] argtype [,,...] ] ) [,,...]
TO { [ GROUP ] rolename | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] )
ON LANGUAGE langname [,,...]
TO { [ GROUP ] rolename | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [,,...] | ALL [ PRIVILEGES ] )
ON SCHEMA schemaname [,,...]
TO { [ GROUP ] rolename | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] )
ON TABLESPACE tablespacename [,,...]
TO { [ GROUP ] rolename | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT role [,,...] TO rolename [,,...] [ WITH ADMIN OPTION ]

**Description**

The **GRANT** command has two basic variants: one that grants privileges on a database object (table, view, sequence, database, function, procedural language, schema, or tablespace), and one that grants membership in a role. These variants are similar in many ways, but they are different enough to be described separately.

As of PostgreSQL 8.1, the concepts of users and groups have been unified into a single kind of entity called a role. It is therefore no longer necessary to use the keyword **GROUP** to identify whether a grantee is a user or a group. **GROUP** is still allowed in the command, but it is a noise word.
GRANT on Database Objects

This variant of the GRANT command gives specific privileges on a database object to one or more roles. These privileges are added to those already granted, if any.

The key word PUBLIC indicates that the privileges are to be granted to all roles, including those that might be created later. PUBLIC can be thought of as an implicitly defined group that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC.

If WITH GRANT OPTION is specified, the recipient of the privilege can in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to PUBLIC.

There is no need to grant privileges to the owner of an object (usually the user that created it), as the owner has all privileges by default. (The owner could, however, choose to revoke some of his own privileges for safety.) The right to drop an object, or to alter its definition in any way is not described by a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. The owner implicitly has all grant options for the object, too.

Depending on the type of object, the initial default privileges might include granting some privileges to PUBLIC. The default is no public access for tables, schemas, and tablespaces; CONNECT privilege and TEMP table creation privilege for databases; EXECUTE privilege for functions; and USAGE privilege for languages. The object owner can of course revoke these privileges. (For maximum security, issue the REVOKE in the same transaction that creates the object; then there is no window in which another user can use the object.)

The possible privileges are:

SELECT

Allows SELECT from any column of the specified table, view, or sequence. Also allows the use of COPY TO. This privilege is also needed to reference existing column values in UPDATE or DELETE. For sequences, this privilege also allows the use of the currval function.

INSERT

Allows INSERT of a new row into the specified table. Also allows COPY FROM.

UPDATE

Allows UPDATE of any column of the specified table. (In practice, any nontrivial UPDATE command will require SELECT privilege as well, since it must reference table columns to determine which rows to update, and/or to compute new values for columns.) SELECT ... FOR UPDATE and SELECT ... FOR SHARE also require this privilege, in addition to the SELECT privilege. For sequences, this privilege allows the use of the nextval and setval functions.

DELETE

Allows DELETE of a row from the specified table. (In practice, any nontrivial DELETE command will require SELECT privilege as well, since it must reference table columns to determine which rows to delete.)

REFERENCES

To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

TRIGGER

Allows the creation of a trigger on the specified table. (See the CREATE TRIGGER statement.)
GRANT

CREATE

For databases, allows new schemas to be created within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have this privilege for the containing schema.

For tablespaces, allows tables, indexes, and temporary files to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)

CONNECT

Allows the user to connect to the specified database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by \texttt{pg\_hba.conf}).

TEMPORARY TEMP

Allows temporary tables to be created while using the specified database.

EXECUTE

Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)

USAGE

For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

For schemas, allows access to objects contained in the specified schema (assuming that the objects’ own privilege requirements are also met). Essentially this allows the grantee to “look up” objects within the schema. Without this permission, it is still possible to see the object names, e.g. by querying the system tables. Also, after revoking this permission, existing backends might have statements that have previously performed this lookup, so this is not a completely secure way to prevent object access.

For sequences, this privilege allows the use of the \texttt{currval} and \texttt{nextval} functions.

ALL PRIVILEGES

Grant all of the available privileges at once. The \texttt{PRIVILEGES} key word is optional in PostgreSQL, though it is required by strict SQL.

The privileges required by other commands are listed on the reference page of the respective command.

GRANT on Roles

This variant of the \texttt{GRANT} command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If \texttt{WITH ADMIN OPTION} is specified, the member can in turn grant membership in the role to others, and revoke membership in the role as well. Without the admin option, ordinary users cannot do that. However, database superusers can grant or revoke membership in any role to anyone. Roles having \texttt{CREATEROLE} privilege can grant or revoke membership in any role that is not a superuser.

Unlike the case with privileges, membership in a role cannot be granted to \texttt{PUBLIC}. Note also that this form of the command does not allow the noise word \texttt{GROUP}.
Notes

The REVOKE command is used to revoke access privileges.

When a non-owner of an object attempts to GRANT privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will grant only those privileges for which the user has grant options. The GRANT ALL PRIVILEGES forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

It should be noted that database superusers can access all objects regardless of object privilege settings. This is comparable to the rights of root in a Unix system. As with root, it’s unwise to operate as a superuser except when absolutely necessary.

If a superuser chooses to issue a GRANT or REVOKE command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will appear to have been granted by the object owner. (For role membership, the membership appears to have been granted by the containing role itself.)

GRANT and REVOKE can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges WITH GRANT OPTION on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges WITH GRANT OPTION. For example, if table t1 is owned by role g1, of which role u1 is a member, then u1 can grant privileges on t1 to u2, but those privileges will appear to have been granted directly by g1. Any other member of role g1 could revoke them later.

If the role executing GRANT holds the required privileges indirectly via more than one role membership path, it is unspecified which containing role will be recorded as having done the grant. In such cases it is best practice to use SET ROLE to become the specific role you want to do the GRANT as.

Granting permission on a table does not automatically extend permissions to any sequences used by the table, including sequences tied to SERIAL columns. Permissions on sequence must be set separately.

Currently, PostgreSQL does not support granting or revoking privileges for individual columns of a table. One possible workaround is to create a view having just the desired columns and then grant privileges to that view.

Use psql’s \z command to obtain information about existing privileges, for example:

```
=> \z mytable

Access privileges for database "lusitania"
Schema | Name   | Type | Access privileges
--------+--------+-------+---------------------------------------------------
public | mytable| table | {miriam=arwdxt/miriam,=r/miriam,admin=arw/miriam}
(1 row)
```

The entries shown by \z are interpreted thus:

- rolename=xxxx -- privileges granted to a role
- =xxxx -- privileges granted to PUBLIC
  - r -- SELECT ("read")
  - w -- UPDATE ("write")
  - a -- INSERT ("append")
  - d -- DELETE

1083
The above example display would be seen by user **miriam** after creating table **mytable** and doing:

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
```

If the “Access privileges” column is empty for a given object, it means the object has default privileges (that is, its privileges column is null). Default privileges always include all privileges for the owner, and can include some privileges for PUBLIC depending on the object type, as explained above. The first GRANT or REVOKE on an object will instantiate the default privileges (producing, for example, `{miriam=arwdxt/miriam}`) and then modify them per the specified request.

Notice that the owner’s implicit grant options are not marked in the access privileges display. **A * will appear only when grant options have been explicitly granted to someone.**

### Examples

Grant insert privilege to all users on table **films**:

```
GRANT INSERT ON films TO PUBLIC;
```

Grant all available privileges to user **manuel** on view **kinds**:

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

Note that while the above will indeed grant all privileges if executed by a superuser or the owner of **kinds**, when executed by someone else it will only grant those permissions for which the someone else has grant options.

Grant membership in role **admins** to user **joe**:

```
GRANT admins TO joe;
```

### Compatibility

According to the SQL standard, the **PRIVILEGES** key word in **ALL PRIVILEGES** is required. The SQL standard does not support setting the privileges on more than one object per command.
PostgreSQL allows an object owner to revoke his own ordinary privileges: for example, a table owner can make the table read-only to himself by revoking his own INSERT, UPDATE, and DELETE privileges. This is not possible according to the SQL standard. The reason is that PostgreSQL treats the owner’s privileges as having been granted by the owner to himself; therefore he can revoke them too. In the SQL standard, the owner’s privileges are granted by an assumed entity "_SYSTEM". Not being "_SYSTEM", the owner cannot revoke these rights.

PostgreSQL does not support the SQL-standard functionality of setting privileges for individual columns.

The SQL standard provides for a USAGE privilege on other kinds of objects: character sets, collations, translations, domains.

Privileges on databases, tablespaces, schemas, and languages are PostgreSQL extensions.

**See Also**

REVOKE
**INSERT**

**Name**

INSERT — create new rows in a table

**Synopsis**

```sql
INSERT INTO table [ ( column [, ...] ) ]
{ DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
[ RETURNING * | output_expression [ AS output_name ] [, ...] ]
```

**Description**

INSERT inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

The target column names can be listed in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order; or the first \( N \) column names, if there are only \( N \) columns supplied by the VALUES clause or query. The values supplied by the VALUES clause or query are associated with the explicit or implicit column list left-to-right.

Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or null if there is none.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

The optional RETURNING clause causes INSERT to compute and return value(s) based on each row actually inserted. This is primarily useful for obtaining values that were supplied by defaults, such as a serial sequence number. However, any expression using the table’s columns is allowed. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have INSERT privilege on a table in order to insert into it, and SELECT privilege on it to use RETURNING. If you use the query clause to insert rows from a query, you also need to have SELECT privilege on any table used in the query.

**Parameters**

- **table**
  The name (optionally schema-qualified) of an existing table.

- **column**
  The name of a column in table. The column name can be qualified with a subfield name or array subscript, if needed. (Inserting into only some fields of a composite column leaves the other fields null.)

- **DEFAULT VALUES**
  All columns will be filled with their default values.
**expression**

An expression or value to assign to the corresponding column.

**DEFAULT**

The corresponding column will be filled with its default value.

**query**

A query (SELECT statement) that supplies the rows to be inserted. Refer to the SELECT statement for a description of the syntax.

**output_expression**

An expression to be computed and returned by the INSERT command after each row is inserted. The expression can use any column names of the table. Write * to return all columns of the inserted row(s).

**output_name**

A name to use for a returned column.

**Outputs**

On successful completion, an INSERT command returns a command tag of the form

\[ \text{INSERT } oid \text{ count} \]

The count is the number of rows inserted. If count is exactly one, and the target table has OIDs, then oid is the OID assigned to the inserted row. Otherwise oid is zero.

If the INSERT command contains a RETURNING clause, the result will be similar to that of a SELECT statement containing the columns and values defined in the RETURNING list, computed over the row(s) inserted by the command.

**Examples**

Insert a single row into table films:

\[
\text{INSERT INTO films VALUES}
\('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
\]

In this example, the len column is omitted and therefore it will have the default value:

\[
\text{INSERT INTO films (code, title, did, date_prod, kind)}
\text{VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');}
\]

This example uses the DEFAULT clause for the date columns rather than specifying a value:

\[
\text{INSERT INTO films VALUES}
\('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
\]

\[
\text{INSERT INTO films (code, title, did, date_prod, kind)}
\text{VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');}
\]
To insert a row consisting entirely of default values:

```
INSERT INTO films DEFAULT VALUES;
```

To insert multiple rows using the multirow VALUES syntax:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

This example inserts some rows into table `films` from a table `tmp_films` with the same column layout as `films`:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

This example inserts into array columns:

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{"","",""},{"","",""},{"","",""}}');
-- The subscripts in the above example aren't really needed
INSERT INTO tictactoe (game, board)
    VALUES (2,('{{X","",""},{"",O,""},{"",X,""}}'));
```

Insert a single row into table `distributors`, returning the sequence number generated by the DEFAULT clause:

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
    RETURNING did;
```

**Compatibility**

`INSERT` conforms to the SQL standard, except that the RETURNING clause is a PostgreSQL extension. Also, the case in which a column name list is omitted, but not all the columns are filled from the VALUES clause or query, is disallowed by the standard.

Possible limitations of the query clause are documented under `SELECT`. 
LISTEN

Name

LISTEN — listen for a notification

Synopsis

LISTEN name

Description

LISTEN registers the current session as a listener on the notification condition name. If the current session is already registered as a listener for this notification condition, nothing is done.

Whenever the command NOTIFY name is invoked, either by this session or another one connected to the same database, all the sessions currently listening on that notification condition are notified, and each will in turn notify its connected client application. See the discussion of NOTIFY for more information.

A session can be unregistered for a given notify condition with the UNLISTEN command. A session’s listen registrations are automatically cleared when the session ends.

The method a client application must use to detect notification events depends on which PostgreSQL application programming interface it uses. With the libpq library, the application issues LISTEN as an ordinary SQL command, and then must periodically call the function PQnotifies to find out whether any notification events have been received. Other interfaces such as libpgtcl provide higher-level methods for handling notify events; indeed, with libpgtcl the application programmer should not even issue LISTEN or UNLISTEN directly. See the documentation for the interface you are using for more details.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Parameters

name

Name of a notify condition (any identifier).

Examples

Configure and execute a listen/notify sequence from psql:

LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
Compatibility

There is no LISTEN statement in the SQL standard.

See Also

NOTIFY, UNLISTEN
LOAD

Name
LOAD — load a shared library file

Synopsis
LOAD 'filename'

Description
This command loads a shared library file into the PostgreSQL server’s address space. If the file has been loaded already, the command does nothing. Shared library files that contain C functions are automatically loaded whenever one of their functions is called. Therefore, an explicit LOAD is usually only needed to load a library that modifies the server’s behavior through “hooks” rather than providing a set of functions.

The file name is specified in the same way as for shared library names in CREATE FUNCTION; in particular, one can rely on a search path and automatic addition of the system’s standard shared library file name extension. See Section 34.9 for more information on this topic.

Non-superusers can only apply LOAD to library files located in $libdir/plugins/ — the specified filename must begin with exactly that string. (It is the database administrator’s responsibility to ensure that only “safe” libraries are installed there.)

Compatibility
LOAD is a PostgreSQL extension.

See Also
CREATE FUNCTION
LOCK

Name

LOCK — lock a table

Synopsis

LOCK [ TABLE ] name [, ...] [ IN lockmode MODE ] [ NOWAIT ]

where lockmode is one of:

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE

Description

LOCK TABLE obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If NOWAIT is specified, LOCK TABLE does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. (There is no UNLOCK TABLE command; locks are always released at transaction end.)

When acquiring locks automatically for commands that reference tables, PostgreSQL always uses the least restrictive lock mode possible. LOCK TABLE provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the Read Committed isolation level and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers, and your LOCK TABLE name IN SHARE MODE statement will wait until any concurrent holders of ROW EXCLUSIVE mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding: furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the Serializable isolation level, you have to execute the LOCK TABLE statement before executing any SELECT or data modification statement. A serializable transaction’s view of data will be frozen when its first SELECT or data modification statement begins. A LOCK TABLE later in the transaction will still prevent concurrent writes — but it won’t ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use SHARE ROW EXCLUSIVE lock mode instead of SHARE mode. This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire SHARE mode, and then be unable to also acquire ROW EXCLUSIVE mode to actually perform their updates. (Note that a transaction’s own locks never conflict, so a transaction can acquire ROW EXCLUSIVE mode when it holds SHARE mode — but not if anyone else holds SHARE mode.) To avoid deadlocks, make sure all transactions acquire locks on the same objects in the same order, and if multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

More information about the lock modes and locking strategies can be found in Section 13.3.
Parameters

name

The name (optionally schema-qualified) of an existing table to lock.

The command \texttt{LOCK TABLE a, b;} is equivalent to \texttt{LOCK TABLE a;} \texttt{LOCK TABLE b;} . The tables are locked one-by-one in the order specified in the \texttt{LOCK TABLE} command.

lockmode

The lock mode specifies which locks this lock conflicts with. Lock modes are described in Section 13.3.

If no lock mode is specified, then \texttt{ACCESS EXCLUSIVE}, the most restrictive mode, is used.

NOWAIT

Specifies that \texttt{LOCK TABLE} should not wait for any conflicting locks to be released: if the specified lock(s) cannot be acquired immediately without waiting, the transaction is aborted.

Notes

\texttt{LOCK TABLE ... IN ACCESS SHARE MODE} requires \texttt{SELECT} privileges on the target table. All other forms of \texttt{LOCK} require \texttt{UPDATE} and/or \texttt{DELETE} privileges.

\texttt{LOCK TABLE} is useful only inside a transaction block (\texttt{BEGIN/COMMIT} pair), since the lock is dropped as soon as the transaction ends. A \texttt{LOCK TABLE} command appearing outside any transaction block forms a self-contained transaction, so the lock will be dropped as soon as it is obtained.

\texttt{LOCK TABLE} only deals with table-level locks, and so the mode names involving \texttt{ROW} are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, \texttt{ROW EXCLUSIVE} mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as \texttt{LOCK TABLE} is concerned, differing only in the rules about which modes conflict with which. For information on how to acquire an actual row-level lock, see Section 13.3.2 and the \texttt{FOR UPDATE/FOR SHARE Clause} in the \texttt{SELECT} reference documentation.

Examples

Obtain a \texttt{SHARE} lock on a primary key table when going to perform inserts into a foreign key table:

\begin{verbatim}
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Do ROLLBACK if record was not returned
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
\end{verbatim}

Take a \texttt{SHARE ROW EXCLUSIVE} lock on a primary key table when going to perform a delete operation:

\begin{verbatim}
BEGIN WORK;
\end{verbatim}
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN 
  (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;

Compatibility

There is no LOCK TABLE in the SQL standard, which instead uses SET TRANSACTION to specify concurrency levels on transactions. PostgreSQL supports that too; see SET TRANSACTION for details.

Except for ACCESS SHARE, ACCESS EXCLUSIVE, and SHARE UPDATE EXCLUSIVE lock modes, the PostgreSQL lock modes and the LOCK TABLE syntax are compatible with those present in Oracle.
MOVE

Name
MOVE — position a cursor

Synopsis
MOVE [ direction { FROM | IN } ] cursorname

Description
MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only positions the cursor and does not return rows.
Refer to FETCH for details on syntax and usage.

Outputs
On successful completion, a MOVE command returns a command tag of the form
MOVE count

The count is the number of rows that a FETCH command with the same parameters would have returned (possibly zero).

Examples
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Skip the first 5 rows:
MOVE FORWARD 5 IN liahona;
MOVE 5

-- Fetch the 6th row from the cursor liahona:
FETCH 1 FROM liahona;

---+-----+----------+------------+--------+-------
code | title | did | date_prod | kind | len
-------------------------
P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- Close the cursor liahona and end the transaction:
CLOSE liahona;
COMMIT WORK;
Compatibility
There is no `MOVE` statement in the SQL standard.

See Also
`CLOSE, DECLARE, FETCH`
NOTIFY

Name
NOTIFY — generate a notification

Synopsis
NOTIFY name

Description
The NOTIFY command sends a notification event to each client application that has previously executed LISTEN name for the specified notification name in the current database.

NOTIFY provides a simple form of signal or interprocess communication mechanism for a collection of processes accessing the same PostgreSQL database. Higher-level mechanisms can be built by using tables in the database to pass additional data (beyond a mere notification name) from notifier to listener(s).

The information passed to the client for a notification event includes the notification name and the notifying session’s server process PID. It is up to the database designer to define the notification names that will be used in a given database and what each one means.

Commonly, the notification name is the same as the name of some table in the database, and the notify event essentially means, “I changed this table, take a look at it to see what’s new”. But no such association is enforced by the NOTIFY and LISTEN commands. For example, a database designer could use several different notification names to signal different sorts of changes to a single table.

When NOTIFY is used to signal the occurrence of changes to a particular table, a useful programming technique is to put the NOTIFY in a rule that is triggered by table updates. In this way, notification happens automatically when the table is changed, and the application programmer cannot accidentally forget to do it.

NOTIFY interacts with SQL transactions in some important ways. Firstly, if a NOTIFY is executed inside a transaction, the notify events are not delivered until and unless the transaction is committed. This is appropriate, since if the transaction is aborted, all the commands within it have had no effect, including NOTIFY. But it can be disconcerting if one is expecting the notification events to be delivered immediately. Secondly, if a listening session receives a notification signal while it is within a transaction, the notification event will not be delivered to its connected client until just after the transaction is completed (either committed or aborted). Again, the reasoning is that if a notification were delivered within a transaction that was later aborted, one would want the notification to be undone somehow — but the server cannot “take back” a notification once it has sent it to the client. So notification events are only delivered between transactions. The upshot of this is that applications using NOTIFY for real-time signaling should try to keep their transactions short.

NOTIFY behaves like Unix signals in one important respect: if the same notification name is signaled multiple times in quick succession, recipients might get only one notification event for several executions of NOTIFY. So it is a bad idea to depend on the number of notifications received. Instead, use NOTIFY to wake up applications that need to pay attention to something, and use a database object (such as a sequence) to keep track of what happened or how many times it happened.
It is common for a client that executes `NOTIFY` to be listening on the same notification name itself. In that case it will get back a notification event, just like all the other listening sessions. Depending on the application logic, this could result in useless work, for example, reading a database table to find the same updates that that session just wrote out. It is possible to avoid such extra work by noticing whether the notifying session’s server process PID (supplied in the notification event message) is the same as one’s own session’s PID (available from libpq). When they are the same, the notification event is one’s own work bouncing back, and can be ignored. (Despite what was said in the preceding paragraph, this is a safe technique. PostgreSQL keeps self-notifications separate from notifications arriving from other sessions, so you cannot miss an outside notification by ignoring your own notifications.)

### Parameters

`name`

Name of the notification to be signaled (any identifier).

### Examples

Configure and execute a listen/notify sequence from psql:

```sql
LISTEN virtual;
NOTIFY virtual;
```

Asynchronous notification "virtual" received from server process with PID 8448.

### Compatibility

There is no `NOTIFY` statement in the SQL standard.

### See Also

`LISTEN, UNLISTEN`
**PREPARE**

**Name**

PREPARE — prepare a statement for execution

**Synopsis**

PREPARE name [ ( datatype [, ...] ) ] AS statement

**Description**

PREPARE creates a prepared statement. A prepared statement is a server-side object that can be used to optimize performance. When the PREPARE statement is executed, the specified statement is parsed, rewritten, and planned. When an EXECUTE command is subsequently issued, the prepared statement need only be executed. Thus, the parsing, rewriting, and planning stages are only performed once, instead of every time the statement is executed.

Prepared statements can take parameters: values that are substituted into the statement when it is executed. When creating the prepared statement, refer to parameters by position, using $1, $2, etc. A corresponding list of parameter data types can optionally be specified. When a parameter’s data type is not specified or is declared as unknown, the type is inferred from the context in which the parameter is used (if possible). When executing the statement, specify the actual values for these parameters in the EXECUTE statement. Refer to EXECUTE for more information about that.

Prepared statements only last for the duration of the current database session. When the session ends, the prepared statement is forgotten, so it must be recreated before being used again. This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create their own prepared statement to use. The prepared statement can be manually cleaned up using the DEALLOCATE command.

Prepared statements have the largest performance advantage when a single session is being used to execute a large number of similar statements. The performance difference will be particularly significant if the statements are complex to plan or rewrite, for example, if the query involves a join of many tables or requires the application of several rules. If the statement is relatively simple to plan and rewrite but relatively expensive to execute, the performance advantage of prepared statements will be less noticeable.

**Parameters**

*name*

An arbitrary name given to this particular prepared statement. It must be unique within a single session and is subsequently used to execute or deallocate a previously prepared statement.

*datatype*

The data type of a parameter to the prepared statement. If the data type of a particular parameter is unspecified or is specified as unknown, it will be inferred from the context in which the parameter is used. To refer to the parameters in the prepared statement itself, use $1, $2, etc.
**PREPARE**

A **PREPARE** statement

Any SELECT, INSERT, UPDATE, DELETE, or VALUES statement.

**Notes**

In some situations, the query plan produced for a prepared statement will be inferior to the query plan that would have been chosen if the statement had been submitted and executed normally. This is because when the statement is planned and the planner attempts to determine the optimal query plan, the actual values of any parameters specified in the statement are unavailable. PostgreSQL collects statistics on the distribution of data in the table, and can use constant values in a statement to make guesses about the likely result of executing the statement. Since this data is unavailable when planning prepared statements with parameters, the chosen plan might be suboptimal. To examine the query plan PostgreSQL has chosen for a prepared statement, use `EXPLAIN`.

For more information on query planning and the statistics collected by PostgreSQL for that purpose, see the `ANALYZE` documentation.

You can see all available prepared statements of a session by querying the `pg_prepared_statements` system view.

**Examples**

Create a prepared statement for an INSERT statement, and then execute it:

```sql
PREPARE fooplan (int, text, bool, numeric) AS
  INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Create a prepared statement for a SELECT statement, and then execute it:

```sql
PREPARE usrrptplan (int) AS
  SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
  AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Note that the data type of the second parameter is not specified, so it is inferred from the context in which `$2` is used.

**Compatibility**

The SQL standard includes a **PREPARE** statement, but it is only for use in embedded SQL. This version of the **PREPARE** statement also uses a somewhat different syntax.

**See Also**

`DEALLOCATE`, `EXECUTE`
**PREPARE TRANSACTION**

**Name**

PREPARE TRANSACTION — prepare the current transaction for two-phase commit

**Synopsis**

PREPARE TRANSACTION transaction_id

**Description**

PREPARE TRANSACTION prepares the current transaction for two-phase commit. After this command, the transaction is no longer associated with the current session; instead, its state is fully stored on disk, and there is a very high probability that it can be committed successfully, even if a database crash occurs before the commit is requested.

Once prepared, a transaction can later be committed or rolled back with COMMIT PREPARED or ROLLBACK PREPARED, respectively. Those commands can be issued from any session, not only the one that executed the original transaction.

From the point of view of the issuing session, PREPARE TRANSACTION is not unlike a ROLLBACK command: after executing it, there is no active current transaction, and the effects of the prepared transaction are no longer visible. (The effects will become visible again if the transaction is committed.)

If the PREPARE TRANSACTION command fails for any reason, it becomes a ROLLBACK: the current transaction is canceled.

**Parameters**

_transaction_id_

An arbitrary identifier that later identifies this transaction for COMMIT PREPARED or ROLLBACK PREPARED. The identifier must be written as a string literal, and must be less than 200 bytes long. It must not be the same as the identifier used for any currently prepared transaction.

**Notes**

This command must be used inside a transaction block. Use BEGIN to start one.

It is not currently allowed to PREPARE a transaction that has executed any operations involving temporary tables, created any cursors WITH HOLD, or executed LISTEN or UNLISTEN. Those features are too tightly tied to the current session to be useful in a transaction to be prepared.

If the transaction modified any run-time parameters with SET (without the LOCAL option), those effects persist after PREPARE TRANSACTION, and will not be affected by any later COMMIT PREPARED or ROLLBACK PREPARED. Thus, in this one respect PREPARE TRANSACTION acts more like COMMIT than ROLLBACK.

All currently available prepared transactions are listed in the pg_prepared_xacts system view.
PREPARE TRANSACTION

From a performance standpoint, it is unwise to leave transactions in the prepared state for a long time: this will for instance interfere with the ability of VACUUM to reclaim storage. Keep in mind also that the transaction continues to hold whatever locks it held. The intended usage of the feature is that a prepared transaction will normally be committed or rolled back as soon as an external transaction manager has verified that other databases are also prepared to commit.

If you make any serious use of prepared transactions, you will probably want to increase the value of max_prepared_transactions, as the default setting is quite small (to avoid wasting resources for those who don’t use it). It is recommendable to make it at least equal to max_connections, so that every session can have a prepared transaction pending.

Examples

Prepare the current transaction for two-phase commit, using foobar as the transaction identifier:

```
PREPARE TRANSACTION 'foobar';
```

See Also

COMMIT PREPARED, ROLLBACK PREPARED
**REASSIGN OWNED**

**Name**
REASSIGN OWNED — change the ownership of database objects owned by a database role

**Synopsis**
REASSIGN OWNED BY old_role [, ...] TO new_role

**Description**
REASSIGN OWNED instructs the system to change the ownership of the database objects owned by one of the old_roles, to new_role.

**Parameters**

*old_role*

The name of a role. The ownership of all the objects in the current database owned by this role will be reassigned to *new_role*.

*new_role*

The name of the role that will be made the new owner of the affected objects.

**Notes**
REASSIGN OWNED is often used to prepare for the removal of one or more roles. Because REASSIGN OWNED only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed. The DROP OWNED command is an alternative that drops all the database objects owned by one or more roles.

The REASSIGN OWNED command does not affect the privileges granted to the old_roles in objects that are not owned by them. Use DROP OWNED to revoke those privileges.

**Compatibility**
The REASSIGN OWNED statement is a PostgreSQL extension.

**See Also**
DROP OWNED, DROP ROLE
REINDEX

Name

REINDEX — rebuild indexes

Synopsis

REINDEX { INDEX | TABLE | DATABASE | SYSTEM } name [ FORCE ]

Description

REINDEX rebuilds an index using the data stored in the index’s table, replacing the old copy of the index. There are several scenarios in which to use REINDEX:

• An index has become corrupted, and no longer contains valid data. Although in theory this should never happen, in practice indexes can become corrupted due to software bugs or hardware failures. REINDEX provides a recovery method.

• An index has become “bloated”, that it is contains many empty or nearly-empty pages. This can occur with B-tree indexes in PostgreSQL under certain uncommon access patterns. REINDEX provides a way to reduce the space consumption of the index by writing a new version of the index without the dead pages. See Section 23.2 for more information.

• You have altered a storage parameter (such as fillfactor) for an index, and wish to ensure that the change has taken full effect.

• An index build with the CONCURRENTLY option failed, leaving an “invalid” index. Such indexes are useless but it can be convenient to use REINDEX to rebuild them. Note that REINDEX will not perform a concurrent build. To build the index without interfering with production you should drop the index and reissue the CREATE INDEX CONCURRENTLY command.

Parameters

INDEX

Recreate the specified index.

TABLE

Recreate all indexes of the specified table. If the table has a secondary “TOAST” table, that is reindexed as well.

DATABASE

Recreate all indexes within the current database. Indexes on shared system catalogs are skipped except in stand-alone mode (see below). This form of REINDEX cannot be executed inside a transaction block.
REINDEX

SYSTEM

Recreate all indexes on system catalogs within the current database. Indexes on user tables are not processed. Also, indexes on shared system catalogs are skipped except in stand-alone mode (see below). This form of REINDEX cannot be executed inside a transaction block.

name

The name of the specific index, table, or database to be reindexed. Index and table names can be schema-qualified. Presently, REINDEX DATABASE and REINDEX SYSTEM can only reindex the current database, so their parameter must match the current database’s name.

FORCE

This is an obsolete option; it is ignored if specified.

Notes

If you suspect corruption of an index on a user table, you can simply rebuild that index, or all indexes on the table, using REINDEX INDEX or REINDEX TABLE.

Things are more difficult if you need to recover from corruption of an index on a system table. In this case it’s important for the system to not have used any of the suspect indexes itself. (Indeed, in this sort of scenario you might find that server processes are crashing immediately at start-up, due to reliance on the corrupted indexes.) To recover safely, the server must be started with the \(-P\) option, which prevents it from using indexes for system catalog lookups.

One way to do this is to shut down the server and start a single-user PostgreSQL server with the \(-P\) option included on its command line. Then, REINDEX DATABASE, REINDEX SYSTEM, REINDEX TABLE, or REINDEX INDEX can be issued, depending on how much you want to reconstruct. If in doubt, use REINDEX SYSTEM to select reconstruction of all system indexes in the database. Then quit the single-user server session and restart the regular server. See the postgres reference page for more information about how to interact with the single-user server interface.

Alternatively, a regular server session can be started with \(-P\) included in its command line options. The method for doing this varies across clients, but in all libpq-based clients, it is possible to set the \(\text{PGOPTIONS}\) environment variable to \(-P\) before starting the client. Note that while this method does not require locking out other clients, it might still be wise to prevent other users from connecting to the damaged database until repairs have been completed.

If corruption is suspected in the indexes of any of the shared system catalogs (which are pg_authid, pg_auth_members, pg_database, pg_pltemplate, pg_shdepend, pg_shdescription, and pg_tablespace), then a standalone server must be used to repair it. REINDEX will not process shared catalogs in multiuser mode.

For all indexes except the shared system catalogs, REINDEX is crash-safe and transaction-safe. REINDEX is not crash-safe for shared indexes, which is why this case is disallowed during normal operation. If a failure occurs while reindexing one of these catalogs in standalone mode, it will not be possible to restart the regular server until the problem is rectified. (The typical symptom of a partially rebuilt shared index is “index is not a btree” errors.)

REINDEX is similar to a drop and recreate of the index in that the index contents are rebuilt from scratch. However, the locking considerations are rather different. REINDEX locks out writes but not reads of the index’s parent table. It also takes an exclusive lock on the specific index being processed, which will block reads that attempt to use that index. In contrast, DROP INDEX momentarily takes exclusive lock on the parent table, blocking both writes and reads. The subsequent CREATE INDEX
locks out writes but not reads; since the index is not there, no read will attempt to use it, meaning that there will be no blocking but reads might be forced into expensive sequential scans.

Reindexing a single index or table requires being the owner of that index or table. Reindexing a database requires being the owner of the database (note that the owner can therefore rebuild indexes of tables owned by other users). Of course, superusers can always reindex anything.

Prior to PostgreSQL 8.1, REINDEX DATABASE processed only system indexes, not all indexes as one would expect from the name. This has been changed to reduce the surprise factor. The old behavior is available as REINDEX SYSTEM.

Prior to PostgreSQL 7.4, REINDEX TABLE did not automatically process TOAST tables, and so those had to be reindexed by separate commands. This is still possible, but redundant.

Examples

Rebuild a single index:

```
REINDEX INDEX my_index;
```

Rebuild all the indexes on the table my_table:

```
REINDEX TABLE my_table;
```

Rebuild all indexes in a particular database, without trusting the system indexes to be valid already:

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

Compatibility

There is no REINDEX command in the SQL standard.
RELEASE SAVEPOINT

Name
RELEASE SAVEPOINT — destroy a previously defined savepoint

Synopsis
RELEASE [ SAVEPOINT ] savepoint_name

Description
RELEASE SAVEPOINT destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. (To do that, see ROLLBACK TO SAVEPOINT.) Destroying a savepoint when it is no longer needed allows the system to reclaim some resources earlier than transaction end.

RELEASE SAVEPOINT also destroys all savepoints that were established after the named savepoint was established.

Parameters
savepoint_name

The name of the savepoint to destroy.

Notes
Specifying a savepoint name that was not previously defined is an error.

It is not possible to release a savepoint when the transaction is in an aborted state.

If multiple savepoints have the same name, only the one that was most recently defined is released.

Examples
To establish and later destroy a savepoint:

BEGIN;
  INSERT INTO table1 VALUES (3);
  SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES (4);
  RELEASE SAVEPOINT my_savepoint;
COMMIT;

The above transaction will insert both 3 and 4.
Compatibility

This command conforms to the SQL standard. The standard specifies that the key word `SAVEPOINT` is mandatory, but PostgreSQL allows it to be omitted.

See Also

`BEGIN`, `COMMIT`, `ROLLBACK`, `ROLLBACK TO SAVEPOINT`, `SAVEPOINT`
RESET

Name
RESET — restore the value of a run-time parameter to the default value

Synopsis
RESET configuration_parameter
RESET ALL

Description
RESET restores run-time parameters to their default values. RESET is an alternative spelling for

SET configuration_parameter TO DEFAULT

Refer to SET for details.

The default value is defined as the value that the parameter would have had, if no SET had ever been
issued for it in the current session. The actual source of this value might be a compiled-in default, the
configuration file, command-line options, or per-database or per-user default settings. This is subtly
different from defining it as “the value that the parameter had at session start”, because if the value
came from the configuration file, it will be reset to whatever is specified by the configuration file now.
See Chapter 18 for details.

The transactional behavior of RESET is the same as SET: its effects will be undone by transaction
rollback.

Parameters

configuration_parameter
Name of a settable run-time parameter. Available parameters are documented in Chapter 18 and
on the SET reference page.

ALL
Resets all settable run-time parameters to default values.

Examples
Set the timezone configuration variable to its default value:

RESET timezone;
Compatibility

RESET is a PostgreSQL extension.

See Also

SET, SHOW
**REVOKE**

**Name**

REVOKE — remove access privileges

**Synopsis**

REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER } 
[,...] | ALL [ PRIVILEGES ] }  
ON [ TABLE ] tablename [, ...]  
FROM { [ GROUP ] rolename | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ { USAGE | SELECT | UPDATE } 
[,...] | ALL [ PRIVILEGES ] }  
ON SEQUENCE sequencename [, ...]  
FROM { [ GROUP ] rolename | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }  
ON DATABASE dbname [, ...]  
FROM { [ GROUP ] rolename | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ EXECUTE | ALL [ PRIVILEGES ] }  
ON FUNCTION funcname ( [ [ argmode ] [ argname ] argtype [, ...] ] ) [, ...]  
FROM { [ GROUP ] rolename | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ USAGE | ALL [ PRIVILEGES ] }  
ON LANGUAGE langname [, ...]  
FROM { [ GROUP ] rolename | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }  
ON SCHEMA schemaname [, ...]  
FROM { [ GROUP ] rolename | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ CREATE | ALL [ PRIVILEGES ] }  
ON TABLESPACE tablespacename [, ...]  
FROM { [ GROUP ] rolename | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
role [, ...] FROM rolename [, ...]  
[ CASCADE | RESTRICT ]
Description

The `REVOKE` command revokes previously granted privileges from one or more roles. The key word `PUBLIC` refers to the implicitly defined group of all roles.

See the description of the `GRANT` command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`. Thus, for example, revoking `SELECT` privilege from `PUBLIC` does not necessarily mean that all roles have lost `SELECT` privilege on the object; those who have it granted directly or via another role will still have it.

If `GRANT OPTION FOR` is specified, only the grant option for the privilege is revoked, not the privilege itself. Otherwise, both the privilege and the grant option are revoked.

If a user holds a privilege with grant option and has granted it to other users then the privileges held by those other users are called dependent privileges. If the privilege or the grant option held by the first user is being revoked and dependent privileges exist, those dependent privileges are also revoked if `CASCADE` is specified; if it is not, the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of users that is traceable to the user that is the subject of this `REVOKE` command. Thus, the affected users might effectively keep the privilege if it was also granted through other users.

When revoking membership in a role, `GRANT OPTION` is instead called `ADMIN OPTION`, but the behavior is similar. Note also that this form of the command does not allow the noise word `GROUP`.

Notes

Use `psql`'s `\z` command to display the privileges granted on existing objects. See `GRANT` for information about the format.

A user can only revoke privileges that were granted directly by that user. If, for example, user A has granted a privilege with grant option to user B, and user B has in turn granted it to user C, then user A cannot revoke the privilege directly from C. Instead, user A could revoke the grant option from user B and use the `CASCADE` option so that the privilege is in turn revoked from user C. For another example, if both A and B have granted the same privilege to C, A can revoke his own grant but not B’s grant, so C will still effectively have the privilege.

When a non-owner of an object attempts to `REVOKE` privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will revoke only those privileges for which the user has grant options. The `REVOKE ALL PRIVILEGES` forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. Since all privileges ultimately come from the object owner (possibly indirectly via chains of grant options), it is possible for a superuser to revoke all privileges, but this might require use of `CASCADE` as stated above.

`REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the command is performed as though it were issued by the containing role that actually owns the object or holds the privileges `WITH GRANT OPTION`. For example, if table `t1` is owned by role `g1`, of which role `u1` is a member, then `u1` can revoke privileges on `t1` that are
REVOKE

recorded as being granted by g1. This would include grants made by u1 as well as by other members of role g1.

If the role executing REVOKE holds privileges indirectly via more than one role membership path, it is unspecified which containing role will be used to perform the command. In such cases it is best practice to use SET ROLE to become the specific role you want to do the REVOKE as. Failure to do so might lead to revoking privileges other than the ones you intended, or not revoking anything at all.

Examples

Revoke insert privilege for the public on table films:

REVOKE INSERT ON films FROM PUBLIC;

Revoke all privileges from user manuel on view kinds:

REVOKE ALL PRIVILEGES ON kinds FROM manuel;

Note that this actually means “revoke all privileges that I granted”.

Revoke membership in role admins from user joe:

REVOKE admins FROM joe;

Compatibility

The compatibility notes of the GRANT command apply analogously to REVOKE. The keyword RESTRICT or CASCADE is required according to the standard, but PostgreSQL assumes RESTRICT by default.

See Also

GRANT
ROLLBACK

Name
ROLLBACK — abort the current transaction

Synopsis
ROLLBACK [ WORK | TRANSACTION ]

Description
ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters
WORK
TRANSACTION
Optional key words. They have no effect.

Notes
Use COMMIT to successfully terminate a transaction.
Issuing ROLLBACK when not inside a transaction does no harm, but it will provoke a warning message.

Examples
To abort all changes:
ROLLBACK;

Compatibility
The SQL standard only specifies the two forms ROLLBACK and ROLLBACK WORK. Otherwise, this command is fully conforming.

See Also
BEGIN, COMMIT, ROLLBACK TO SAVEPOINT
ROLLBACK PREPARED

Name
ROLLBACK PREPARED — cancel a transaction that was earlier prepared for two-phase commit

Synopsis
ROLLBACK PREPARED transaction_id

Description
ROLLBACK PREPARED rolls back a transaction that is in prepared state.

Parameters

transaction_id

The transaction identifier of the transaction that is to be rolled back.

Notes
To roll back a prepared transaction, you must be either the same user that executed the transaction originally, or a superuser. But you do not have to be in the same session that executed the transaction.

This command cannot be executed inside a transaction block. The prepared transaction is rolled back immediately.

All currently available prepared transactions are listed in the pg_prepared_xacts system view.

Examples
Roll back the transaction identified by the transaction identifier foobar:

ROLLBACK PREPARED 'foobar';

See Also
PREPARE TRANSACTION, COMMIT PREPARED
ROLLBACK TO SAVEPOINT

Name

ROLLBACK TO SAVEPOINT — roll back to a savepoint

Synopsis

ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name

Description

Roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

ROLLBACK TO SAVEPOINT implicitly destroys all savepoints that were established after the named savepoint.

Parameters

savepoint_name

The savepoint to roll back to.

Notes

Use RELEASE SAVEPOINT to destroy a savepoint without discarding the effects of commands executed after it was established.

Specifying a savepoint name that has not been established is an error.

Cursors have somewhat non-transactional behavior with respect to savepoints. Any cursor that is opened inside a savepoint will be closed when the savepoint is rolled back. If a previously opened cursor is affected by a FETCH command inside a savepoint that is later rolled back, the cursor position remains at the position that FETCH left it pointing to (that is, FETCH is not rolled back). Closing a cursor is not undone by rolling back, either. A cursor whose execution causes a transaction to abort is put in a cannot-execute state, so while the transaction can be restored using ROLLBACK TO SAVEPOINT, the cursor can no longer be used.

Examples

To undo the effects of the commands executed after my_savepoint was established:

ROLLBACK TO SAVEPOINT my_savepoint;

Cursor positions are not affected by savepoint rollback:
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
?column?
----------
1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
?column?
----------
2

COMMIT;

Compatibility

The SQL standard specifies that the key word SAVEPOINT is mandatory, but PostgreSQL and Oracle allow it to be omitted. SQL allows only WORK, not TRANSACTION, as a noise word after ROLLBACK. Also, SQL has an optional clause AND [ NO ] CHAIN which is not currently supported by PostgreSQL. Otherwise, this command conforms to the SQL standard.

See Also

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT
SAVEPOINT

Name
SAVEPOINT — define a new savepoint within the current transaction

Synopsis
SAVEPOINT savepoint_name

Description
SAVEPOINT establishes a new savepoint within the current transaction.
A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

savepoint_name

The name to give to the new savepoint.

Notes
Use ROLLBACK TO SAVEPOINT to rollback to a savepoint. Use RELEASE SAVEPOINT to destroy a savepoint, keeping the effects of commands executed after it was established.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

Examples
To establish a savepoint and later undo the effects of all commands executed after it was established:

BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;

The above transaction will insert the values 1 and 3, but not 2.
To establish and later destroy a savepoint:

BEGIN;
SAVEPOINT

```
  INSERT INTO table1 VALUES (3);
  SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES (4);
  RELEASE SAVEPOINT my_savepoint;
  COMMIT;
```

The above transaction will insert both 3 and 4.

**Compatibility**

SQL requires a savepoint to be destroyed automatically when another savepoint with the same name is established. In PostgreSQL, the old savepoint is kept, though only the more recent one will be used when rolling back or releasing. (Releasing the newer savepoint will cause the older one to again become accessible to `ROLLBACK TO SAVEPOINT` and `RELEASE SAVEPOINT`.) Otherwise, `SAVEPOINT` is fully SQL conforming.

**See Also**

`BEGIN`, `COMMIT`, `RELEASE SAVEPOINT`, `ROLLBACK`, `ROLLBACK TO SAVEPOINT`
**SELECT**

**Name**

SELECT — retrieve rows from a table or view

**Synopsis**

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ... ] ) ] ]
  * | expression [ AS output_name [ , ... ] ]
  [ FROM from_item [, ... ] ]
  [ WHERE condition ]
  [ GROUP BY expression [, ... ] ]
  [ HAVING condition [, ... ] ]
  [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
  [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ... ]
  [ LIMIT { count | ALL } ]
  [ OFFSET start ]
  [ FOR { UPDATE | SHARE } { OF table_name [, ... ] [ NOWAIT ] [ ... ] ]
```

where `from_item` can be one of:

```
[ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ... ] ) ] ]
( select ) [ AS ] alias [ ( column_alias [, ... ] ) ]
function_name ( [ argument [, ... ] ] ) [ AS ] alias [ ( column_alias [, ... ] ) ]
function_name ( [ argument [, ... ] ] ) AS ( column_definition [, ... ] )
from_item [ NATURAL ] join_type from_item [ ON join_condition | USING { join_column [, ... ] } ]
```

**Description**

`SELECT` retrieves rows from zero or more tables. The general processing of `SELECT` is as follows:

1. All elements in the `FROM` list are computed. (Each element in the `FROM` list is a real or virtual table.) If more than one element is specified in the `FROM` list, they are cross-joined together. (See `FROM Clause` below.)
2. If the `WHERE` clause is specified, all rows that do not satisfy the condition are eliminated from the output. (See `WHERE Clause` below.)
3. If the `GROUP BY` clause is specified, the output is divided into groups of rows that match on one or more values. If the `HAVING` clause is present, it eliminates groups that do not satisfy the given condition. (See `GROUP BY Clause` and `HAVING Clause` below.)
4. The actual output rows are computed using the `SELECT` output expressions for each selected row. (See `SELECT List` below.)
5. Using the operators `UNION`, `INTERSECT`, and `EXCEPT`, the output of more than one `SELECT` statement can be combined to form a single result set. The `UNION` operator returns all rows that are in one or both of the result sets. The `INTERSECT` operator returns all rows that are strictly in both result sets. The `EXCEPT` operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated unless `ALL` is specified. (See `UNION Clause`, `INTERSECT Clause`, and `EXCEPT Clause` below.)
6. If the ORDER BY clause is specified, the returned rows are sorted in the specified order. If ORDER
BY is not given, the rows are returned in whatever order the system finds fastest to produce. (See
ORDER BY Clause below.)

7. DISTINCT eliminates duplicate rows from the result. DISTINCT ON eliminates rows that match
on all the specified expressions. ALL (the default) will return all candidate rows, including duplic-
cates. (See DISTINCT Clause below.)

8. If the LIMIT or OFFSET clause is specified, the SELECT statement only returns a subset of the
result rows. (See LIMIT Clause below.)

9. If FOR UPDATE or FOR SHARE is specified, the SELECT statement locks the selected rows against
concurrent updates. (See FOR UPDATE/FOR SHARE Clause below.)

You must have SELECT privilege on a table to read its values. The use of FOR UPDATE or FOR SHARE
requires UPDATE privilege as well.

Parameters

FROM Clause

The FROM clause specifies one or more source tables for the SELECT. If multiple sources are specified,
the result is the Cartesian product (cross join) of all the sources. But usually qualification conditions
are added to restrict the returned rows to a small subset of the Cartesian product.

The FROM clause can contain the following elements:

table_name

The name (optionally schema-qualified) of an existing table or view. If ONLY is specified before
the table name, only that table is scanned. If ONLY is not specified, the table and all its descendant
tables (if any) are scanned. Optionally, * can be specified after the table name to explicitly
indicate that descendant tables are included.

alias

A substitute name for the FROM item containing the alias. An alias is used for brevity or to
eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an
alias is provided, it completely hides the actual name of the table or function; for example given
FROM foo AS f, the remainder of the SELECT must refer to this FROM item as f not foo. If
an alias is written, a column alias list can also be written to provide substitute names for one or
more columns of the table.

select

A sub-SELECT can appear in the FROM clause. This acts as though its output were created as
a temporary table for the duration of this single SELECT command. Note that the sub-SELECT
must be surrounded by parentheses, and an alias must be provided for it. A VALUES command
can also be used here.

function_name

Function calls can appear in the FROM clause. (This is especially useful for functions that return
result sets, but any function can be used.) This acts as though its output were created as a tempo-
rary table for the duration of this single SELECT command. An alias can also be used. If an alias
is written, a column alias list can also be written to provide substitute names for one or more
attributes of the function’s composite return type. If the function has been defined as returning
the record data type, then an alias or the key word AS must be present, followed by a column
definition list in the form (column_name data_type [, ... ]). The column definition list
must match the actual number and types of columns returned by the function.

\[ \text{JOIN} \]

One of

- [ INNER ] JOIN
- LEFT [ OUTER ] JOIN
- RIGHT [ OUTER ] JOIN
- FULL [ OUTER ] JOIN
- CROSS JOIN

For the INNER and OUTER join types, a join condition must be specified, namely exactly one of
\text{NATURAL}, \text{ON join_condition}, or \text{USING (join_column [, ...])}. See below for the mean-
ing. For CROSS JOIN, none of these clauses can appear.

A JOIN clause combines two FROM items. Use parentheses if necessary to determine the order
of nesting. In the absence of parentheses, JOINs nest left-to-right. In any case JOIN binds more
tightly than the commas separating FROM items.

\text{CROSS JOIN} and \text{INNER JOIN} produce a simple Cartesian product, the same result as you get
from listing the two items at the top level of FROM, but restricted by the join condition (if any).
\text{CROSS JOIN} is equivalent to \text{INNER JOIN ON (TRUE)}, that is, no rows are removed by qual-
ification. These join types are just a notational convenience, since they do nothing you couldn’t
do with plain FROM and WHERE.

\text{LEFT OUTER JOIN} returns all rows in the qualified Cartesian product (i.e., all combined rows
that pass its join condition), plus one copy of each row in the left-hand table for which there was
no right-hand row that passed the join condition. This left-hand row is extended to the full width
of the joined table by inserting null values for the right-hand columns. Note that only the JOIN
clause’s own condition is considered while deciding which rows have matches. Outer conditions
are applied afterwards.

Conversely, \text{RIGHT OUTER JOIN} returns all the joined rows, plus one row for each unmatched
right-hand row (extended with nulls on the left). This is just a notational convenience, since you
could convert it to a LEFT OUTER JOIN by switching the left and right inputs.

\text{FULL OUTER JOIN} returns all the joined rows, plus one row for each unmatched left-hand row
(extended with nulls on the right), plus one row for each unmatched right-hand row (extended
with nulls on the left).

\text{ON join_condition}

\text{join_condition} is an expression resulting in a value of type boolean (similar to a WHERE
clause) that specifies which rows in a join are considered to match.

\text{USING (join_column [, ...])}

A clause of the form \text{USING (a, b, ...)} is shorthand for \text{ON left_table.a =
right_table.a AND left_table.b = right_table.b ...}. Also, \text{USING} implies that
only one of each pair of equivalent columns will be included in the join output, not both.

\text{NATURAL}

\text{NATURAL} is shorthand for a \text{USING} list that mentions all columns in the two tables that have the
same names.
**WHERE Clause**

The optional WHERE clause has the general form

```
WHERE condition
```

where `condition` is any expression that evaluates to a result of type boolean. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns true when the actual row values are substituted for any variable references.

**GROUP BY Clause**

The optional GROUP BY clause has the general form

```
GROUP BY expression [, ...]
```

GROUP BY will condense into a single row all selected rows that share the same values for the grouped expressions. `expression` can be an input column name, or the name or ordinal number of an output column (SELECT list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name will be interpreted as an input-column name rather than an output column name.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group (whereas without GROUP BY, an aggregate produces a single value computed across all the selected rows). When GROUP BY is present, it is not valid for the SELECT list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

**HAVING Clause**

The optional HAVING clause has the general form

```
HAVING condition
```

where `condition` is the same as specified for the WHERE clause.

HAVING eliminates group rows that do not satisfy the condition. HAVING is different from WHERE: WHERE filters individual rows before the application of GROUP BY, while HAVING filters group rows created by GROUP BY. Each column referenced in condition must unambiguously reference a grouping column, unless the reference appears within an aggregate function.

The presence of HAVING turns a query into a grouped query even if there is no GROUP BY clause. This is the same as what happens when the query contains aggregate functions but no GROUP BY clause. All the selected rows are considered to form a single group, and the SELECT list and HAVING clause can only reference table columns from within aggregate functions. Such a query will emit a single row if the HAVING condition is true, zero rows if it is not true.
**SELECT List**

The **SELECT list** (between the key words **SELECT** and **FROM**) specifies expressions that form the output rows of the **SELECT** statement. The expressions can (and usually do) refer to columns computed in the **FROM** clause. Using the clause **AS output_name**, another name can be specified for an output column. This name is primarily used to label the column for display. It can also be used to refer to the column’s value in **ORDER BY** and **GROUP BY** clauses, but not in the **WHERE** or **HAVING** clauses; there you must write out the expression instead.

Instead of an expression, * can be written in the output list as a shorthand for all the columns of the selected rows. Also, one can write `table_name.*` as a shorthand for the columns coming from just that table.

**UNION Clause**

The **UNION** clause has this general form:

```
select_statement UNION [ ALL ] select_statement
```

**select_statement** is any **SELECT** statement without an **ORDER BY**, **LIMIT**, **FOR UPDATE**, or **FOR SHARE** clause. (**ORDER BY** and **LIMIT** can be attached to a subexpression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the **UNION**, not to its right-hand input expression.)

The **UNION** operator computes the set union of the rows returned by the involved **SELECT** statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two **SELECT** statements that represent the direct operands of the **UNION** must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of **UNION** does not contain any duplicate rows unless the **ALL** option is specified. **ALL** prevents elimination of duplicates. (Therefore, **UNION ALL** is usually significantly quicker than **UNION**; use **ALL** when you can.)

Multiple **UNION** operators in the same **SELECT** statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, **FOR UPDATE** and **FOR SHARE** cannot be specified either for a **UNION** result or for any input of a **UNION**.

**INTERSECT Clause**

The **INTERSECT** clause has this general form:

```
select_statement INTERSECT [ ALL ] select_statement
```

**select_statement** is any **SELECT** statement without an **ORDER BY**, **LIMIT**, **FOR UPDATE**, or **FOR SHARE** clause.

The **INTERSECT** operator computes the set intersection of the rows returned by the involved **SELECT** statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of **INTERSECT** does not contain any duplicate rows unless the **ALL** option is specified. With **ALL**, a row that has m duplicates in the left table and n duplicates in the right table will appear min(m,n) times in the result set.
SELECT

Multiple \texttt{INTERSECT} operators in the same \texttt{SELECT} statement are evaluated left to right, unless parentheses dictate otherwise. \texttt{INTERSECT} binds more tightly than \texttt{UNION}. That is, \texttt{A UNION B INTERSECT C} will be read as \texttt{A UNION (B INTERSECT C)}.

Currently, \texttt{FOR UPDATE} and \texttt{FOR SHARE} cannot be specified either for an \texttt{INTERSECT} result or for any input of an \texttt{INTERSECT}.

**EXCEPT Clause**

The \texttt{EXCEPT} clause has this general form:

\[
\texttt{select\_statement \texttt{EXCEPT [ ALL ] select\_statement}}
\]

\texttt{select\_statement} is any \texttt{SELECT} statement without an \texttt{ORDER BY}, \texttt{LIMIT}, \texttt{FOR UPDATE}, or \texttt{FOR SHARE} clause.

The \texttt{EXCEPT} operator computes the set of rows that are in the result of the left \texttt{SELECT} statement but not in the result of the right one.

The result of \texttt{EXCEPT} does not contain any duplicate rows unless the \texttt{ALL} option is specified. With \texttt{ALL}, a row that has \textit{m} duplicates in the left table and \textit{n} duplicates in the right table will appear \texttt{max(m-n,0)} times in the result set.

Multiple \texttt{EXCEPT} operators in the same \texttt{SELECT} statement are evaluated left to right, unless parentheses dictate otherwise. \texttt{EXCEPT} binds at the same level as \texttt{UNION}.

Currently, \texttt{FOR UPDATE} and \texttt{FOR SHARE} cannot be specified either for an \texttt{EXCEPT} result or for any input of an \texttt{EXCEPT}.

**ORDER BY Clause**

The optional \texttt{ORDER BY} clause has this general form:

\[
\texttt{ORDER BY expression \texttt{[ ASC | DESC | USING } operator \texttt{] [ NULLS \texttt{[ FIRST | LAST ]} \texttt{]} [, ...]}}
\]

The \texttt{ORDER BY} clause causes the result rows to be sorted according to the specified expression(s). If two rows are equal according to the leftmost expression, they are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

Each \texttt{expression} can be the name or ordinal number of an output column (\texttt{SELECT} list item), or it can be an arbitrary expression formed from input-column values.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the \texttt{AS} clause.

It is also possible to use arbitrary expressions in the \texttt{ORDER BY} clause, including columns that do not appear in the \texttt{SELECT} result list. Thus the following statement is valid:

\[
\texttt{SELECT name FROM distributors ORDER BY code;}
\]

A limitation of this feature is that an \texttt{ORDER BY} clause applying to the result of a \texttt{UNION}, \texttt{INTERSECT}, or \texttt{EXCEPT} clause can only specify an output column name or number, not an expression.

If an \texttt{ORDER BY} expression is a simple name that matches both a result column name and an input column name, \texttt{ORDER BY} will interpret it as the result column name. This is the opposite of the choice
that GROUP BY will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one can add the key word ASC (ascending) or DESC (descending) after any expression in the ORDER BY clause. If not specified, ASC is assumed by default. Alternatively, a specific ordering operator name can be specified in the USING clause. An ordering operator must be a less-than or greater-than member of some B-tree operator family. ASC is usually equivalent to USING < and DESC is usually equivalent to USING >. (But the creator of a user-defined data type can define exactly what the default sort ordering is, and it might correspond to operators with other names.)

If NULLS LAST is specified, null values sort after all non-null values; if NULLS FIRST is specified, null values sort before all non-null values. If neither is specified, the default behavior is NULLS LAST when ASC is specified or implied, and NULLS FIRST when DESC is specified (thus, the default is to act as though nulls are larger than non-nulls). When USING is specified, the default nulls ordering depends on whether the operator is a less-than or greater-than operator.

Note that ordering options apply only to the expression they follow; for example ORDER BY x, y DESC does not mean the same thing as ORDER BY x DESC, y DESC.

Character-string data is sorted according to the locale-specific collation order that was established when the database cluster was initialized.

**DISTINCT Clause**

If DISTINCT is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). ALL specifies the opposite: all rows are kept; that is the default.

DISTINCT ON ( expression [, ...] ) keeps only the first row of each set of rows where the given expressions evaluate to equal. The DISTINCT ON expressions are interpreted using the same rules as for ORDER BY (see above). Note that the "first row" of each set is unpredictable unless ORDER BY is used to ensure that the desired row appears first. For example:

```sql
SELECT DISTINCT ON (location) location, time, report
  FROM weather_reports
  ORDER BY location, time DESC;
```

retrieves the most recent weather report for each location. But if we had not used ORDER BY to force descending order of time values for each location, we’d have gotten a report from an unpredictable time for each location.

The DISTINCT ON expression(s) must match the leftmost ORDER BY expression(s). The ORDER BY clause will normally contain additional expression(s) that determine the desired precedence of rows within each DISTINCT ON group.

**LIMIT Clause**

The LIMIT clause consists of two independent sub-clauses:

```sql
LIMIT { count | ALL }
OFFSET start
```

count specifies the maximum number of rows to return, while start specifies the number of rows to skip before starting to return rows. When both are specified, start rows are skipped before starting to count the count rows to be returned.
When using `LIMIT`, it is a good idea to use an `ORDER BY` clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query’s rows — you might be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don’t know what ordering unless you specify `ORDER BY`.

The query planner takes `LIMIT` into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you use for `LIMIT` and `OFFSET`. Thus, using different `LIMIT/OFFSET` values to select different subsets of a query result will give inconsistent results unless you enforce a predictable result ordering with `ORDER BY`. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless `ORDER BY` is used to constrain the order.

It is even possible for repeated executions of the same `LIMIT` query to return different subsets of the rows of a table, if there is not an `ORDER BY` to enforce selection of a deterministic subset. Again, this is not a bug; determinism of the results is simply not guaranteed in such a case.

**FOR UPDATE/FOR SHARE Clause**

The `FOR UPDATE` clause has this form:

```
FOR UPDATE [ OF table_name [, ...] ] [ NOWAIT ]
```

The closely related `FOR SHARE` clause has this form:

```
FOR SHARE [ OF table_name [, ...] ] [ NOWAIT ]
```

`FOR UPDATE` causes the rows retrieved by the `SELECT` statement to be locked as though for update. This prevents them from being modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` of these rows will be blocked until the current transaction ends. Also, if an `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` from another transaction has already locked a selected row or rows, `SELECT FOR UPDATE` will wait for the other transaction to complete, and will then lock and return the updated row (or no row, if the row was deleted). For further discussion see Chapter 13.

To prevent the operation from waiting for other transactions to commit, use the `NOWAIT` option. `SELECT FOR UPDATE NOWAIT` reports an error, rather than waiting, if a selected row cannot be locked immediately. Note that `NOWAIT` applies only to the row-level lock(s) — the required `ROW SHARE` table-level lock is still taken in the ordinary way (see Chapter 13). You can use the `NOWAIT` option of `LOCK` if you need to acquire the table-level lock without waiting.

`FOR SHARE` behaves similarly, except that it acquires a shared rather than exclusive lock on each retrieved row. A shared lock blocks other transactions from performing `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` on these rows, but it does not prevent them from performing `SELECT FOR SHARE`.

If specific tables are named in `FOR UPDATE` or `FOR SHARE`, then only rows coming from those tables are locked; any other tables used in the `SELECT` are simply read as usual. A `FOR UPDATE` or `FOR SHARE` clause without a table list affects all tables used in the command. If `FOR UPDATE` or `FOR SHARE` is applied to a view or sub-query, it affects all tables used in the view or sub-query.

Multiple `FOR UPDATE` and `FOR SHARE` clauses can be written if it is necessary to specify different locking behavior for different tables. If the same table is mentioned (or implicitly affected) by both `FOR UPDATE` and `FOR SHARE` clauses, then it is processed as `FOR UPDATE`. Similarly, a table is processed as `NOWAIT` if that is specified in any of the clauses affecting it.
FOR UPDATE and FOR SHARE cannot be used in contexts where returned rows cannot be clearly identified with individual table rows; for example they cannot be used with aggregation.

Caution
Avoid locking a row and then modifying it within a later savepoint or PL/pgSQL exception block. A subsequent rollback would cause the lock to be lost. For example:

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

After the ROLLBACK, the row is effectively unlocked, rather than returned to its pre-savepoint state of being locked but not modified. This hazard occurs if a row locked in the current transaction is updated or deleted, or if a shared lock is upgraded to exclusive: in all these cases, the former lock state is forgotten. If the transaction is then rolled back to a state between the original locking command and the subsequent change, the row will appear not to be locked at all. This is an implementation deficiency which will be addressed in a future release of PostgreSQL.

Caution
It is possible for a SELECT command using both LIMIT and FOR UPDATE/SHARE clauses to return fewer rows than specified by LIMIT. This is because LIMIT is applied first. The command selects the specified number of rows, but might then block trying to obtain lock on one or more of them. Once the SELECT unblocks, the row might have been deleted or updated so that it does not meet the query WHERE condition anymore, in which case it will not be returned.

Caution
Similarly, it is possible for a SELECT command using ORDER BY and FOR UPDATE/SHARE to return rows out of order. This is because ORDER BY is applied first. The command orders the result, but might then block trying to obtain a lock on one or more of the rows. Once the SELECT unblocks, one of the ordered columns might have been modified and be returned out of order. A workaround is to perform SELECT ... FOR UPDATE/SHARE and then SELECT ... ORDER BY.

Examples
To join the table films with the table distributors:

```sql
SELECT f.title, f.did, d.name, f.date_prod, f.kind
FROM distributors d, films f
WHERE f.did = d.did
```

<table>
<thead>
<tr>
<th>title</th>
<th>did</th>
<th>name</th>
<th>date_prod</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1128</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Third Man | 101 | British Lion | 1949-12-23 | Drama
The African Queen | 101 | British Lion | 1951-08-11 | Romantic
...

To sum the column `len` of all films and group the results by `kind`:

```sql
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```

<table>
<thead>
<tr>
<th>kind</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>07:34</td>
</tr>
<tr>
<td>Comedy</td>
<td>02:58</td>
</tr>
<tr>
<td>Drama</td>
<td>14:28</td>
</tr>
<tr>
<td>Musical</td>
<td>06:42</td>
</tr>
<tr>
<td>Romantic</td>
<td>04:38</td>
</tr>
</tbody>
</table>

To sum the column `len` of all films, group the results by `kind` and show those group totals that are less than 5 hours:

```sql
SELECT kind, sum(len) AS total
FROM films
GROUP BY kind
HAVING sum(len) < interval '5 hours';
```

<table>
<thead>
<tr>
<th>kind</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comedy</td>
<td>02:58</td>
</tr>
<tr>
<td>Romantic</td>
<td>04:38</td>
</tr>
</tbody>
</table>

The following two examples are identical ways of sorting the individual results according to the contents of the second column (name):

```sql
SELECT • FROM distributors ORDER BY name;
SELECT • FROM distributors ORDER BY 2;
```

<table>
<thead>
<tr>
<th>did</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>109</td>
<td>20th Century Fox</td>
</tr>
<tr>
<td>110</td>
<td>Bavaria Atelier</td>
</tr>
<tr>
<td>101</td>
<td>British Lion</td>
</tr>
<tr>
<td>107</td>
<td>Columbia</td>
</tr>
<tr>
<td>102</td>
<td>Jean Luc Godard</td>
</tr>
<tr>
<td>113</td>
<td>Luso films</td>
</tr>
<tr>
<td>104</td>
<td>Mosfilm</td>
</tr>
<tr>
<td>103</td>
<td>Paramount</td>
</tr>
<tr>
<td>106</td>
<td>Toho</td>
</tr>
<tr>
<td>105</td>
<td>United Artists</td>
</tr>
<tr>
<td>111</td>
<td>Walt Disney</td>
</tr>
<tr>
<td>112</td>
<td>Warner Bros.</td>
</tr>
<tr>
<td>108</td>
<td>Westward</td>
</tr>
</tbody>
</table>
The next example shows how to obtain the union of the tables `distributors` and `actors`, restricting the results to those that begin with the letter W in each table. Only distinct rows are wanted, so the key word `ALL` is omitted.

```
<table>
<thead>
<tr>
<th>did</th>
<th>name id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>Westward</td>
<td>Woody Allen</td>
</tr>
<tr>
<td>111</td>
<td>Walt Disney</td>
<td>Warren Beatty</td>
</tr>
<tr>
<td>112</td>
<td>Warner Bros.</td>
<td>Walter Matthau</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

```
SELECT distributors.name
  FROM distributors
  WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
  FROM actors
  WHERE actors.name LIKE 'W%';
```

This example shows how to use a function in the `FROM` clause, both with and without a column definition list:

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
  SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distributors(111);
```

```
<table>
<thead>
<tr>
<th>did</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Walt Disney</td>
</tr>
</tbody>
</table>
```

```
CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
  SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
```

```
<table>
<thead>
<tr>
<th>f1</th>
<th>f2</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Walt Disney</td>
</tr>
</tbody>
</table>
```
Compatibility

Of course, the `SELECT` statement is compatible with the SQL standard. But there are some extensions and some missing features.

Omitted FROM Clauses

PostgreSQL allows one to omit the `FROM` clause. It has a straightforward use to compute the results of simple expressions:

```sql
SELECT 2 + 2;
```

```
?column?
----------
   4
```

Some other SQL databases cannot do this except by introducing a dummy one-row table from which to do the `SELECT`.

Note that if a `FROM` clause is not specified, the query cannot reference any database tables. For example, the following query is invalid:

```sql
SELECT distributors.* WHERE distributors.name = 'Westward';
```

PostgreSQL releases prior to 8.1 would accept queries of this form, and add an implicit entry to the query’s `FROM` clause for each table referenced by the query. This is no longer the default behavior, because it does not comply with the SQL standard, and is considered by many to be error-prone. For compatibility with applications that rely on this behavior the add_missing_from configuration variable can be enabled.

The as Key Word

In the SQL standard, the optional key word `AS` is just noise and can be omitted without affecting the meaning. The PostgreSQL parser requires this key word when renaming output columns because the type extensibility features lead to parsing ambiguities without it. `AS` is optional in `FROM` items, however.

Namespace Available to GROUP BY and ORDER BY

In the SQL-92 standard, an `ORDER BY` clause can only use result column names or numbers, while a `GROUP BY` clause can only use expressions based on input column names. PostgreSQL extends each of these clauses to allow the other choice as well (but it uses the standard’s interpretation if there is ambiguity). PostgreSQL also allows both clauses to specify arbitrary expressions. Note that names appearing in an expression will always be taken as input-column names, not as result-column names.

SQL:1999 and later use a slightly different definition which is not entirely upward compatible with SQL-92. In most cases, however, PostgreSQL will interpret an `ORDER BY` or `GROUP BY` expression the same way SQL:1999 does.

Nonstandard Clauses

The clauses `DISTINCT ON, LIMIT, and OFFSET` are not defined in the SQL standard.
SELECT INTO

Name

SELECT INTO — define a new table from the results of a query

Synopsis

SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
   * | expression [ AS output_name ] [, ...]
   INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table
   [ FROM from_item [, ...] ]
   [ WHERE condition ]
   [ GROUP BY expression [, ...] ]
   [ HAVING condition [, ...] ]
   [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
   [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ... ]]
   [ LIMIT { count | ALL } ]
   [ OFFSET start ]
   [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [ ... ] ]

Description

SELECT INTO creates a new table and fills it with data computed by a query. The data is not returned to the client, as it is with a normal SELECT. The new table’s columns have the names and data types associated with the output columns of the SELECT.

Parameters

TEMPORARY or TEMP

If specified, the table is created as a temporary table. Refer to CREATE TABLE for details.

new_table

The name (optionally schema-qualified) of the table to be created.

All other parameters are described in detail under SELECT.

Notes

CREATE TABLE AS is functionally similar to SELECT INTO. CREATE TABLE AS is the recommended syntax, since this form of SELECT INTO is not available in ECPG or PL/pgSQL, because they interpret the INTO clause differently. Furthermore, CREATE TABLE AS offers a superset of the functionality provided by SELECT INTO.

Prior to PostgreSQL 8.1, the table created by SELECT INTO included OIDs by default. In PostgreSQL 8.1, this is not the case — to include OIDs in the new table, the default_with_oids configuration variable must be enabled. Alternatively, CREATE TABLE AS can be used with the WITH OIDS clause.
Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```sql
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

Compatibility

The SQL standard uses `SELECT INTO` to represent selecting values into scalar variables of a host program, rather than creating a new table. This indeed is the usage found in ECPG (see Chapter 32) and PL/pgSQL (see Chapter 38). The PostgreSQL usage of `SELECT INTO` to represent table creation is historical. It is best to use `CREATE TABLE AS` for this purpose in new code.

See Also

`CREATE TABLE AS`
SET

Name
SET — change a run-time parameter

Synopsis
SET [ SESSION | LOCAL ] configuration_parameter { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }

Description
The SET command changes run-time configuration parameters. Many of the run-time parameters listed in Chapter 18 can be changed on-the-fly with SET. (But some require superuser privileges to change, and others cannot be changed after server or session start.) SET only affects the value used by the current session.

If SET (or equivalently SET SESSION) is issued within a transaction that is later aborted, the effects of the SET command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another SET.

The effects of SET LOCAL last only till the end of the current transaction, whether committed or not. A special case is SET followed by SET LOCAL within a single transaction: the SET LOCAL value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the SET value will take effect.

The effects of SET or SET LOCAL are also canceled by rolling back to a savepoint that is earlier than the command.

If SET LOCAL is used within a function that has a SET option for the same variable (see CREATE FUNCTION), the effects of the SET LOCAL command disappear at function exit; that is, the value in effect when the function was called is restored anyway. This allows SET LOCAL to be used for dynamic or repeated changes of a parameter within a function, while still having the convenience of using the SET option to save and restore the caller’s value. However, a regular SET command overrides any surrounding function’s SET option; its effects will persist unless rolled back.

Note: In PostgreSQL versions 8.0 through 8.2, the effects of a SET LOCAL would be canceled by releasing an earlier savepoint, or by successful exit from a PL/pgSQL exception block. This behavior has been changed because it was deemed unintuitive.

Parameters
SESSION

Specifies that the command takes effect for the current session. (This is the default if neither SESSION nor LOCAL appears.)
SET LOCAL

Specifies that the command takes effect for only the current transaction. After COMMIT or ROLLBACK, the session-level setting takes effect again. Note that SET LOCAL will appear to have no effect if it is executed outside a BEGIN block, since the transaction will end immediately.

configuration_parameter

Name of a settable run-time parameter. Available parameters are documented in Chapter 18 and below.

value

New value of parameter. Values can be specified as string constants, identifiers, numbers, or comma-separated lists of these, as appropriate for the particular parameter. DEFAULT can be written to specify resetting the parameter to its default value (that is, whatever value it would have had if no SET had been executed in the current session).

Besides the configuration parameters documented in Chapter 18, there are a few that can only be adjusted using the SET command or that have a special syntax:

NAMES

SET NAMES value is an alias for SET client_encoding TO value.

SEED

Sets the internal seed for the random number generator (the function random). Allowed values are floating-point numbers between 0 and 1, which are then multiplied by $2^{31}$-1.

The seed can also be set by invoking the function setseed:

SELECT setseed(value);

TIME ZONE

SET TIME ZONE value is an alias for SET timezone TO value. The syntax SET TIME ZONE allows special syntax for the time zone specification. Here are examples of valid values:

'PST8PDT'

The time zone for Berkeley, California.

'Europe/Rome'

The time zone for Italy.

-7

The time zone 7 hours west from UTC (equivalent to PDT). Positive values are east from UTC.

INTERVAL '-08:00' HOUR TO MINUTE

The time zone 8 hours west from UTC (equivalent to PST).

LOCAL

DEFAULT

Set the time zone to your local time zone (that is, the server’s default value of timezone; if this has not been explicitly set anywhere, it will be the zone that the server’s operating system defaults to).

See Section 8.5.3 for more information about time zones.
Notes
The function set_config provides equivalent functionality; see Section 9.23. Also, it is possible to
UPDATE the pg_settings system view to perform the equivalent of SET.

Examples
Set the schema search path:

```
SET search_path TO my_schema, public;
```

Set the style of date to traditional POSTGRES with “day before month” input convention:

```
SET datestyle TO postgres, dmy;
```

Set the time zone for Berkeley, California:

```
SET TIME ZONE 'PST8PDT';
```

Set the time zone for Italy:

```
SET TIME ZONE 'Europe/Rome';
```

Compatibility
SET TIME ZONE extends syntax defined in the SQL standard. The standard allows only numeric time
zone offsets while PostgreSQL allows more flexible time-zone specifications. All other SET features
are PostgreSQL extensions.

See Also

RESET, SHOW
SET CONSTRAINTS

Name

SET CONSTRAINTS — set constraint checking modes for the current transaction

Synopsis

SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }

Description

SET CONSTRAINTS sets the behavior of constraint checking within the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are not checked until transaction commit. Each constraint has its own IMMEDIATE or DEFERRED mode. Upon creation, a constraint is given one of three characteristics: DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE, or NOT DEFERRABLE. The third class is always IMMEDIATE and is not affected by the SET CONSTRAINTS command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by SET CONSTRAINTS. SET CONSTRAINTS with a list of constraint names changes the mode of just those constraints (which must all be deferrable). The current schema search path is used to find the first matching name if no schema name is specified. SET CONSTRAINTS ALL changes the mode of all deferrable constraints. When SET CONSTRAINTS changes the mode of a constraint from DEFERRED to IMMEDIATE, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the SET CONSTRAINTS command. If any such constraint is violated, the SET CONSTRAINTS fails (and does not change the constraint mode). Thus, SET CONSTRAINTS can be used to force checking of constraints to occur at a specific point in a transaction. Currently, only foreign key constraints are affected by this setting. Check and unique constraints are always effectively not deferrable. Triggers that are declared as “constraint triggers” are also affected.

Notes

This command only alters the behavior of constraints within the current transaction. Thus, if you execute this command outside of a transaction block (BEGIN/COMMIT pair), it will not appear to have any effect.

Compatibility

This command complies with the behavior defined in the SQL standard, except for the limitation that, in PostgreSQL, it only applies to foreign-key constraints.


**SET ROLE**

**Name**

SET ROLE — set the current user identifier of the current session

**Synopsis**

SET [ SESSION | LOCAL ] ROLE rolename  
SET [ SESSION | LOCAL ] ROLE NONE  
RESET ROLE

**Description**

This command sets the current user identifier of the current SQL session to be rolename. The role name can be written as either an identifier or a string literal. After SET ROLE, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified rolename must be a role that the current session user is a member of. (If the session user is a superuser, any role can be selected.)

The SESSION and LOCAL modifiers act the same as for the regular SET command.

The NONE and RESET forms reset the current user identifier to be the current session user identifier. These forms can be executed by any user.

**Notes**

Using this command, it is possible to either add privileges or restrict one’s privileges. If the session user role has the INHERITS attribute, then it automatically has all the privileges of every role that it could SET ROLE to; in this case SET ROLE effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving only the privileges available to the named role. On the other hand, if the session user role has the NOINHERITS attribute, SET ROLE drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role.

In particular, when a superuser chooses to SET ROLE to a non-superuser role, she loses her superuser privileges.

SET ROLE has effects comparable to SET SESSION AUTHORIZATION, but the privilege checks involved are quite different. Also, SET SESSION AUTHORIZATION determines which roles are allowable for later SET ROLE commands, whereas changing roles with SET ROLE does not change the set of roles allowed to a later SET ROLE.

SET ROLE cannot be used within a SECURITY DEFINER function.

**Examples**

SELECT SESSION_USER, CURRENT_USER;

<table>
<thead>
<tr>
<th>session_user</th>
<th>current_user</th>
</tr>
</thead>
<tbody>
<tr>
<td>1138</td>
<td></td>
</tr>
</tbody>
</table>
SET ROLE

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>peter</td>
<td>peter</td>
</tr>
</tbody>
</table>

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>session_user</td>
<td>current_user</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>peter</td>
<td>paul</td>
</tr>
</tbody>
</table>

Compatibility

PostgreSQL allows identifier syntax ("rolename"), while the SQL standard requires the role name to be written as a string literal. SQL does not allow this command during a transaction; PostgreSQL does not make this restriction because there is no reason to. The SESSION and LOCAL modifiers are a PostgreSQL extension, as is the RESET syntax.

See Also

SET SESSION AUTHORIZATION
SET SESSION AUTHORIZATION

Name

SET SESSION AUTHORIZATION — set the session user identifier and the current user identifier of the current session

Synopsis

SET [ SESSION | LOCAL ] SESSION AUTHORIZATION username
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION

Description

This command sets the session user identifier and the current user identifier of the current SQL session to be username. The user name can be written as either an identifier or a string literal. Using this command, it is possible, for example, to temporarily become an unprivileged user and later switch back to being a superuser.

The session user identifier is initially set to be the (possibly authenticated) user name provided by the client. The current user identifier is normally equal to the session user identifier, but might change temporarily in the context of SECURITY DEFINER functions and similar mechanisms; it can also be changed by SET ROLE. The current user identifier is relevant for permission checking.

The session user identifier can be changed only if the initial session user (the authenticated user) had the superuser privilege. Otherwise, the command is accepted only if it specifies the authenticated user name.

The SESSION and LOCAL modifiers act the same as for the regular SET command.

The DEFAULT and RESET forms reset the session and current user identifiers to be the originally authenticated user name. These forms can be executed by any user.

Notes

SET SESSION AUTHORIZATION cannot be used within a SECURITY DEFINER function.

Examples

SELECT SESSION_USER, CURRENT_USER;

<table>
<thead>
<tr>
<th>session_user</th>
<th>current_user</th>
</tr>
</thead>
<tbody>
<tr>
<td>peter</td>
<td>peter</td>
</tr>
</tbody>
</table>

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;
session_user | current_user
--------------+--------------
paul | paul

Compatibility

The SQL standard allows some other expressions to appear in place of the literal `username`, but these options are not important in practice. PostgreSQL allows identifier syntax ("username"), which SQL does not. SQL does not allow this command during a transaction; PostgreSQL does not make this restriction because there is no reason to. The `SESSION` and `LOCAL` modifiers are a PostgreSQL extension, as is the `RESET` syntax.

The privileges necessary to execute this command are left implementation-defined by the standard.

See Also

`SET ROLE`
SET TRANSACTION

Name

SET TRANSACTION — set the characteristics of the current transaction

Synopsis

SET TRANSACTION transaction_mode [, ...]
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]

where transaction_mode is one of:

   ISOLATION LEVEL | SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
   READ WRITE | READ ONLY

Description

The `SET TRANSACTION` command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. `SET SESSION CHARACTERISTICS` sets the default transaction characteristics for subsequent transactions of a session. These defaults can be overridden by `SET TRANSACTION` for an individual transaction.

The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only).

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

READ COMMITTED

A statement can only see rows committed before it began. This is the default.

SERIALIZABLE

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

The SQL standard defines two additional levels, READ UNCOMMITTED and REPEATABLE READ. In PostgreSQL READ UNCOMMITTED is treated as READ COMMITTED, while REPEATABLE READ is treated as SERIALIZABLE.

The transaction isolation level cannot be changed after the first query or data-modification statement (SELECT, INSERT, DELETE, UPDATE, FETCH, or COPY) of a transaction has been executed. See Chapter 13 for more information about transaction isolation and concurrency control.

The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: INSERT, UPDATE, DELETE, and COPY FROM if the table they would write to is not a temporary table; all CREATE, ALTER, and DROP commands; COMMENT, GRANT, REVOKE, TRUNCATE; and EXPLAIN ANALYZE and EXECUTE if the command they would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.
Notes

If `SET TRANSACTION` is executed without a prior `START TRANSACTION` or `BEGIN`, it will appear to have no effect, since the transaction will immediately end.

It is possible to dispense with `SET TRANSACTION` by instead specifying the desired `transaction_modes` in `BEGIN` or `START TRANSACTION`.

The session default transaction modes can also be set by setting the configuration parameters `default_transaction_isolation` and `default_transaction_read_only`. (In fact `SET SESSION CHARACTERISTICS` is just a verbose equivalent for setting these variables with `SET`.) This means the defaults can be set in the configuration file, via `ALTER DATABASE`, etc. Consult Chapter 18 for more information.

Compatibility

Both commands are defined in the SQL standard. `SERIALIZABLE` is the default transaction isolation level in the standard. In PostgreSQL the default is ordinarily `READ COMMITTED`, but you can change it as mentioned above. Because of lack of predicate locking, the `SERIALIZABLE` level is not truly serializable. See Chapter 13 for details.

In the SQL standard, there is one other transaction characteristic that can be set with these commands: the size of the diagnostics area. This concept is specific to embedded SQL, and therefore is not implemented in the PostgreSQL server.

The SQL standard requires commas between successive `transaction_modes`, but for historical reasons PostgreSQL allows the commas to be omitted.
SHOW

Name
SHOW — show the value of a run-time parameter

Synopsis
SHOW name
SHOW ALL

Description
SHOW will display the current setting of run-time parameters. These variables can be set using the
SET statement, by editing the postgresql.conf configuration file, through the PGOPTIONS envi-
ronmental variable (when using libpq or a libpq-based application), or through command-line flags
when starting the postgres server. See Chapter 18 for details.

Parameters

name
The name of a run-time parameter. Available parameters are documented in Chapter 18 and on
the SET reference page. In addition, there are a few parameters that can be shown but not set:

SERVER_VERSION
Shows the server’s version number.

SERVER_ENCODING
Shows the server-side character set encoding. At present, this parameter can be shown but
not set, because the encoding is determined at database creation time.

LC_COLLATE
Shows the database’s locale setting for collation (text ordering). At present, this parameter
can be shown but not set, because the setting is determined at initdb time.

LC_CTYPE
Shows the database’s locale setting for character classification. At present, this parameter
can be shown but not set, because the setting is determined at initdb time.

IS_SUPERUSER
True if the current role has superuser privileges.

ALL
Show the values of all configuration parameters, with descriptions.
Notes

The function current_setting produces equivalent output; see Section 9.23. Also, the pg_settings system view produces the same information.

Examples

Show the current setting of the parameter DateStyle:

SHOW DateStyle;
DateStyle
---------
ISO, MDY
(1 row)

Show the current setting of the parameter geqo:

SHOW geqo;
geqo
------
on
(1 row)

Show all settings:

SHOW ALL;

<table>
<thead>
<tr>
<th>name</th>
<th>setting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_missing_from</td>
<td>off</td>
<td>Automatically adds missing table references.</td>
</tr>
<tr>
<td>allow_system_table_mods</td>
<td>off</td>
<td>Allows modifications to system tables.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>work_mem</td>
<td>1024</td>
<td>Sets the maximum memory to be used for query workspaces.</td>
</tr>
<tr>
<td>zero_damaged_pages</td>
<td>off</td>
<td>Continues processing past damaged page headers.</td>
</tr>
</tbody>
</table>

(146 rows)

Compatibility

The SHOW command is a PostgreSQL extension.

See Also

SET, RESET
START TRANSACTION

Name
START TRANSACTION — start a transaction block

Synopsis
START TRANSACTION [ transaction_mode [, ...] ]

where transaction_mode is one of:

- ISOLATION LEVEL | SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED
- READ WRITE | READ ONLY

Description
This command begins a new transaction block. If the isolation level or read/write mode is specified, the new transaction has those characteristics, as if SET TRANSACTION was executed. This is the same as the BEGIN command.

Parameters
Refer to SET TRANSACTION for information on the meaning of the parameters to this statement.

Compatibility
In the standard, it is not necessary to issue START TRANSACTION to start a transaction block: any SQL command implicitly begins a block. PostgreSQL’s behavior can be seen as implicitly issuing a COMMIT after each command that does not follow START TRANSACTION (or BEGIN), and it is therefore often called “autocommit”. Other relational database systems might offer an autocommit feature as a convenience.

The SQL standard requires commas between successive transaction_modes, but for historical reasons PostgreSQL allows the commas to be omitted.

See also the compatibility section of SET TRANSACTION.

See Also
BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION
Name

TRUNCATE — empty a table or set of tables

Synopsis

TRUNCATE [ TABLE ] name [, ...] [ CASCADE | RESTRICT ]

Description

TRUNCATE quickly removes all rows from a set of tables. It has the same effect as an unqualified DELETE on each table, but since it does not actually scan the tables it is faster. Furthermore, it reclaims disk space immediately, rather than requiring a subsequent VACUUM operation. This is most useful on large tables.

Parameters

name

The name (optionally schema-qualified) of a table to be truncated.

CASCADE

Automatically truncate all tables that have foreign-key references to any of the named tables, or to any tables added to the group due to CASCADE.

RESTRICT

Refuse to truncate if any of the tables have foreign-key references from tables that are not to be truncated. This is the default.

Notes

Only the owner of a table can TRUNCATE it.

TRUNCATE cannot be used on a table that has foreign-key references from other tables, unless all such tables are also truncated in the same command. Checking validity in such cases would require table scans, and the whole point is not to do one. The CASCADE option can be used to automatically include all dependent tables — but be very careful when using this option, or else you might lose data you did not intend to!

TRUNCATE will not run any ON DELETE triggers that might exist for the tables.
Warning

TRUNCATE is not MVCC-safe (see Chapter 13 for general information about MVCC). After truncation, the table will appear empty to all concurrent transactions, even if they are using a snapshot taken before the truncation occurred. This will only be an issue for a transaction that did not access the truncated table before the truncation happened — any transaction that has done so would hold at least an ACCESS SHARE lock, which would block TRUNCATE until that transaction completes. So truncation will not cause any apparent inconsistency in the table contents for successive queries on the same table, but it could cause visible inconsistency between the contents of the truncated table and other tables in the database.

TRUNCATE is transaction-safe, however: the truncation will be safely rolled back if the surrounding transaction does not commit.

Examples

Truncate the tables bigtable and fattable:

TRUNCATE bigtable, fattable;

Truncate the table othertable, and cascade to any tables that reference othertable via foreign-key constraints:

TRUNCATE othertable CASCADE;

Compatibility

There is no TRUNCATE command in the SQL standard.
UNLISTEN

Name
UNLISTEN — stop listening for a notification

Synopsis
UNLISTEN { name | * }

Description
UNLISTEN is used to remove an existing registration for NOTIFY events. UNLISTEN cancels any existing registration of the current PostgreSQL session as a listener on the notification name. The special wildcard * cancels all listener registrations for the current session.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Parameters
name
Name of a notification (any identifier).

* All current listen registrations for this session are cleared.

Notes
You can unlisten something you were not listening for; no warning or error will appear.
At the end of each session, UNLISTEN * is automatically executed.

Examples
To make a registration:

LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.

Once UNLISTEN has been executed, further NOTIFY commands will be ignored:

UNLISTEN virtual;
NOTIFY virtual;
-- no NOTIFY event is received
Compatibility

There is no `UNLISTEN` command in the SQL standard.

See Also

`LISTEN`, `NOTIFY`
**UPDATE**

**Name**

UPDATE — update rows of a table

**Synopsis**

```
UPDATE [ ONLY ] table [ * ] [ [ AS ] alias ]
  SET { column = { expression | DEFAULT } |
    ( column [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
  [ FROM fromlist ]
  [ WHERE condition | WHERE CURRENT OF cursor_name ]
  [ RETURNING * | output_expression [ AS output_name ] [, ...] ]
```

**Description**

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

There are two ways to modify a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the FROM clause. Which technique is more appropriate depends on the specific circumstances.

The optional RETURNING clause causes UPDATE to compute and return value(s) based on each row actually updated. Any expression using the table’s columns, and/or columns of other tables mentioned in FROM, can be computed. The new (post-update) values of the table’s columns are used. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have the UPDATE privilege on the table to update it, as well as the SELECT privilege to any table whose values are read in the expressions or condition.

**Parameters**

- **table**
  
The name (optionally schema-qualified) of the table to update. If ONLY is specified before the table name, matching rows are updated in the named table only. If ONLY is not specified, matching rows are also updated in any tables inheriting from the named table. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

- **alias**
  
A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given UPDATE foo AS f, the remainder of the UPDATE statement must refer to this table as f not foo.

- **column**
  
The name of a column in table. The column name can be qualified with a subfield name or array subscript, if needed. Do not include the table’s name in the specification of a target column — for example, UPDATE tab SET tab.col = 1 is invalid.
**UPDATE**

*expression*

An expression to assign to the column. The expression can use the old values of this and other columns in the table.

**DEFAULT**

Set the column to its default value (which will be NULL if no specific default expression has been assigned to it).

*fromlist*

A list of table expressions, allowing columns from other tables to appear in the **WHERE** condition and the update expressions. This is similar to the list of tables that can be specified in the **FROM Clause** of a **SELECT** statement. Note that the target table must not appear in the **fromlist**, unless you intend a self-join (in which case it must appear with an alias in the **fromlist**).

*condition*

An expression that returns a value of type **boolean**. Only rows for which this expression returns **true** will be updated.

*cursor_name*

The name of the cursor to use in a **WHERE CURRENT OF** condition. The row to be updated is the one most recently fetched from this cursor. The cursor must be a simple (non-join, non-aggregate) query on the **UPDATE**'s target table. Note that **WHERE CURRENT OF** cannot be specified together with a Boolean condition.

*output_expression*

An expression to be computed and returned by the **UPDATE** command after each row is updated. The expression can use any column names of the **table** or **table(s)** listed in **FROM**. Write * to return all columns.

*output_name*

A name to use for a returned column.

**Outputs**

On successful completion, an **UPDATE** command returns a command tag of the form

**UPDATE count**

The **count** is the number of rows updated. If **count** is 0, no rows matched the **condition** (this is not considered an error).

If the **UPDATE** command contains a **RETURNING** clause, the result will be similar to that of a **SELECT** statement containing the columns and values defined in the **RETURNING** list, computed over the row(s) updated by the command.

**Notes**

When a **FROM** clause is present, what essentially happens is that the target table is joined to the tables mentioned in the **fromlist**, and each output row of the join represents an update operation for the target table. When using **FROM** you should ensure that the join produces at most one output row for each row to be modified. In other words, a target row shouldn’t join to more than one row from the
other table(s). If it does, then only one of the join rows will be used to update the target row, but which one will be used is not readily predictable. Because of this indeterminacy, referencing other tables only within sub-selects is safer, though often harder to read and slower than using a join.

**Examples**

Change the word Drama to Dramatic in the column kind of the table films:

```sql
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Adjust temperature entries and reset precipitation to its default value in one row of the table weather:

```sql
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
    WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Perform the same operation and return the updated entries:

```sql
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
    WHERE city = 'San Francisco' AND date = '2003-07-03'
    RETURNING temp_lo, temp_hi, prcp;
```

Use the alternative column-list syntax to do the same update:

```sql
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
    WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, using the FROM clause syntax:

```sql
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
    WHERE accounts.name = 'Acme Corporation'
    AND employees.id = accounts.sales_person;
```

Perform the same operation, using a sub-select in the WHERE clause:

```sql
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

Attempt to insert a new stock item along with the quantity of stock. If the item already exists, instead update the stock count of the existing item. To do this without failing the entire transaction, use savepoints:

```sql
BEGIN;
    -- other operations
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
```
UPDATE

-- Assume the above fails because of a unique key violation,
-- so now we issue these commands:
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- continue with other operations, and eventually
COMMIT;

Change the kind column of the table films in the row on which the cursor c_films is currently positioned:

UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;

Compatibility

This command conforms to the SQL standard, except that the FROM and RETURNING clauses are PostgreSQL extensions.

According to the standard, the column-list syntax should allow a list of columns to be assigned from a single row-valued expression, such as a sub-select:

UPDATE accounts SET (contact_last_name, contact_first_name) =
(SELECT last_name, first_name FROM salesmen
WHERE salesmen.id = accounts.sales_id);

This is not currently implemented — the source must be a list of independent expressions.

Some other database systems offer a FROM option in which the target table is supposed to be listed again within FROM. That is not how PostgreSQL interprets FROM. Be careful when porting applications that use this extension.
**VACUUM**

**Name**

VACUUM — garbage-collect and optionally analyze a database

**Synopsis**

VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]

**Description**

VACUUM reclaims storage occupied by dead tuples. In normal PostgreSQL operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is done. Therefore it’s necessary to do VACUUM periodically, especially on frequently-updated tables.

With no parameter, VACUUM processes every table in the current database. With a parameter, VACUUM processes only that table.

VACUUM ANALYZE performs a VACUUM and then an ANALYZE for each selected table. This is a handy combination form for routine maintenance scripts. See ANALYZE for more details about its processing.

Plain VACUUM (without FULL) simply reclaims space and makes it available for re-use. This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. VACUUM FULL does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

**Parameters**

**FULL**

Selects “full” vacuum, which can reclaim more space, but takes much longer and exclusively locks the table.

**FREEZE**

Selects aggressive “freezing” of tuples. Specifying FREEZE is equivalent to performing VACUUM with the vacuum_freeze_min_age parameter set to zero. The FREEZE option is deprecated and will be removed in a future release; set the parameter instead.

**VERBOSE**

Prints a detailed vacuum activity report for each table. Can be used to help determine appropriate settings for max_fsm_pages, max_fsm_relations, and default_statistics_target.

**ANALYZE**

Updates statistics used by the planner to determine the most efficient way to execute a query.
The name (optionally schema-qualified) of a specific table to vacuum. Defaults to all tables in the current database.

The name of a specific column to analyze. Defaults to all columns.

When `VERBOSE` is specified, `VACUUM` emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

`VACUUM` cannot be executed inside a transaction block.

We recommend that active production databases be vacuumed frequently (at least nightly), in order to remove dead rows. After adding or deleting a large number of rows, it might be a good idea to issue a `VACUUM ANALYZE` command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the PostgreSQL query planner to make better choices in planning queries.

The `FULL` option is not recommended for routine use, but might be useful in special cases. An example is when you have deleted or updated most of the rows in a table and would like the table to physically shrink to occupy less disk space and allow faster table scans. `VACUUM FULL` will usually shrink the table more than a plain `VACUUM` would. The `FULL` option does not shrink indexes; a periodic `REINDEX` is still recommended. In fact, it is often faster to drop all indexes, `VACUUM FULL`, and recreate the indexes.

`VACUUM` causes a substantial increase in I/O traffic, which might cause poor performance for other active sessions. Therefore, it is sometimes advisable to use the cost-based vacuum delay feature. See Section 18.4.4 for details.

PostgreSQL includes an “autovacuum” facility which can automate routine vacuum maintenance. For more information about automatic and manual vacuuming, see Section 23.1.

The following is an example from running `VACUUM` on a table in the regression database:

```
regression=# VACUUM VERBOSE ANALYZE onek;
INFO: vacuuming "public.onek"
INFO: index "onek_unique1" now contains 1000 tuples in 14 pages
DETAIL: 3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.18 sec.
INFO: index "onek_unique2" now contains 1000 tuples in 16 pages
DETAIL: 3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.07u sec elapsed 0.23 sec.
INFO: index "onek_hundred" now contains 1000 tuples in 13 pages
DETAIL: 3000 index tuples were removed.
```
VACUUM

0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.17 sec.
INFO: index "onek_stringu1" now contains 1000 tuples in 48 pages
DETAIL: 3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.09u sec elapsed 0.59 sec.
INFO: "onek": removed 3000 tuples in 108 pages
DETAIL: CPU 0.01s/0.06u sec elapsed 0.07 sec.
INFO: "onek": found 3000 removable, 1000 nonremovable tuples in 143 pages
DETAIL: 0 dead tuples cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.07s/0.39u sec elapsed 1.56 sec.
INFO: analyzing "public.onek"
INFO: "onek": 36 pages, 1000 rows sampled, 1000 estimated total rows
VACUUM

Compatibility

There is no VACUUM statement in the SQL standard.

See Also

vacuumdb, Cost-Based Vacuum Delay, The Auto-Vacuum Daemon
VALUES

Name

VALUES — compute a set of rows

Synopsis

VALUES ( expression [, ...] ) [, ...]
[ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start ]

Description

VALUES computes a row value or set of row values specified by value expressions. It is most commonly used to generate a “constant table” within a larger command, but it can be used on its own.

When more than one row is specified, all the rows must have the same number of elements. The data types of the resulting table’s columns are determined by combining the explicit or inferred types of the expressions appearing in that column, using the same rules as for \texttt{UNION} (see Section 10.5).

Within larger commands, VALUES is syntactically allowed anywhere that SELECT is. Because it is treated like a SELECT by the grammar, it is possible to use the ORDER BY, LIMIT, and OFFSET clauses with a VALUES command.

Parameters

description

expression

A constant or expression to compute and insert at the indicated place in the resulting table (set of rows). In a VALUES list appearing at the top level of an INSERT, an expression can be replaced by DEFAULT to indicate that the destination column’s default value should be inserted. DEFAULT cannot be used when VALUES appears in other contexts.

sort_expression

An expression or integer constant indicating how to sort the result rows. This expression can refer to the columns of the VALUES result as column1, column2, etc. For more details see \texttt{ORDER BY Clause}.

operator

A sorting operator. For details see \texttt{ORDER BY Clause}.

count

The maximum number of rows to return. For details see \texttt{LIMIT Clause}.

start

The number of rows to skip before starting to return rows. For details see \texttt{LIMIT Clause}.
Notes
VALUES lists with very large numbers of rows should be avoided, as you might encounter out-of-memory failures or poor performance. VALUES appearing within INSERT is a special case (because the desired column types are known from the INSERT’s target table, and need not be inferred by scanning the VALUES list), so it can handle larger lists than are practical in other contexts.

Examples
A bare VALUES command:
VALUES (1, 'one'), (2, 'two'), (3, 'three');
This will return a table of two columns and three rows. It’s effectively equivalent to:
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';

More usually, VALUES is used within a larger SQL command. The most common use is in INSERT:
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');

In the context of INSERT, entries of a VALUES list can be DEFAULT to indicate that the column default should be used here instead of specifying a value:
INSERT INTO films VALUES
('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);

VALUES can also be used where a sub-SELECT might be written, for example in a FROM clause:
SELECT f.*
FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
WHERE employees.depno = v.depno AND employees.sales >= v.target;

Note that an AS clause is required when VALUES is used in a FROM clause, just as is true for SELECT. It is not required that the AS clause specify names for all the columns, but it’s good practice to do so. (The default column names for VALUES are column1, column2, etc in PostgreSQL, but these names might be different in other database systems.)

When VALUES is used in INSERT, the values are all automatically coerced to the data type of the corresponding destination column. When it’s used in other contexts, it might be necessary to specify the correct data type. If the entries are all quoted literal constants, coercing the first is sufficient to determine the assumed type for all:
VALUES

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

Tip: For simple IN tests, it's better to rely on the list-of-scalars form of IN than to write a VALUES query as shown above. The list of scalars method requires less writing and is often more efficient.

Compatibility

VALUES conforms to the SQL standard, except that LIMIT and OFFSET are PostgreSQL extensions.

See Also

INSERT, SELECT
II. PostgreSQL Client Applications

This part contains reference information for PostgreSQL client applications and utilities. Not all of these commands are of general utility, some might require special privileges. The common feature of these applications is that they can be run on any host, independent of where the database server resides.
**clusterdb**

**Name**

`clusterdb` — cluster a PostgreSQL database

**Synopsis**

```
clusterdb [connection-option...] [--table | -t table] [dbname]
clusterdb [connection-option...] [--all | -a]
```

**Description**

`clusterdb` is a utility for reclustering tables in a PostgreSQL database. It finds tables that have previously been clustered, and clusters them again on the same index that was last used. Tables that have never been clustered are not affected.

`clusterdb` is a wrapper around the SQL command `CLUSTER`. There is no effective difference between clustering databases via this utility and via other methods for accessing the server.

**Options**

`clusterdb` accepts the following command-line arguments:

- `-a`
  --all
  Cluster all databases.

- `[-d] dbname`

  Specifies the name of the database to be clustered. If this is not specified and `-a` (or `--all`) is not used, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used.

- `-e`
  --echo
  Echo the commands that `clusterdb` generates and sends to the server.

- `-q`
  --quiet
  Do not display progress messages.

- `[-t] table`
  --table table
  Cluster `table` only.
clusterdb also accepts the following command-line arguments for connection parameters:

- **-h** host
- **--host** host

  Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- **-p** port
- **--port** port

  Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- **-U** username
- **--username** username

  User name to connect as.

- **-W**
- **--password**

  Force clusterdb to prompt for a password before connecting to a database.

  This option is never essential, since clusterdb will automatically prompt for a password if the server demands password authentication. However, clusterdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

### Environment

- PGDATABASE
- PGHOST
- PGPORT
- PGUSER

  Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

### Diagnostics

In case of difficulty, see `CLUSTER` and `psql` for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

### Examples

To cluster the database `test`:

```
$ clusterdb test
```
To cluster a single table `foo` in a database named `xyzzy`:

```
$ clusterdb --table foo xyzzy
```

**See Also**

`CLUSTER`
createdb

**Name**

createdb — create a new PostgreSQL database

**Synopsis**

createdb [option...] [dbname] [description]

**Description**

createdb creates a new PostgreSQL database.

Normally, the database user who executes this command becomes the owner of the new database. However a different owner can be specified via the `-O` option, if the executing user has appropriate privileges.

createdb is a wrapper around the SQL command `CREATE DATABASE`. There is no effective difference between creating databases via this utility and via other methods for accessing the server.

**Options**

createdb accepts the following command-line arguments:

- **dbname**
  
  Specifies the name of the database to be created. The name must be unique among all PostgreSQL databases in this cluster. The default is to create a database with the same name as the current system user.

- **description**
  
  Specifies a comment to be associated with the newly created database.

- **-D tablespace**
  
  `-tablespace tablespace`

  Specifies the default tablespace for the database.

- **-e**
  
  `--echo`

  Echo the commands that createdb generates and sends to the server.

- **-E encoding**
  
  `--encoding encoding`

  Specifies the character encoding scheme to be used in this database. The character sets supported by the PostgreSQL server are described in Section 22.2.1.

- **-O owner**
  
  `--owner owner`

  Specifies the database user who will own the new database.
-T template
--template template

Specifies the template database from which to build this database.

The options -D, -E, -O, and -T correspond to options of the underlying SQL command CREATE DATABASE; see there for more information about them.

createdb also accepts the following command-line arguments for connection parameters:

-h host
--host host

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

-p port
--port port

Specifies the TCP port or the local Unix domain socket file extension on which the server is listening for connections.

-U username
--username username

User name to connect as.

-W
--password

Force createdb to prompt for a password before connecting to a database.

This option is never essential, since createdb will automatically prompt for a password if the server demands password authentication. However, createdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing -W to avoid the extra connection attempt.

Environment

PGDATABASE

If set, the name of the database to create, unless overridden on the command line.

PGHOST
PGPORT
PGUSER

Default connection parameters. PGUSER also determines the name of the database to create, if it is not specified on the command line or by PGDATABASE.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).
Diagnostics

In case of difficulty, see CREATE DATABASE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To create the database demo using the default database server:

$ createdb demo

To create the database demo using the server on host eden, port 5000, using the LATIN1 encoding scheme with a look at the underlying command:

$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE demo ENCODING 'LATIN1';

See Also

dropdb, CREATE DATABASE
createlang

Name
createlang — define a new PostgreSQL procedural language

Synopsis
createlang[connection-option...] langname [dbname]
createlang[connection-option...] --list [-l] dbname

Description
createlang is a utility for adding a new programming language to a PostgreSQL database. createlang is just a wrapper around the CREATE LANGUAGE command.

Options
createlang accepts the following command-line arguments:

langname
   Specifies the name of the procedural programming language to be defined.
[-d] dbname
[--dbname] dbname
   Specifies to which database the language should be added. The default is to use the database with the same name as the current system user.
-e
   --echo
   Display SQL commands as they are executed.
-l
   --list
   Show a list of already installed languages in the target database.

createlang also accepts the following command-line arguments for connection parameters:

-h host
   --host host
   Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.
-p port
   --port port
   Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.
-U username
--username username

User name to connect as.

-W
--password

Force createlang to prompt for a password before connecting to a database.

This option is never essential, since createlang will automatically prompt for a password if the server demands password authentication. However, createlang will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing -W to avoid the extra connection attempt.

Environment

PGDATABASE
PGHOST
PGPORT
PGUSER

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

Diagnostics

Most error messages are self-explanatory. If not, run createlang with the --echo option and see under the respective SQL command for details. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

Use droplang to remove a language.

Examples

To install the language pltcl into the database template1:

$ createlang pltcl template1

Note that installing the language into template1 will cause it to be automatically installed into subsequently-created databases as well.
See Also

droplang, CREATE LANGUAGE
createuser

Name
createuser — define a new PostgreSQL user account

Synopsis
createuser [option...] [username]

Description
createuser creates a new PostgreSQL user (or more precisely, a role). Only superusers and users with CREATEROLE privilege can create new users, so createuser must be invoked by someone who can connect as a superuser or a user with CREATEROLE privilege.

If you wish to create a new superuser, you must connect as a superuser, not merely with CREATEROLE privilege. Being a superuser implies the ability to bypass all access permission checks within the database, so superuserdom should not be granted lightly.

createuser is a wrapper around the SQL command CREATE ROLE. There is no effective difference between creating users via this utility and via other methods for accessing the server.

Options
createuser accepts the following command-line arguments:

username
Specifies the name of the PostgreSQL user to be created. This name must be different from all existing roles in this PostgreSQL installation.

-s
--superuser
The new user will be a superuser.

-S
--no-superuser
The new user will not be a superuser. This is the default.

-d
--createdb
The new user will be allowed to create databases.

-D
--no-createdb
The new user will not be allowed to create databases. This is the default.
createuser

-\texttt{r}
--\texttt{createrole}

The new user will be allowed to create new roles (that is, this user will have \texttt{CREATEROLE}
privilege).

-\texttt{R}
--\texttt{no-createrole}

The new user will not be allowed to create new roles. This is the default.

-\texttt{l}
--\texttt{login}

The new user will be allowed to log in (that is, the user name can be used as the initial session
user identifier). This is the default.

-\texttt{L}
--\texttt{no-login}

The new user will not be allowed to log in. (A role without login privilege is still useful as a
means of managing database permissions.)

-\texttt{i}
--\texttt{inherit}

The new role will automatically inherit privileges of roles it is a member of. This is the default.

-\texttt{I}
--\texttt{no-inherit}

The new role will not automatically inherit privileges of roles it is a member of.

-\texttt{c \hspace{3pt} \texttt{number}}
--\texttt{connection-limit \hspace{3pt} \texttt{number}}

Set a maximum number of connections for the new user. The default is to set no limit.

-\texttt{P}
--\texttt{pwprompt}

If given, createuser will issue a prompt for the password of the new user. This is not necessary if
you do not plan on using password authentication.

-\texttt{E}
--\texttt{encrypted}

Encrypts the user’s password stored in the database. If not specified, the default password behav-
ior is used.

-\texttt{N}
--\texttt{unencrypted}

Does not encrypt the user’s password stored in the database. If not specified, the default password
behavior is used.

-\texttt{e}
--\texttt{echo}

Echo the commands that createuser generates and sends to the server.

You will be prompted for a name and other missing information if it is not specified on the command
line.
createuser also accepts the following command-line arguments for connection parameters:

- `-h host`
  `-host host`
  Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- `-p port`
  `-port port`
  Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- `-U username`
  `-username username`
  User name to connect as (not the user name to create).

- `-W`
  `---password`
  Force createuser to prompt for a password (for connecting to the server, not for the password of the new user).
  This option is never essential, since createuser will automatically prompt for a password if the server demands password authentication. However, createuser will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

### Environment

- `PGHOST`
- `PGPORT`
- `PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

### Diagnostics

In case of difficulty, see `CREATE ROLE` and `psql` for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

### Examples

To create a user `joe` on the default database server:

```
$ createuser joe
Shall the new role be a superuser? [y/n] n
Shall the new role be allowed to create databases? [y/n] n
```
Shall the new role be allowed to create more new roles? (y/n) n

To create the same user joe using the server on host eden, port 5000, avoiding the prompts and taking a look at the underlying command:

```bash
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

To create the user joe as a superuser, and assign a password immediately:

```bash
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;
```

In the above example, the new password isn’t actually echoed when typed, but we show what was typed for clarity. As you see, the password is encrypted before it is sent to the client. If the option --unencrypted is used, the password will appear in the echoed command (and possibly also in the server log and elsewhere), so you don’t want to use -e in that case, if anyone else can see your screen.

**See Also**

dropuser, CREATE ROLE
dropdb

Name

dropdb — remove a PostgreSQL database

Synopsis

dropdb [option...] dbname

Description

dropdb destroys an existing PostgreSQL database. The user who executes this command must be a
database superuser or the owner of the database.
dropdb is a wrapper around the SQL command DROP DATABASE. There is no effective difference
between dropping databases via this utility and via other methods for accessing the server.

Options

dropdb accepts the following command-line arguments:

dbname
    Specifies the name of the database to be removed.
-e
--echo
    Echo the commands that dropdb generates and sends to the server.
-i
--interactive
    Issues a verification prompt before doing anything destructive.

dropdb also accepts the following command-line arguments for connection parameters:

-h host
--host host
    Specifies the host name of the machine on which the server is running. If the value begins with a
slash, it is used as the directory for the Unix domain socket.
-p port
--port port
    Specifies the TCP port or local Unix domain socket file extension on which the server is listening
for connections.
dropdb

-\texttt{U \ username}
-\texttt{--username \ username}

User name to connect as.

-\texttt{W}
-\texttt{--password}

Force dropdb to prompt for a password before connecting to a database.

This option is never essential, since dropdb will automatically prompt for a password if the server demands password authentication. However, dropdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing \texttt{-W} to avoid the extra connection attempt.

\textbf{Environment}

\texttt{PGHOST}
\texttt{PGPORT}
\texttt{PGUSER}

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

\textbf{Diagnostics}

In case of difficulty, see \texttt{DROP DATABASE} and \texttt{psql} for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

\textbf{Examples}

To destroy the database \texttt{demo} on the default database server:

\$ \texttt{dropdb demo}

To destroy the database \texttt{demo} using the server on host \texttt{eden}, port 5000, with verification and a peek at the underlying command:

\$ \texttt{dropdb \ -p \ 5000 \ -h \ eden \ -i \ -e \ demo}
Database "demo" will be permanently deleted.
Are you sure? (y/n) \texttt{y}
DROP DATABASE demo;
See Also

createdb, DROP DATABASE
droplang

Name
droplang — remove a PostgreSQL procedural language

Synopsis
droplang [connection-option... langname [dbname]]
droplang [connection-option... --list -l dbname]

Description
droplang is a utility for removing an existing programming language from a PostgreSQL database.
droplang can drop any procedural language, even those not supplied by the PostgreSQL distribution.
Although backend programming languages can be removed directly using several SQL commands, it
is recommended to use droplang because it performs a number of checks and is much easier to use.
See DROP LANGUAGE for more.

Options
droplang accepts the following command line arguments:

langname
   Specifies the name of the backend programming language to be removed.

[-d] dbname
[--dbname] dbname
   Specifies from which database the language should be removed. The default is to use the database
   with the same name as the current system user.

-e
--echo
   Display SQL commands as they are executed.

-l
--list
   Show a list of already installed languages in the target database.

droplang also accepts the following command line arguments for connection parameters:

-h host
--host host
   Specifies the host name of the machine on which the server is running. If host begins with a
   slash, it is used as the directory for the Unix domain socket.
-p port
--port port
    Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server
    is listening for connections.
-U username
--username username
    User name to connect as.
-W
--password
    Force droplang to prompt for a password before connecting to a database.
    This option is never essential, since droplang will automatically prompt for a password if the
    server demands password authentication. However, droplang will waste a connection attempt
    finding out that the server wants a password. In some cases it is worth typing -W to avoid the
    extra connection attempt.

Environment

PGDATABASE
PGHOST
PGPORT
PGUSER
    Default connection parameters
This utility, like most other PostgreSQL utilities, also uses the environment variables supported by
libpq (see Section 30.12).

Diagnostics

Most error messages are self-explanatory. If not, run droplang with the --echo option and see un-
der the respective SQL command for details. Also, any default connection settings and environment
variables used by the libpq front-end library will apply.

Notes

Use createlang to add a language.

Examples

To remove the language pltcl:

$ droplang pltcl dbname
See Also

createlang, DROP LANGUAGE
dropuser

Name

dropuser — remove a PostgreSQL user account

Synopsis

dropuser [option...] [username]

Description

dropuser removes an existing PostgreSQL user. Only superusers and users with the CREATEROLE privilege can remove PostgreSQL users. (To remove a superuser, you must yourself be a superuser.) dropuser is a wrapper around the SQL command DROP ROLE. There is no effective difference between dropping users via this utility and via other methods for accessing the server.

Options

dropuser accepts the following command-line arguments:

username

  Specifies the name of the PostgreSQL user to be removed. You will be prompted for a name if none is specified on the command line.

-e
  --echo

  Echo the commands that dropuser generates and sends to the server.

-i
  --interactive

  Prompt for confirmation before actually removing the user.

dropuser also accepts the following command-line arguments for connection parameters:

-h host
  --host host

  Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

-p port
  --port port

  Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.
dropuser

- U  username
  --username  username

    User name to connect as (not the user name to drop).

- W
  --password

    Force dropuser to prompt for a password before connecting to a database.

    This option is never essential, since dropuser will automatically prompt for a password if the
    server demands password authentication. However, dropuser will waste a connection attempt
    finding out that the server wants a password. In some cases it is worth typing -W to avoid the
    extra connection attempt.

Environment

PGHOST
PGPORT
PGUSER

    Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by
libpq (see Section 30.12).

Diagnostics

In case of difficulty, see DROP ROLE and psql for discussions of potential problems and error mes-
sages. The database server must be running at the targeted host. Also, any default connection settings
and environment variables used by the libpq front-end library will apply.

Examples

To remove user joe from the default database server:

 $ dropuser joe

To remove user joe using the server on host eden, port 5000, with verification and a peek at the
underlying command:

 $ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
See Also

createuser, DROP ROLE
ecpg

Name

ecpg — embedded SQL C preprocessor

Synopsis

ecpg [option...] file...

Description

ecpg is the embedded SQL preprocessor for C programs. It converts C programs with embedded SQL statements to normal C code by replacing the SQL invocations with special function calls. The output files can then be processed with any C compiler tool chain.

ecpg will convert each input file given on the command line to the corresponding C output file. Input files preferably have the extension .pgc, in which case the extension will be replaced by .c to determine the output file name. If the extension of the input file is not .pgc, then the output file name is computed by appending .c to the full file name. The output file name can also be overridden using the -o option.

This reference page does not describe the embedded SQL language. See Chapter 32 for more information on that topic.

Options

ecpg accepts the following command-line arguments:

- `c`

  Automatically generate certain C code from SQL code. Currently, this works for EXEC SQL TYPE.

- `C mode`

  Set a compatibility mode. mode can be INFORMIX or INFORMIX_SE.

- `D symbol`

  Define a C preprocessor symbol.

- `i`

  Parse system include files as well.

- `I directory`

  Specify an additional include path, used to find files included via EXEC SQL INCLUDE. Defaults are . (current directory), /usr/local/include, the PostgreSQL include directory which is defined at compile time (default: /usr/local/pgsql/include), and /usr/include, in that order.
ecpg

-o filename
   Specifies that ecpg should write all its output to the given filename.

-r option
   Selects a run-time behavior. Currently, option can only be no_indicator.

-t
   Turn on autocommit of transactions. In this mode, each SQL command is automatically committed unless it is inside an explicit transaction block. In the default mode, commands are committed only when EXEC SQL COMMIT is issued.

-v
   Print additional information including the version and the include path.

--help
   Show a brief summary of the command usage, then exit.

--version
   Output version information, then exit.

Notes

When compiling the preprocessed C code files, the compiler needs to be able to find the ECPG header files in the PostgreSQL include directory. Therefore, one might have to use the -I option when invoking the compiler (e.g., -I/usr/local/pgsql/include).

Programs using C code with embedded SQL have to be linked against the libecpg library, for example using the linker options -L/usr/local/pgsql/lib -lecpg.

The value of either of these directories that is appropriate for the installation can be found out using pg_config.

Examples

If you have an embedded SQL C source file named progl.pgc, you can create an executable program using the following sequence of commands:

ecpg progl.pgc
cc -I/usr/local/pgsql/include -c progl.c
cc -o progl progl.o -L/usr/local/pgsql/lib -lecpg
**Name**

*pg_config* — retrieve information about the installed version of PostgreSQL

**Synopsis**

`pg_config [option...]`

**Description**

The *pg_config* utility prints configuration parameters of the currently installed version of PostgreSQL. It is intended, for example, to be used by software packages that want to interface to PostgreSQL to facilitate finding the required header files and libraries.

**Options**

To use *pg_config*, supply one or more of the following options:

```bash
--bindir
  Print the location of user executables. Use this, for example, to find the *psql* program. This is normally also the location where the *pg_config* program resides.

--docdir
  Print the location of documentation files. (This will be an empty string if *--without-docdir* was specified when PostgreSQL was built.)

--includedir
  Print the location of C header files of the client interfaces.

--pkgincludedir
  Print the location of other C header files.

--includedir-server
  Print the location of C header files for server programming.

--libdir
  Print the location of object code libraries.

--pkglibdir
  Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files might also be installed in this directory.)

--localedir
  Print the location of locale support files. (This will be an empty string if locale support was not configured when PostgreSQL was built.)
```
```
--mandir
    Print the location of manual pages.
--sharedir
    Print the location of architecture-independent support files.
--sysconfdir
    Print the location of system-wide configuration files.
--pgxs
    Print the location of extension makefiles.
--configure
    Print the options that were given to the configure script when PostgreSQL was configured for building. This can be used to reproduce the identical configuration, or to find out with what options a binary package was built. (Note however that binary packages often contain vendor-specific custom patches.) See also the examples below.
--cc
    Print the value of the CC variable that was used for building PostgreSQL. This shows the C compiler used.
--cppflags
    Print the value of the CPPFLAGS variable that was used for building PostgreSQL. This shows C compiler switches needed at preprocessing time (typically, -I switches).
--cflags
    Print the value of the CFLAGS variable that was used for building PostgreSQL. This shows C compiler switches.
--cflags_sl
    Print the value of the CFLAGS_SL variable that was used for building PostgreSQL. This shows extra C compiler switches used for building shared libraries.
--ldflags
    Print the value of the LDFLAGS variable that was used for building PostgreSQL. This shows linker switches.
--ldflags_sl
    Print the value of the LDFLAGS_SL variable that was used for building PostgreSQL. This shows linker switches used for building shared libraries.
--libs
    Print the value of the LIBS variable that was used for building PostgreSQL. This normally contains -l switches for external libraries linked into PostgreSQL.
--version
    Print the version of PostgreSQL.
```

If more than one option is given, the information is printed in that order, one item per line. If no options are given, all available information is printed, with labels.
Notes

The option --includedir-server was new in PostgreSQL 7.2. In prior releases, the server include files were installed in the same location as the client headers, which could be queried with the option --includedir. To make your package handle both cases, try the newer option first and test the exit status to see whether it succeeded.

The options --docdir, --pkgincludedir, --localedir, --mandir, --sharedir, --sysconfdir, --cc, --cppflags, --cflags, --cflags_sl, --ldflags, --ldflags_sl, and --libs are new in PostgreSQL 8.1.

In releases prior to PostgreSQL 7.1, before pg_config came to be, a method for finding the equivalent configuration information did not exist.

Example

To reproduce the build configuration of the current PostgreSQL installation, run the following command:

eval ./configure 'pg_config --configure'

The output of pg_config --configure contains shell quotation marks so arguments with spaces are represented correctly. Therefore, using eval is required for proper results.


**pg_dump**

**Name**

pg_dump — extract a PostgreSQL database into a script file or other archive file

**Synopsis**

pg_dump [option...] [dbname]

**Description**

pg_dump is a utility for backing up a PostgreSQL database. It makes consistent backups even if the database is being used concurrently. pg_dump does not block other users accessing the database (readers or writers).

Dumps can be output in script or archive file formats. Script dumps are plain-text files containing the SQL commands required to reconstruct the database to the state it was in at the time it was saved. To restore from such a script, feed it to psql. Script files can be used to reconstruct the database even on other machines and other architectures; with some modifications even on other SQL database products.

The alternative archive file formats must be used with pg_restore to rebuild the database. They allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive file formats are designed to be portable across architectures.

When used with one of the archive file formats and combined with pg_restore, pg_dump provides a flexible archival and transfer mechanism. pg_dump can be used to backup an entire database, then pg_restore can be used to examine the archive and/or select which parts of the database are to be restored. The most flexible output file format is the “custom” format (-Fc). It allows for selection and reordering of all archived items, and is compressed by default. The tar format (-Ft) is not compressed and it is not possible to reorder data when loading, but it is otherwise quite flexible; moreover, it can be manipulated with standard Unix tools such as tar.

While running pg_dump, one should examine the output for any warnings (printed on standard error), especially in light of the limitations listed below.

**Options**

The following command-line options control the content and format of the output.

- **dbname**
  - Specifies the name of the database to be dumped. If this is not specified, the environment variable PGDATABASE is used. If that is not set, the user name specified for the connection is used.
  - **-a**
  - **--data-only**
    - Dump only the data, not the schema (data definitions).
This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

- **-b**
  --blobs
  Include large objects in the dump. This is the default behavior except when --schema, --table, or --schema-only is specified, so the -b switch is only useful to add large objects to selective dumps.

- **-c**
  --clean
  Output commands to clean (drop) database objects prior to (the commands for) creating them.
  This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

- **-C**
  --create
  Begin the output with a command to create the database itself and reconnect to the created database. (With a script of this form, it doesn’t matter which database you connect to before running the script.)
  This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

- **-d**
  --inserts
  Dump data as INSERT commands (rather than COPY). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents. Note that the restore might fail altogether if you have rearranged column order. The -D option is safe against column order changes, though even slower.

- **-D**
  --column-inserts
  --attribute-inserts
  Dump data as INSERT commands with explicit column names (INSERT INTO table (column, ...) VALUES ...). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. Also, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents.

- **-E**
  encoding
  --encoding=encoding
  Create the dump in the specified character set encoding. By default, the dump is created in the database encoding. (Another way to get the same result is to set the PGCLIENTENCODING environment variable to the desired dump encoding.)

- **-f**
  file
  --file=file
  Send output to the specified file. If this is omitted, the standard output is used.
pg_dump

-F format

--format=format

Selects the format of the output. format can be one of the following:

p
plain

Output a plain-text SQL script file (the default).

c
custom

Output a custom archive suitable for input into pg_restore. This is the most flexible format in that it allows reordering of loading data as well as object definitions. This format is also compressed by default.

t
tar

Output a tar archive suitable for input into pg_restore. Using this archive format allows reordering and/or exclusion of database objects at the time the database is restored. It is also possible to limit which data is reloaded at restore time.

-i

--ignore-version

Ignore version mismatch between pg_dump and the database server.

pg_dump can dump from servers running previous releases of PostgreSQL, but very old versions are not supported anymore (currently, those prior to 7.0). Dumping from a server newer than pg_dump is likely not to work at all. Use this option if you need to override the version check (and if pg_dump then fails, don’t say you weren’t warned).

-n schema

--schema=schema

Dump only schemas matching schema; this selects both the schema itself, and all its contained objects. When this option is not specified, all non-system schemas in the target database will be dumped. Multiple schemas can be selected by writing multiple -n switches. Also, the schema parameter is interpreted as a pattern according to the same rules used by psql’s \d commands (see Patterns), so multiple schemas can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards.

Note: When -n is specified, pg_dump makes no attempt to dump any other database objects that the selected schema(s) might depend upon. Therefore, there is no guarantee that the results of a specific-schema dump can be successfully restored by themselves into a clean database.

Note: Non-schema objects such as blobs are not dumped when -n is specified. You can add blobs back to the dump with the --blobs switch.
pg_dump

-N schema
--exclude-schema=schema

Do not dump any schemas matching the schema pattern. The pattern is interpreted according to
the same rules as for -n. -N can be given more than once to exclude schemas matching any of
several patterns.

When both -n and -N are given, the behavior is to dump just the schemas that match at least one
-n switch but no -N switches. If -N appears without -n, then schemas matching -N are excluded
from what is otherwise a normal dump.

-o
--oids

Dump object identifiers (OIDs) as part of the data for every table. Use this option if your appli-
cation references the OID columns in some way (e.g., in a foreign key constraint). Otherwise,
this option should not be used.

-O
--no-owner

Do not output commands to set ownership of objects to match the original database. By default,
pg_dump issues ALTER OWNER or SET SESSION AUTHORIZATION statements to set ownership
of created database objects. These statements will fail when the script is run unless it is started
by a superuser (or the same user that owns all of the objects in the script). To make a script that
can be restored by any user, but will give that user ownership of all the objects, specify -O.

This option is only meaningful for the plain-text format. For the archive formats, you can specify
the option when you call pg_restore.

-R
--no-reconnect

This option is obsolete but still accepted for backwards compatibility.

-s
--schema-only

Dump only the object definitions (schema), not data.

-S username
--superuser=username

Specify the superuser user name to use when disabling triggers. This is only relevant if
--disable-triggers is used. (Usually, it’s better to leave this out, and instead start the
resulting script as superuser.)

-t table
--table=table

Dump only tables (or views or sequences) matching table. Multiple tables can be selected by
writing multiple -t switches. Also, the table parameter is interpreted as a pattern according
to the same rules used by psql’s \d commands (see Patterns), so multiple tables can also be
selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote
the pattern if needed to prevent the shell from expanding the wildcards.

The -n and -N switches have no effect when -t is used, because tables selected by -t will be
dumped regardless of those switches, and non-table objects will not be dumped.

Note: When -t is specified, pg_dump makes no attempt to dump any other database objects
that the selected table(s) might depend upon. Therefore, there is no guarantee that the
results of a specific-table dump can be successfully restored by themselves into a clean database.

**Note:** The behavior of the `-t` switch is not entirely upward compatible with pre-8.2 PostgreSQL versions. Formerly, writing `-t tab` would dump all tables named `tab`, but now it just dumps whichever one is visible in your default search path. To get the old behavior you can write `-t '.tab'`. Also, you must write something like `-t sch.tab` to select a table in a particular schema, rather than the old locution of `-n sch -t tab`.

- `--exclude-table=table`

  Do not dump any tables matching the `table` pattern. The pattern is interpreted according to the same rules as for `-t`. `-T` can be given more than once to exclude tables matching any of several patterns.

  When both `-t` and `-T` are given, the behavior is to dump just the tables that match at least one `-t` switch but no `-T` switches. If `-T` appears without `-t`, then tables matching `-T` are excluded from what is otherwise a normal dump.

- `-v`

  Specifies verbose mode. This will cause `pg_dump` to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

- `-x`

  Prevent dumping of access privileges (grant/revoke commands).

- `--disable-dollar-quoting`

  This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

- `--disable-triggers`

  This option is only relevant when creating a data-only dump. It instructs `pg_dump` to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

  Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser.

  This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

- `--use-set-session-authorization`

  Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, might not restore properly. Also, a dump using `SET SESSION AUTHORIZATION` will certainly require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.
Specify the compression level to use. Zero means no compression. For the custom archive format, this specifies compression of individual table-data segments, and the default is to compress at a moderate level. For plain text output, setting a nonzero compression level causes the entire output file to be compressed, as though it had been fed through gzip; but the default is not to compress. The tar archive format currently does not support compression at all.

The following command-line options control the database connection parameters.

```
-Z 0..9
--compress=0..9
```

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the PGHOST environment variable, if set, else a Unix domain socket connection is attempted.

```
-p port
--port=port
```

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the PGPORT environment variable, if set, or a compiled-in default.

```
-U username
--username=username
```

User name to connect as.

```
-W
--password
```

Force pg_dump to prompt for a password before connecting to a database.

This option is never essential, since pg_dump will automatically prompt for a password if the server demands password authentication. However, pg_dump will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `\W` to avoid the extra connection attempt.

### Environment

PGDATABASE
PGHOST
PGPORT
PGUSER

Default connection parameters.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).
Diagnostics

pg_dump internally executes SELECT statements. If you have problems running pg_dump, make sure you are able to select information from the database using, for example, psql. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

If your database cluster has any local additions to the template1 database, be careful to restore the output of pg_dump into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from template0 not template1, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_dump has a limitation; when a data-only dump is chosen and the option --disable-triggers is used, pg_dump emits commands to disable triggers on user tables before inserting the data and commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs might be left in the wrong state.

Members of tar archives are limited to a size less than 8 GB. (This is an inherent limitation of the tar file format.) Therefore this format cannot be used if the textual representation of any one table exceeds that size. The total size of a tar archive and any of the other output formats is not limited, except possibly by the operating system.

The dump file produced by pg_dump does not contain the statistics used by the optimizer to make query planning decisions. Therefore, it is wise to run ANALYZE after restoring from a dump file to ensure good performance. The dump file also does not contain any ALTER DATABASE ... SET commands; these settings are dumped by pg_dumpall, along with database users and other installation-wide settings.

Because pg_dump is used to transfer data to newer versions of PostgreSQL, the output of pg_dump can be loaded into newer PostgreSQL databases. It also can read older PostgreSQL databases. However, it usually cannot read newer PostgreSQL databases or produce dump output that can be loaded into older database versions. To do this, manual editing of the dump file might be required.

Examples

To dump a database called mydb into a SQL-script file:

```
$ pg_dump mydb > db.sql
```

To reload such a script into a (freshly created) database named newdb:

```
$ psql -d newdb -f db.sql
```

To dump a database into a custom-format archive file:

```
$ pg_dump --Fc mydb > db.dump
```

1195
To reload an archive file into a (freshly created) database named newdb:

```bash
$ pg_restore -d newdb db.dump
```

To dump a single table named mytab:

```bash
$ pg_dump -t mytab mydb > db.sql
```

To dump all tables whose names start with `emp` in the `detroit` schema, except for the table named `employee_log`:

```bash
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

To dump all schemas whose names start with `east` or `west` and end in `gsm`, excluding any schemas whose names contain the word `test`:

```bash
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

The same, using regular expression notation to consolidate the switches:

```bash
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

To dump all database objects except for tables whose names begin with `ts_`:

```bash
$ pg_dump -T 'ts_*' mydb > db.sql
```

To specify an upper-case or mixed-case name in `-t` and related switches, you need to double-quote the name; else it will be folded to lower case (see Patterns). But double quotes are special to the shell, so in turn they must be quoted. Thus, to dump a single table with a mixed-case name, you need something like

```bash
$ pg_dump -t "MixedCaseName" mydb > mytab.sql
```

See Also

`pg_dumpall`, `pg_restore`, `psql`
**pg_dumpall**

**Name**
pg_dumpall — extract a PostgreSQL database cluster into a script file

**Synopsis**
pg_dumpall [option...]

**Description**
pg_dumpall is a utility for writing out (“dumping”) all PostgreSQL databases of a cluster into one script file. The script file contains SQL commands that can be used as input to psql to restore the databases. It does this by calling pg_dump for each database in a cluster. pg_dumpall also dumps global objects that are common to all databases. (pg_dump does not save these objects.) This currently includes information about database users and groups, and access permissions that apply to databases as a whole.

Since pg_dumpall reads tables from all databases you will most likely have to connect as a database superuser in order to produce a complete dump. Also you will need superuser privileges to execute the saved script in order to be allowed to add users and groups, and to create databases.

The SQL script will be written to the standard output. Shell operators should be used to redirect it into a file.

pg_dumpall needs to connect several times to the PostgreSQL server (once per database). If you use password authentication it will ask for a password each time. It is convenient to have a ~/.pgpass file in such cases. See Section 30.13 for more information.

**Options**
The following command-line options control the content and format of the output.

- `a`
  --data-only
  Dump only the data, not the schema (data definitions).

- `c`
  --clean
  Include SQL commands to clean (drop) databases before recreating them. DROP commands for roles and tablespaces are added as well.

- `d`
  --inserts
  Dump data as INSERT commands (rather than COPY). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. Note that the restore might fail altogether if you have rearranged column order. The -d option is safer, though even slower.
Dump data as INSERT commands with explicit column names (INSERT INTO table (column, ...) VALUES ...). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases.

Send output to the specified file. If this is omitted, the standard output is used.

Dump only global objects (roles and tablespaces), no databases.

Ignore version mismatch between pg_dumpall and the database server.

pg_dumpall can handle databases from previous releases of PostgreSQL, but very old versions are not supported anymore (currently prior to 7.0). Use this option if you need to override the version check (and if pg_dumpall then fails, don’t say you weren’t warned).

Dump object identifiers (OIDs) as part of the data for every table. Use this option if your application references the OID columns in some way (e.g., in a foreign key constraint). Otherwise, this option should not be used.

Do not output commands to set ownership of objects to match the original database. By default, pg_dumpall issues ALTER OWNER or SET SESSION AUTHORIZATION statements to set ownership of created schema elements. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify --no-owner.

Dump only roles, no databases or tablespaces.

Dump only the object definitions (schema), not data.

Specify the superuser user name to use when disabling triggers. This is only relevant if --disable-triggers is used. (Usually, it’s better to leave this out, and instead start the resulting script as superuser.)
pg_dumpall

-t
--tablespaces-only

    Dump only tablespaces, no databases or roles.

-v
--verbose

    Specifies verbose mode. This will cause pg_dumpall to output start/stop times to the dump file, and progress messages to standard error. It will also enable verbose output in pg_dump.

-x
--no-privileges
--no-acl

    Prevent dumping of access privileges (grant/revoke commands).

--disable-dollar-quoting

    This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

--disable-triggers

    This option is only relevant when creating a data-only dump. It instructs pg_dumpall to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

    Presently, the commands emitted for --disable-triggers must be done as superuser. So, you should also specify a superuser name with -S, or preferably be careful to start the resulting script as a superuser.

--use-set-session-authorization

    Output SQL-standard SET SESSION AUTHORIZATION commands instead of ALTER OWNER commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, might not restore properly.

The following command-line options control the database connection parameters.

-h host
--host=host

    Specifies the host name of the machine on which the database server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the PGHOST environment variable, if set, else a Unix domain socket connection is attempted.

-d dbname
--database=dbname

    Specifies the name of the database to connect to to dump global objects and discover what other databases should be dumped. If not specified, the “postgres” database will be used, and if that does not exist, “template1” will be used.

-p port
--port=port

    Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the PGPORT environment variable, if set, or a compiled-in default.
-U username
--username=username
  User name to connect as.
-W
--password
  Force pg_dumpall to prompt for a password before connecting to a database.

This option is never essential, since pg_dumpall will automatically prompt for a password if the server demands password authentication. However, pg_dumpall will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing -W to avoid the extra connection attempt.

Note that the password prompt will occur again for each database to be dumped. Usually, it’s better to set up a ~/.pgpass file than to rely on manual password entry.

Environment

PGHOST
PGPORT
PGUSER

  Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

Notes

Since pg_dumpall calls pg_dump internally, some diagnostic messages will refer to pg_dump.

Once restored, it is wise to run ANALYZE on each database so the optimizer has useful statistics. You can also run vacuumdb -a -z to analyze all databases.

pg_dumpall requires all needed tablespace directories to exist before the restore or database creation will fail for databases in non-default locations.

Examples

To dump all databases:

```
$ pg_dumpall > db.out
```

To reload this database use, for example:

```
$ psql -f db.out postgres
```

(It is not important to which database you connect here since the script file created by pg_dumpall will contain the appropriate commands to create and connect to the saved databases.)
See Also

Check pg_dump for details on possible error conditions.
pg_restore

Name
pg_restore — restore a PostgreSQL database from an archive file created by pg_dump

Synopsis
pg_restore [option...] [filename]

Description
pg_restore is a utility for restoring a PostgreSQL database from an archive created by pg_dump in one of the non-plain-text formats. It will issue the commands necessary to reconstruct the database to the state it was in at the time it was saved. The archive files also allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive files are designed to be portable across architectures.

pg_restore can operate in two modes. If a database name is specified, pg_restore connects to that database and restores archive contents directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created and written to a file or standard output. This script output is equivalent to the plain text output format of pg_dump. Some of the options controlling the output are therefore analogous to pg_dump options.

Obviously, pg_restore cannot restore information that is not present in the archive file. For instance, if the archive was made using the “dump data as INSERT commands” option, pg_restore will not be able to load the data using COPY statements.

Options
pg_restore accepts the following command line arguments.

filename
Specifies the location of the archive file to be restored. If not specified, the standard input is used.

-a
--data-only
    Restore only the data, not the schema (data definitions).

-c
--clean
    Clean (drop) database objects before recreating them.

-C
--create
    Create the database before restoring into it. (When this option is used, the database named with -d is used only to issue the initial CREATE DATABASE command. All data is restored into the database name that appears in the archive.)
Connect to database `dbname` and restore directly into the database.

Exit if an error is encountered while sending SQL commands to the database. The default is to continue and to display a count of errors at the end of the restoration.

Specify output file for generated script, or for the listing when used with `-l`. Default is the standard output.

Specify format of the archive. It is not necessary to specify the format, since `pg_restore` will determine the format automatically. If specified, it can be one of the following:

- `tar`
  The archive is a `tar` archive.

- `custom`
  The archive is in the custom format of `pg_dump`.

Ignore database version checks.

Restore definition of named index only.

List the contents of the archive. The output of this operation can be used as input to the `-L` option. Note that if filtering switches such as `-n` or `-t` are used with `-l`, they will restrict the items listed.

Restore only those archive elements that are listed in `list-file`, and restore them in the order they appear in the file. Note that if filtering switches such as `-n` or `-t` are used with `-L`, they will further restrict the items restored.

`list-file` is normally created by editing the output of a previous `-l` operation. Lines can be moved or removed, and can also be commented out by placing a semicolon (`;`) at the start of the line. See below for examples.
-n namespace
--schema=schema

Restore only objects that are in the named schema. This can be combined with the -t option to restore just a specific table.

-O
--no-owner

Do not output commands to set ownership of objects to match the original database. By default, pg_restore issues ALTER OWNER or SET SESSION AUTHORIZATION statements to set ownership of created schema elements. These statements will fail unless the initial connection to the database is made by a superuser (or the same user that owns all of the objects in the script). With -O, any user name can be used for the initial connection, and this user will own all the created objects.

-P function-name(argtype [, ...])
--function=function-name(argtype [, ...])

Restore the named function only. Be careful to spell the function name and arguments exactly as they appear in the dump file’s table of contents.

-R
--no-reconnect

This option is obsolete but still accepted for backwards compatibility.

-s
--schema-only

Restore only the schema (data definitions), not the data (table contents). Sequence current values will not be restored, either. (Do not confuse this with the --schema option, which uses the word “schema” in a different meaning.)

-S username
--superuser=username

Specify the superuser user name to use when disabling triggers. This is only relevant if --disable-triggers is used.

-t table
--table=table

Restore definition and/or data of named table only.

-T trigger
--trigger=trigger

Restore named trigger only.

-v
--verbose

Specifies verbose mode.

-x
--no-privileges
--no-acl

Prevent restoration of access privileges (grant/revoke commands).
pg_restore

--disable-triggers

This option is only relevant when performing a data-only restore. It instructs pg_restore to execute commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for --disable-triggers must be done as superuser. So, you should also specify a superuser name with -S, or preferably run pg_restore as a PostgreSQL superuser.

--use-set-session-authorization

Output SQL-standard SET SESSION AUTHORIZATION commands instead of ALTER OWNER commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, might not restore properly.

--no-data-for-failed-tables

By default, table data is restored even if the creation command for the table failed (e.g., because it already exists). With this option, data for such a table is skipped. This behavior is useful if the target database already contains the desired table contents. For example, auxiliary tables for PostgreSQL extensions such as PostGIS might already be loaded in the target database; specifying this option prevents duplicate or obsolete data from being loaded into them.

This option is effective only when restoring directly into a database, not when producing SQL script output.

pg_restore also accepts the following command line arguments for connection parameters:

-h host
--host=host

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the PGHOST environment variable, if set, else a Unix domain socket connection is attempted.

-p port
--port=port

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the PGPORT environment variable, if set, or a compiled-in default.

-U username
--username=username

User name to connect as.

-W
--password

Force pg_restore to prompt for a password before connecting to a database.

This option is never essential, since pg_restore will automatically prompt for a password if the server demands password authentication. However, pg_restore will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing -W to avoid the extra connection attempt.
-1
--single-transaction

Execute the restore as a single transaction (that is, wrap the emitted commands in `BEGIN/COMMIT`). This ensures that either all the commands complete successfully, or no changes are applied. This option implies `--exit-on-error`.

Environment

PGHOST
PGPORT
PGUSER

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

Diagnostics

When a direct database connection is specified using the `-d` option, pg_restore internally executes SQL statements. If you have problems running pg_restore, make sure you are able to select information from the database using, for example, psql. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

If your installation has any local additions to the `template1` database, be careful to load the output of pg_restore into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

The limitations of pg_restore are detailed below.

- When restoring data to a pre-existing table and the option `--disable-triggers` is used, pg_restore emits commands to disable triggers on user tables before inserting the data then emits commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs might be left in the wrong state.
- pg_restore will not restore large objects for a single table. If an archive contains large objects, then all large objects will be restored.

See also the pg_dump documentation for details on limitations of pg_dump.

Once restored, it is wise to run `ANALYZE` on each restored table so the optimizer has useful statistics.
Examples

Assume we have dumped a database called `mydb` into a custom-format dump file:

```
$ pg_dump -Fc mydb > db.dump
```

To drop the database and recreate it from the dump:

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

The database named in the `-d` switch can be any database existing in the cluster; `pg_restore` only uses it to issue the `CREATE DATABASE` command for `mydb`. With `-C`, data is always restored into the database name that appears in the dump file.

To reload the dump into a new database called `newdb`:

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

Notice we don’t use `-C`, and instead connect directly to the database to be restored into. Also note that we clone the new database from `template0` not `template1`, to ensure it is initially empty.

To reorder database items, it is first necessary to dump the table of contents of the archive:

```
$ pg_restore -l db.dump > db.list
```

The listing file consists of a header and one line for each item, e.g.:

```
; ; Archive created at Fri Jul 28 22:28:36 2000
; ; dbname: mydb
; ; TOC Entries: 74
; ; Compression: 0
; ; Dump Version: 1.4-0
; ; Format: CUSTOM
;
; ; Selected TOC Entries:
;
2; 145344 TABLE species postgres
3; 145344 ACL species
4; 145359 TABLE nt_header postgres
5; 145359 ACL nt_header
6; 145402 TABLE species_records postgres
7; 145402 ACL species_records
8; 145416 TABLE ss_old postgres
9; 145416 ACL ss_old
10; 145433 TABLE map_resolutions postgres
11; 145433 ACL map_resolutions
12; 145443 TABLE hs_old postgres
13; 145443 ACL hs_old
```

Semicolons start a comment, and the numbers at the start of lines refer to the internal archive ID assigned to each item.

Lines in the file can be commented out, deleted, and reordered. For example:
pg_restore

10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres

could be used as input to pg_restore and would only restore items 10 and 6, in that order:

$ pg_restore -L db.list db.dump

See Also

pg_dump, pg_dumpall, psql
psql

Name

psql — PostgreSQL interactive terminal

Synopsis

psql [option...] [dbname [username]]

Description

psql is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Options

-a
--echo-all
Print all input lines to standard output as they are read. This is more useful for script processing rather than interactive mode. This is equivalent to setting the variable ECHO to all.

-A
--no-align
Switches to unaligned output mode. (The default output mode is otherwise aligned.)

-c command
--command command
Specifies that psql is to execute one command string, command, and then exit. This is useful in shell scripts.

command must be either a command string that is completely parsable by the server (i.e., it contains no psql specific features), or a single backslash command. Thus you cannot mix SQL and psql meta-commands with this option. To achieve that, you could pipe the string into psql, like this: echo '\x \x SELECT * FROM foo;' | psql. (\x is the separator meta-command.)

If the command string contains multiple SQL commands, they are processed in a single transaction, unless there are explicit BEGIN/COMMIT commands included in the string to divide it into multiple transactions. This is different from the behavior when the same string is fed to psql's standard input.

-d dbname
--dbname dbname
Specifies the name of the database to connect to. This is equivalent to specifying dbname as the first non-option argument on the command line.
If this parameter contains an = sign, it is treated as a conninfo string. See Section 30.1 for more information.

-e
   --echo-queries
   Copy all SQL commands sent to the server to standard output as well. This is equivalent to setting the variable ECHO to queries.

-E
   --echo-hidden
   Echo the actual queries generated by \d and other backslash commands. You can use this to study psql’s internal operations. This is equivalent to setting the variable ECHO_HIDDEN from within psql.

-f filename
   --file filename
   Use the file filename as the source of commands instead of reading commands interactively. After the file is processed, psql terminates. This is in many ways equivalent to the internal command \i.

   If filename is - (hyphen), then standard input is read.

   Using this option is subtly different from writing psql < filename. In general, both will do what you expect, but using -f enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the start-up overhead. On the other hand, the variant using the shell’s input redirection is (in theory) guaranteed to yield exactly the same output that you would have gotten had you entered everything by hand.

-F separator
   --field-separator separator
   Use separator as the field separator for unaligned output. This is equivalent to \pset fieldsep or \f.

-h hostname
   --host hostname
   Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix-domain socket.

-H
   --html
   Turn on HTML tabular output. This is equivalent to \pset format html or the \H command.

-l
   --list
   List all available databases, then exit. Other non-connection options are ignored. This is similar to the internal command \list.

-L filename
   --log-file filename
   Write all query output into file filename, in addition to the normal output destination.

-n
   --no-readline
   Do not use readline for line editing and do not use the history. This can be useful to turn off tab expansion when cutting and pasting.
-o filename
--output filename

Put all query output into file filename. This is equivalent to the command \o.

-p port
--port port

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the PGPORT environment variable or, if not set, to the port specified at compile time, usually 5432.

-P assignment
--pset assignment

Allows you to specify printing options in the style of \pset on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write -P format=latex.

-q
--quiet

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the -c option. Within psql you can also set the QUIET variable to achieve the same effect.

-R separator
--record-separator separator

Use separator as the record separator for unaligned output. This is equivalent to the \pset recordsep command.

-s
--single-step

Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

-S
--single-line

Runs in single-line mode where a newline terminates an SQL command, as a semicolon does.

Note: This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

-t
--tuples-only

Turn off printing of column names and result row count footers, etc. This is equivalent to the \t command.

-T table_options
--table-attr table_options

Allows you to specify options to be placed within the HTML table tag. See \pset for details.
psql

-\U username
--username username

Connect to the database as the user username instead of the default. (You must have permission to do so, of course.)

-\V assignment
--set assignment
--variable assignment

Perform a variable assignment, like the \set internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To just set a variable without a value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes might get overwritten later.

-\V
--version

Print the psql version and exit.

-\W
--password

Force psql to prompt for a password before connecting to a database.

This option is never essential, since psql will automatically prompt for a password if the server demands password authentication. However, psql will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing -\W to avoid the extra connection attempt.

Note that this option will remain set for the entire session, and so it affects uses of the meta-command \connect as well as the initial connection attempt.

-\x
--expanded

Turn on the expanded table formatting mode. This is equivalent to the \x command.

-\x,
--no-psqlrc

Do not read the start-up file (neither the system-wide psqlrc file nor the user’s ~/.psqlrc file).

-\1
--single-transaction

When psql executes a script with the -f option, adding this option wraps BEGIN/COMMIT around the script to execute it as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

If the script itself uses BEGIN, COMMIT, or ROLLBACK, this option will not have the desired effects. Also, if the script contains any command that cannot be executed inside a transaction block, specifying this option will cause that command (and hence the whole transaction) to fail.

-? --help

Show help about psql command line arguments, and exit.
Exit Status

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the server went bad and the session was not interactive, and 3 if an error occurred in a script and the variable ON_ERROR_STOP was set.

Usage

Connecting To A Database

psql is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the host name and port number of the server and what user name you want to connect as. psql can be told about those parameters via command line options, namely -d, -h, -p, and -U respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is already given). Not all these options are required; there are useful defaults. If you omit the host name, psql will connect via a Unix-domain socket to a server on the local host, or via TCP/IP to localhost on machines that don’t have Unix-domain sockets. The default port number is determined at compile time. Since the database server uses the same default, you will not have to specify the port in most cases. The default user name is your Unix user name, as is the default database name. Note that you cannot just connect to any database under any user name. Your database administrator should have informed you about your access rights.

When the defaults aren’t quite right, you can save yourself some typing by setting the environment variables PGDATABASE, PGHOST, PGPORT and/or PGUSER to appropriate values. (For additional environment variables, see Section 30.12.) It is also convenient to have a ~/.pgpass file to avoid regularly having to type in passwords. See Section 30.13 for more information.

An alternative way to specify connection parameters is in a conninfo string, which is used instead of a database name. This mechanism give you very wide control over the connection. For example:

```
$ psql "service=myservice sslmode=require"
```

This way you can also use LDAP for connection parameter lookup as described in Section 30.15. See Section 30.1 for more information on all the available connection options.

If the connection could not be made for any reason (e.g., insufficient privileges, server is not running on the targetted host, etc.), psql will return an error and terminate.

Entering SQL Commands

In normal operation, psql provides a prompt with the name of the database to which psql is currently connected, followed by the string -->. For example:

```
$ psql testdb
Welcome to psql 8.3.23, the PostgreSQL interactive terminal.
```

Type: \
copyright for distribution terms
\h for help with SQL commands
\? for help with psql commands
\g or terminate with semicolon to execute query
\q to quit
At the prompt, the user can type in SQL commands. Ordinarily, input lines are sent to the server when a command-terminating semicolon is reached. An end of line does not terminate a command. Thus commands can be spread over several lines for clarity. If the command was sent and executed without error, the results of the command are displayed on the screen.

Whenever a command is executed, psql also polls for asynchronous notification events generated by \listen and \notify.

Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processed by psql itself. These commands help make psql more useful for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace into an argument you can quote it with a single quote. To include a single quote into such an argument, use two single quotes. Anything contained in single quotes is furthermore subject to C-like substitutions for \n (new line), \t (tab), \digits (octal), and \x\digits (hexadecimal).

If an unquoted argument begins with a colon (:), it is taken as a psql variable and the value of the variable is used as the argument instead.

Arguments that are enclosed in backquotes (`) are taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) is taken as the argument value. The above escape sequences also apply in backquotes.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes ("") protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, FOO"BAR"BAZ is interpreted as fooBARbaz, and "A weird"" name" becomes A weird" name.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\ ` (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

\a

If the current table output format is unaligned, it is switched to aligned. If it is not unaligned, it is set to unaligned. This command is kept for backwards compatibility. See \psset for a more general solution.

\cd [ directory ]

Changes the current working directory to directory. Without argument, changes to the current user’s home directory.

Tip: To print your current working directory, use \ ! pwd.
\C [ title ]

Sets the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to \pset title title. (The name of this command derives from “caption”, as it was previously only used to set the caption in an HTML table.)

\connect (or \c) [ dbname [ username ] [ host ] [ port ] ]

Establishes a new connection to a PostgreSQL server. If the new connection is successfully made, the previous connection is closed. If any of dbname, username, host or port are omitted or specified as -, the value of that parameter from the previous connection is used. If there is no previous connection, the libpq default for the parameter’s value is used.

If the connection attempt failed (wrong user name, access denied, etc.), the previous connection will only be kept if psqll is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

\copy { table [ ( column_list ) ] | ( query ) } { from | to } { filename
| stdin | stdout | stdin | stdout } [ with | [ binary ] [ oids ] [ delimiter [ as | \ ' character ' ] [ null [ as | \ ' string ' ] [ csv [ header ] [ quote [ as | \ ' character ' ] [ escape [ as | \ ' character ' ] [ force quote column_list ] [ force not null column_list ] ]

Performs a frontend (client) copy. This is an operation that runs an SQL COPY command, but instead of the server reading or writing the specified file, psqll reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

The syntax of the command is similar to that of the SQL COPY command. Note that, because of this, special parsing rules apply to the \copy command. In particular, the variable substitution rules and backslash escapes do not apply.

\copy ... from stdin | to stdout reads/writes based on the command input and output respectively. All rows are read from the same source that issued the command, continuing until \. is read or the stream reaches EOF. Output is sent to the same place as command output. To read/write from psqll’s standard input or output, use stdin or stdout. This option is useful for populating tables in-line within a SQL script file.

Tip: This operation is not as efficient as the SQL COPY command because all data must pass through the client/server connection. For large amounts of data the SQL command might be preferable.

\copyright

Shows the copyright and distribution terms of PostgreSQL.

\d [ pattern ]
\d+ [ pattern ]

For each relation (table, view, index, or sequence) matching the pattern, show all columns, their types, the tablespace (if not the default) and any special attributes such as NOT NULL or defaults, if any. Associated indexes, constraints, rules, and triggers are also shown, as is the view definition if the relation is a view. (“Matching the pattern” is defined below.)
The command form \d+ is identical, except that more information is displayed: any comments associated with the columns of the table are shown, as is the presence of OIDs in the table.

**Note:** If \d is used without a pattern argument, it is equivalent to \dtvs which will show a list of all tables, views, and sequences. This is purely a convenience measure.

\da [ pattern ]

Lists all available aggregate functions, together with their return type and the data types they operate on. If pattern is specified, only aggregates whose names match the pattern are shown.

\db [ pattern ]
\db+ [ pattern ]

Lists all available tablespaces. If pattern is specified, only tablespaces whose names match the pattern are shown. If + is appended to the command name, each object is listed with its associated permissions.

\dc [ pattern ]

Lists all available conversions between character-set encodings. If pattern is specified, only conversions whose names match the pattern are listed.

\dC

Lists all available type casts.

\dd [ pattern ]

Shows the descriptions of objects matching the pattern, or of all visible objects if no argument is given. But in either case, only objects that have a description are listed. (“Object” covers aggregates, functions, operators, types, relations (tables, views, indexes, sequences, large objects), rules, and triggers.) For example:

=> \dd version

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>version</td>
<td>function</td>
<td>PostgreSQL version string</td>
</tr>
</tbody>
</table>

(1 row)

Descriptions for objects can be created with the **COMMENT** SQL command.

\dD [ pattern ]

Lists all available domains. If pattern is specified, only matching domains are shown.

\df [ pattern ]
\df+ [ pattern ]

Lists available functions, together with their argument and return types. If pattern is specified, only functions whose names match the pattern are shown. If the form \df+ is used, additional information about each function, including volatility, language, source code and description, is shown.

**Note:** To look up functions taking argument or returning values of a specific type, use your pager’s search capability to scroll through the \df output.

To reduce clutter, \df does not show data type I/O functions. This is implemented by ignoring functions that accept or return type cstring.
Lists available text search configurations. If pattern is specified, only configurations whose names match the pattern are shown. If the form \dF+ is used, a full description of each configuration is shown, including the underlying text search parser and the dictionary list for each parser token type.

Lists available text search dictionaries. If pattern is specified, only dictionaries whose names match the pattern are shown. If the form \dF+ is used, additional information is shown about each selected dictionary, including the underlying text search template and the option values.

Lists available text search parsers. If pattern is specified, only parsers whose names match the pattern are shown. If the form \dFp+ is used, a full description of each parser is shown, including the underlying functions and the list of recognized token types.

Lists available text search templates. If pattern is specified, only templates whose names match the pattern are shown. If the form \dFt+ is used, additional information is shown about each template, including the underlying function names.

Lists all database roles. If pattern is specified, only those roles whose names match the pattern are listed. (This command is now effectively the same as \du.)

This is not the actual command name: the letters i, s, t, v, S stand for index, sequence, table, view, and system table, respectively. You can specify any or all of these letters, in any order, to obtain a listing of all the matching objects. The letter S restricts the listing to system objects; without S, only non-system objects are shown. If + is appended to the command name, each object is listed with its associated description, if any.

If pattern is specified, only objects whose names match the pattern are listed.

This is an alias for \lo_list, which shows a list of large objects.

Lists all available schemas (namespaces). If pattern (a regular expression) is specified, only schemas whose names match the pattern are listed. Non-local temporary schemas are suppressed. If + is appended to the command name, each object is listed with its associated permissions and description, if any.

Lists available operators with their operand and return types. If pattern is specified, only operators whose names match the pattern are listed.
\dp [ pattern ]

   Produces a list of all available tables, views and sequences with their associated access privileges.  
   If pattern is specified, only tables, views and sequences whose names match the pattern are  
   listed.

   The GRANT and REVOKE commands are used to set access privileges.

\dT [ pattern ]
\dT+ [ pattern ]

   Lists all data types or only those that match pattern. The command form \dT+ shows extra  
   information.

\du [ pattern ]

   Lists all database roles, or only those that match pattern.

\edit (or \e) [ filename ]

   If filename is specified, the file is edited; after the editor exits, its content is copied back to the  
   query buffer. If no argument is given, the current query buffer is copied to a temporary file which  
   is then edited in the same fashion.

   The new query buffer is then re-parsed according to the normal rules of psql, where the whole  
   buffer is treated as a single line. (Thus you cannot make scripts this way. Use \i for that.)  
   This means also that if the query ends with (or rather contains) a semicolon, it is immediately  
   executed. In other cases it will merely wait in the query buffer.

   Tip: psql searches the environment variables PSQL_EDITOR, EDITOR, and VISUAL (in that  
   order) for an editor to use. If all of them are unset, vi is used on Unix systems, notepad.exe  
   on Windows systems.

\echo text [ ... ]

   Prints the arguments to the standard output, separated by one space and followed by a newline.  
   This can be useful to intersperse information in the output of scripts. For example:
   
   => \echo \"date\"  
   Tue Oct 26 21:40:57 CEST 1999

   If the first argument is an unquoted \-n the trailing newline is not written.

   Tip: If you use the \o command to redirect your query output you might wish to use \qecho  
   instead of this command.

\encoding [ encoding ]

   Sets the client character set encoding. Without an argument, this command shows the current  
   encoding.

\f [ string ]

   Sets the field separator for unaligned query output. The default is the vertical bar (\). See also  
   \pset for a generic way of setting output options.

\g [ [ filename\|command ] ]

   Sends the current query input buffer to the server and optionally stores the query’s output in  
   filename or pipes the output into a separate Unix shell executing command. A bare \g is
virtually equivalent to a semicolon. A \g with argument is a “one-shot” alternative to the \o command.

\help (or \h) [ command ]

Gives syntax help on the specified SQL command. If command is not specified, then psql will list all the commands for which syntax help is available. If command is an asterisk (*), then syntax help on all SQL commands is shown.

Note: To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type \help alter table.

\H

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see \pset about setting other output options.

\i filename

Reads input from the file filename and executes it as though it had been typed on the keyboard.

Note: If you want to see the lines on the screen as they are read you must set the variable ECHO to all.

\l (or \list)
\l+ (or \list+)

List the names, owners, and character set encodings of all the databases in the server. If + is appended to the command name, database descriptions are also displayed.

\lo_export loid filename

Reads the large object with OID loid from the database and writes it to filename. Note that this is subtly different from the server function lo_export, which acts with the permissions of the user that the database server runs as and on the server’s file system.

Tip: Use \lo_list to find out the large object’s OID.

\lo_import filename [ comment ]

Stores the file into a PostgreSQL large object. Optionally, it associates the given comment with the object. Example:

```bash
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object ID 152801, which can be used to access the newly-created large object in the future. For the sake of readability, it is recommended to always associate a human-readable comment with every object. Both OIDs and comments can be viewed with the \lo_list command.

Note that this command is subtly different from the server-side lo_import because it acts as the local user on the local file system, rather than the server’s user and file system.
\lo_list

Shows a list of all PostgreSQL large objects currently stored in the database, along with any
comments provided for them.

\lo_unlink loid

Deletes the large object with OID \texttt{loid} from the database.

\textbf{Tip:} Use \texttt{\lo_list} to find out the large object’s OID.

/o \[ \{filename\mid\text{command}\} \]

Saves future query results to the file \texttt{filename} or pipes future results into a separate Unix shell
to execute \texttt{command}. If no arguments are specified, the query output will be reset to the standard
output.

“Query results” includes all tables, command responses, and notices obtained from the database
server, as well as output of various backslash commands that query the database (such as \texttt{\d}),
but not error messages.

\textbf{Tip:} To intersperse text output in between query results, use \texttt{\qecho}.

\p

Print the current query buffer to the standard output.

\password \[ \texttt{username} \]

Changes the password of the specified user (by default, the current user). This command prompts
for the new password, encrypts it, and sends it to the server as an \texttt{ALTER ROLE} command. This
makes sure that the new password does not appear in cleartext in the command history, the server
log, or elsewhere.

\prompt \[ \texttt{text} \] \texttt{name}

Prompts the user to set variable \texttt{name}. An optional prompt, \texttt{text}, can be specified. (For multi-
word prompts, use single-quotes.)

By default, \texttt{\prompt} uses the terminal for input and output. However, if the \texttt{-f} command line
switch is used, \texttt{\prompt} uses standard input and standard output.

\pset parameter \[ \texttt{value} \]

This command sets options affecting the output of query result tables. \texttt{parameter} describes
which option is to be set. The semantics of \texttt{value} depend thereon.

Adjustable printing options are:

\texttt{format}

Sets the output format to one of \texttt{unaligned}, \texttt{aligned}, \texttt{html}, \texttt{latex}, or \texttt{troff-ms}.
Unique abbreviations are allowed. (That would mean one letter is enough.)

“Unaligned” writes all columns of a row on a line, separated by the currently active field
separator. This is intended to create output that might be intended to be read in by other pro-
grams (tab-separated, comma-separated). “Aligned” mode is the standard, human-readable,
nicely formatted text output that is default. The “HTML” and “LaTeX” modes put out tables
that are intended to be included in documents using the respective mark-up language. They
are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

**border**

The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border-...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

**expanded (or x)**

You can specify an optional second argument, if it is provided it may be either on or off which will enable or disable expanded mode. If the second argument is not provided then we will toggle between regular and expanded format. When expanded format is enabled, query results are displayed in two columns, with the column name on the left and the data on the right. This mode is useful if the data wouldn’t fit on the screen in the normal “horizontal” mode.

Expanded mode is supported by all four output formats.

**null**

The second argument is a string that should be printed whenever a column is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null '(null)'`.

**fieldsep**

Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is ‘|’ (a vertical bar).

**footer**

You can specify an optional second argument, if it is provided it may be either on or off which will enable or disable display of the default footer (x rows). If the second argument is not provided then we will toggle between on and off.

**numericlocale**

You can specify an optional second argument, if it is provided it may be either on or off which will enable or disable display of a locale-aware character to separate groups of digits to the left of the decimal marker. If the second argument is not provided then we will toggle between on and off.

**recordsep**

Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.

**tuples_only (or t)**

You can specify an optional second argument, if it is provided it may be either on or off which will enable or disable the tuples only mode. If the second argument is not provided then we will toggle between tuples only and full display. Full display shows extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.
title [ text ]

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.

tableattr (or T) [ text ]

Allows you to specify any attributes to be placed inside the HTML table tag. This could for example be cellpadding or bgcolor. Note that you probably don’t want to specify border here, as that is already taken care of by \pset border.

pager

Controls use of a pager for query and psql help output. If the environment variable PAGER is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as more) is used.

When the pager is off, the pager is not used. When the pager is on, the pager is used only when appropriate, i.e. the output is to a terminal and will not fit on the screen. (psql does not do a perfect job of estimating when to use the pager.) \pset pager turns the pager on and off. Pager can also be set to always, which causes the pager to be always used, or you can set the pager to on which will enable the usage of the pager when appropriate, or you can set the pager to off which will disable the pager.

Illustrations on how these different formats look can be seen in the Examples section.

Tip: There are various shortcut commands for \pset. See \a, \C, \H, \t, \T, and \x.

Note: It is an error to call \pset without arguments. In the future this call might show the current status of all printing options.

\q or \quit

Quits the psql program. In a script file, only execution of that script is terminated.

\qecho text [ ... ]

This command is identical to \echo except that the output will be written to the query output channel, as set by \o.

\r

Resets (clears) the query buffer.

\s [ filename ]

Print or save the command line history to filename. If filename is omitted, the history is written to the standard output. This option is only available if psql is configured to use the GNU Readline library.

\set [ name [ value [ ... ] ] ]

Sets the internal variable name to value or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with no value. To unset a variable, use the \unset command.

Valid variable names can contain characters, digits, and underscores. See the section Variables below for details. Variable names are case-sensitive.
Although you are welcome to set any variable to anything you want, psql treats several variables as special. They are documented in the section about variables.

Note: This command is totally separate from the SQL command SET.

\t
  Toggles the display of output column name headings and row count footer. This command is equivalent to \pset tuples_only and is provided for convenience.

\T table_options
  Allows you to specify attributes to be placed within the table tag in HTML tabular output mode. This command is equivalent to \pset tableattr table_options.

\timing
  Toggles a display of how long each SQL statement takes, in milliseconds.

\w [filename | command]
  Outputs the current query buffer to the file filename or pipes it to the Unix command command.

\x
  Toggles expanded table formatting mode. As such it is equivalent to \pset expanded.

\z [ pattern ]
  Produces a list of all available tables, views and sequences with their associated access privileges. If a pattern is specified, only tables, views and sequences whose names match the pattern are listed.

  The GRANT and REVOKE commands are used to set access privileges.

  This is an alias for \dp ("display privileges").

\! [ command ]
  Escapes to a separate Unix shell or executes the Unix command command. The arguments are not further interpreted, the shell will see them as is.

\?
  Shows help information about the backslash commands.

Patterns

The various \d commands accept a pattern parameter to specify the object name(s) to be displayed. In the simplest case, a pattern is just the exact name of the object. The characters within a pattern are normally folded to lower case, just as in SQL names; for example, \dt FOO will display the table named foo. As in SQL names, placing double quotes around a pattern stops folding to lower case. Should you need to include an actual double quote character in a pattern, write it as a pair of double quotes within a double-quote sequence; again this is in accord with the rules for SQL quoted identifiers. For example, \dt "FOO"BAR" will display the table named FOO"BAR (not foo"bar). Unlike the normal rules for SQL names, you can put double quotes around just part of a pattern, for instance \dt FOO"FOO"BAR will display the table named fooFOOBar.

Within a pattern, * matches any sequence of characters (including no characters) and ? matches any single character. (This notation is comparable to Unix shell file name patterns.) For example, \dt
int* displays all tables whose names begin with int. But within double quotes, * and ? lose these special meanings and are just matched literally.

A pattern that contains a dot (.) is interpreted as a schema name pattern followed by an object name pattern. For example, \dt foo*.bar* displays all tables whose table name includes bar that are in schemas whose schema name starts with foo. When no dot appears, then the pattern matches only objects that are visible in the current schema search path. Again, a dot within double quotes loses its special meaning and is matched literally.

Advanced users can use regular-expression notations such as character classes, for example [0-9] to match any digit. All regular expression special characters work as specified in Section 9.7.3, except for . which is taken as a separator as mentioned above, * which is translated to the regular-expression notation .*, ? which is translated to ., and $ which is matched literally. You can emulate these pattern characters at need by writing ? for ., (\$+|) for \$, or (r?) for $r?. $ is not needed as a regular-expression character since the pattern must match the whole name, unlike the usual interpretation of regular expressions (in other words, $ is automatically appended to your pattern). Write * at the beginning and/or end if you don’t wish the pattern to be anchored. Note that within double quotes, all regular expression special characters lose their special meanings and are matched literally. Also, the regular expression special characters are matched literally in operator name patterns (i.e., the argument of \do).

Whenever the pattern parameter is omitted completely, the \d commands display all objects that are visible in the current schema search path — this is equivalent to using the pattern *. To see all objects in the database, use the pattern *.*.

Advanced features

Variables

psql provides variable substitution features similar to common Unix command shells. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the psql meta-command \set:

testdb=> \set foo bar

sets the variable foo to the value bar. To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

testdb=> \echo :foo
bar

Note: The arguments of \set are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as \set :foo ’something’ and get “soft links” or “variable variables” of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is no way to do anything useful with these constructs. On the other hand, \set bar :foo is a perfectly valid way to copy a variable.

If you call \set without a second argument, the variable is set, with an empty string as value. To unset (or delete) a variable, use the command \unset.
psql’s internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of these variables are treated specially by psql. They indicate certain option settings that can be changed at run time by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes. A list of all specially treated variables follows.

**AUTO_COMMIT**

When on (the default), each SQL command is automatically committed upon successful completion. To postpone commit in this mode, you must enter a `BEGIN` or `START TRANSACTION` SQL command. When off or unset, SQL commands are not committed until you explicitly issue `COMMIT` or `END`. The autocommit-off mode works by issuing an implicit `BEGIN` for you, just before any command that is not already in a transaction block and is not itself a `BEGIN` or other transaction-control command, nor a command that cannot be executed inside a transaction block (such as `VACUUM`).

**Note:** In autocommit-off mode, you must explicitly abandon any failed transaction by entering `ABORT` or `ROLLBACK`. Also keep in mind that if you exit the session without committing, your work will be lost.

**Note:** The autocommit-on mode is PostgreSQL’s traditional behavior, but autocommit-off is closer to the SQL spec. If you prefer autocommit-off, you might wish to set it in the system-wide `psqlrc` file or your `~/.psqlrc` file.

**DBNAME**

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

**ECHO**

If set to `all`, all lines entered from the keyboard or from a script are written to the standard output before they are parsed or executed. To select this behavior on program start-up, use the switch `-a`. If set to `queries`, psql merely prints all queries as they are sent to the server. The switch for this is `-e`.

**ECHO_HIDDEN**

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the PostgreSQL internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch `-E`.) If you set the variable to the value `noexec`, the queries are just shown but are not actually sent to the server and executed.

**ENCODING**

The current client character set encoding.

**FETCH_COUNT**

If this variable is set to an integer value > 0, the results of `SELECT` queries are fetched and displayed in groups of that many rows, rather than the default behavior of collecting the entire
result set before display. Therefore only a limited amount of memory is used, regardless of the size of the result set. Settings of 100 to 1000 are commonly used when enabling this feature. Keep in mind that when using this feature, a query might fail after having already displayed some rows.

Tip: Although you can use any output format with this feature, the default aligned format tends to look bad because each group of FETCH_COUNT rows will be formatted separately, leading to varying column widths across the row groups. The other output formats work better.

HISTCONTROL

If this variable is set to ignorespace, lines which begin with a space are not entered into the history list. If set to a value of ignoredups, lines matching the previous history line are not entered. A value of ignoreboth combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

Note: This feature was shamelessly plagiarized from Bash.

HISTFILE

The file name that will be used to store the history list. The default value is ~/.psql_history. For example, putting:
\set HISTFILE ~/.psql_history- :DBNAME
in ~/.psqlrc will cause psql to maintain a separate history for each database.

Note: This feature was shamelessly plagiarized from Bash.

HISTORY

The number of commands to store in the command history. The default value is 500.

Note: This feature was shamelessly plagiarized from Bash.

HOST

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

IGNOREEOF

If unset, sending an EOF character (usually Control+D) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

Note: This feature was shamelessly plagiarized from Bash.
LASTOID

The value of the last affected OID, as returned from an INSERT or lo_insert command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

ON_ERROR_ROLLBACK

When on, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When interactive, such errors are only ignored in interactive sessions, and not when reading script files. When off (the default), a statement in a transaction block that generates an error aborts the entire transaction. The on_error_rollback-on mode works by issuing an implicit SAVEPOINT for you, just before each command that is in a transaction block, and rolls back to the savepoint on error.

ON_ERROR_STOP

By default, if non-interactive scripts encounter an error, such as a malformed SQL command or internal meta-command, processing continues. This has been the traditional behavior of psql but it is sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive psql session but rather using the -f option, psql will return error code 3, to distinguish this case from fatal error conditions (error code 1).

PORT

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

PROMPT1
PROMPT2
PROMPT3

These specify what the prompts psql issues should look like. See Prompting below.

QUIET

This variable is equivalent to the command line option -q. It is probably not too useful in interactive mode.

SINGLELINE

This variable is equivalent to the command line option -S.

SINGESTEP

This variable is equivalent to the command line option -s.

USER

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

VERBOSITY

This variable can be set to the values default, verbose, or terse to control the verbosity of error reports.
**SQL Interpolation**

An additional useful feature of `psql` variables is that you can substitute (“interpolate”) them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (``):

```
\set foo 'my_table'
```

```
\set content "'cat my_file.txt'"
```

would then query the table `my_table`. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statements to build a foreign key scenario. Another possible use of this mechanism is to copy the contents of a file into a table column. First load the file into a variable and then proceed as above:

```
\set content "'cat my_file.txt'"
```

```
INSERT INTO my_table VALUES (:content);
```

One problem with this approach is that `my_file.txt` might contain single quotes. These need to be escaped so that they don’t cause a syntax error when the second line is processed. This could be done with the program `sed`:

```
\set content "'sed -e "s/'/"/g" < my_file.txt'"
```

If you are using non-standard-conforming strings then you’ll also need to double backslashes. This is a bit tricky:

```
\set content "'sed -e "s/'/"/g" -e 's/\\/\\\\\\/g' < my_file.txt'"
```

Note the use of different shell quoting conventions so that neither the single quote marks nor the backslashes are special to the shell. Backslashes are still special to `sed`, however, so we need to double them. (Perhaps at one point you thought it was great that all Unix commands use the same escape character.)

Since colons can legally appear in SQL commands, the following rule applies: the character sequence `":"name"` is not changed unless “name” is the name of a variable that is currently set. In any case you can escape a colon with a backslash to protect it from substitution. (The colon syntax for variables is standard SQL for embedded query languages, such as ECPG. The colon syntax for array slices and type casts are PostgreSQL extensions, hence the conflict.)

**Prompting**

The prompts `psql` issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when `psql` requests a new command. Prompt 2 is issued when more input is expected during command input because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL `COPY` command and you are expected to type in the row values on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (%) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:
The full host name (with domain name) of the database server, or \[local\] if the connection is over a Unix domain socket, or \[local:/dir/name\], if the Unix domain socket is not at the compiled in default location.

The host name of the database server, truncated at the first dot, or \[local\] if the connection is over a Unix domain socket.

The port number at which the database server is listening.

The database session user name. (The expansion of this value might change during a database session as the result of the command \texttt{SET SESSION AUTHORIZATION}.)

The name of the current database.

Like %/, but the output is ~ (tilde) if the database is your default database.

If the session user is a database superuser, then a #, otherwise a >. (The expansion of this value might change during a database session as the result of the command \texttt{SET SESSION AUTHORIZATION}.)

In prompt 1 normally =, but ^ if in single-line mode, and ! if the session is disconnected from the database (which can happen if \texttt{\connect} fails). In prompt 2 the sequence is replaced by ~, *, a single quote, a double quote, or a dollar sign, depending on whether psql expects more input because the command wasn’t terminated yet, because you are inside a /* ... */ comment, or because you are inside a quoted or dollar-escaped string. In prompt 3 the sequence doesn’t produce anything.

Transaction status: an empty string when not in a transaction block, or * when in a transaction block, or ! when in a failed transaction block, or ? when the transaction state is indeterminate (for example, because there is no connection).

The character with the indicated octal code is substituted.

The value of the psql variable \texttt{name}. See the section \texttt{Variables} for details.

The output of \texttt{command}, similar to ordinary “back-tick” substitution.

Prompts can contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for the line editing features of Readline to work properly, these non-printing control characters must be...
designated as invisible by surrounding them with \% and \%. Multiple pairs of these can occur within the prompt. For example:

testdb=> \set PROMPT1 '%[\%033\[1;33;40m\]%n@%/R%\%033\[0m\]%# '
results in a boldfaced (1;) yellow-on-black (33;40) prompt on VT100-compatible, color-capable terminals.

To insert a percent sign into your prompt, write \%%. The default prompts are '%/%R%# ' for prompts 1 and 2, and '>> ' for prompt 3.

Note: This feature was shamelessly plagiarized from tcsh.

Command-Line Editing

psql supports the Readline library for convenient line editing and retrieval. The command history is automatically saved when psql exits and is reloaded when psql starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. If for some reason you do not like the tab completion, you can turn it off by putting this in a file named .inputrc in your home directory:

$if psql
set disable-completion on
$endif

(This is not a psql but a Readline feature. Read its documentation for further details.)

Environment

PAGER

If the query results do not fit on the screen, they are piped through this command. Typical values are more or less. The default is platform-dependent. The use of the pager can be disabled by using the \pset command.

PGDATABASE

Default connection database

PGHOST
PGPORT
PGUSER

Default connection parameters

PSQL_EDITOR
EDITOR
VISUAL

Editor used by the \e command. The variables are examined in the order listed; the first that is set is used.

SHELL

Command executed by the \! command.
TMPDIR

Directory for storing temporary files. The default is /tmp.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

Files

- Before starting up, psql attempts to read and execute commands from the system-wide psqlrc file and the user’s ~/.psqlrc file. (On Windows, the user’s startup file is named %APPDATA%\postgresql\psqlrc.conf.) See PREFIX/share/psqlrc.sample for information on setting up the system-wide file. It could be used to set up the client or the server to taste (using the \set and SET commands).
- Both the system-wide psqlrc file and the user’s ~/.psqlrc file can be made version-specific by appending a dash and the PostgreSQL release number, for example ~/.psqlrc-8.3.23. A matching version-specific file will be read in preference to a non-version-specific file.
- The command-line history is stored in the file ~/.psql_history, or %APPDATA%\postgresql\psql\psql_history on Windows.

Notes

- In an earlier life psql allowed the first argument of a single-letter backslash command to start directly after the command, without intervening whitespace. For compatibility this is still supported to some extent, but we are not going to explain the details here as this use is discouraged. If you get strange messages, keep this in mind. For example:

```
testdb=> \foo
Field separator is "oo".
which is perhaps not what one would expect.
```

- psql only works smoothly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up. Backslash commands are particularly likely to fail if the server is of a different version.

Notes for Windows users

psql is built as a “console application”. Since the Windows console windows use a different encoding than the rest of the system, you must take special care when using 8-bit characters within psql. If psql detects a problematic console code page, it will warn you at startup. To change the console code page, two things are necessary:

- Set the code page by entering cmd.exe /c chcp 1252. (1252 is a code page that is appropriate for German; replace it with your value.) If you are using Cygwin, you can put this command in /etc/profile.
- Set the console font to Lucida Console, because the raster font does not work with the ANSI code page.
Examples

The first example shows how to spread a command over several lines of input. Notice the changing prompt:

```
psql testdb=> CREATE TABLE my_table ( 
psql testdb=> first integer not null default 0, 
psql testdb=> second text) 
psql testdb=> ;
```

CREATE TABLE

Now look at the table definition again:

```
psql testdb=> \d my_table
```

Table "my_table"

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>integer</td>
<td>not null default 0</td>
</tr>
<tr>
<td>second</td>
<td>text</td>
<td></td>
</tr>
</tbody>
</table>

Now we change the prompt to something more interesting:

```
psql testdb=> \set PROMPT1 '%n@%m %~%R%# '
```

```
peter@localhost testdb=>
```

Let’s assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
```

```
+-------+--------+
| first | second |
+-------+--------+
| 1     | one    |
| 2     | two    |
| 3     | three  |
| 4     | four   |
+-------+--------+
(4 rows)
```

You can display tables in different ways by using the \pset command:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
```

```
+-------+--------+
| first | second |
+-------+--------+
| 1     | one    |
| 2     | two    |
| 3     | three  |
| 4     | four   |
+-------+--------+
(4 rows)
```

```
peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
```

```
1232
```
first second
----- ------
  1 one
  2 two
  3 three
  4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4

Alternatively, use the short commands:

peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
- [ RECORD 1 ] -
first | 1
second | one
- [ RECORD 2 ] -
first | 2
second | two
- [ RECORD 3 ] -
first | 3
second | three
- [ RECORD 4 ] -
first | 4
second | four
reindexdb

Name
reindexdb — reindex a PostgreSQL database

Synopsis
reindexdb [connection-option...] [--table | -t table] [--index | -i index] [dbname]
reindexdb [connection-option...] [--all | -a]
reindexdb [connection-option...] [--system | -s] [dbname]

Description
reindexdb is a utility for rebuilding indexes in a PostgreSQL database.
reindexdb is a wrapper around the SQL command REINDEX. There is no effective difference between
reindexing databases via this utility and via other methods for accessing the server.

Options
reindexdb accepts the following command-line arguments:

-a
--all
Reindex all databases.

[-d] dbname
[--dbname] dbname
Specifies the name of the database to be reindexed. If this is not specified and -a (or --all) is
not used, the database name is read from the environment variable PGDATABASE. If that is not
set, the user name specified for the connection is used.

-e
--echo
Echo the commands that reindexdb generates and sends to the server.

-i index
--index index
Recreate index only.

-q
--quiet
Do not display progress messages.

-s
--system
Reindex database’s system catalogs.
-t table
--table table

Reindex table only.

reindexdb also accepts the following command-line arguments for connection parameters:

-h host
--host host

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

-p port
--port port

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

-U username
--username username

User name to connect as.

-W
--password

Force reindexdb to prompt for a password before connecting to a database.

This option is never essential, since reindexdb will automatically prompt for a password if the server demands password authentication. However, reindexdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing -W to avoid the extra connection attempt.

Environment

PGDATABASE
PGHOST
PGPORT
PGUSER

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

Diagnostics

In case of difficulty, see REINDEX and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.
Notes

reindexdb might need to connect several times to the PostgreSQL server, asking for a password each time. It is convenient to have a ~/.pgpass file in such cases. See Section 30.13 for more information.

Examples

To reindex the database test:

$ reindexdb test

To reindex the table foo and the index bar in a database named abcd:

$ reindexdb --table foo --index bar abcd

See Also

REINDEX
vacuumdb

Name
vacuumdb — garbage-collect and analyze a PostgreSQL database

Synopsis
vacuumdb [connection-option...][-full | -f][-verbose | -v][-analyze | -z][-table | -t table [([column [...])]]] [dbname]
vacuumdb [connection-options...][-all | -a][-full | -f][-verbose | -v][-analyze | -z]

Description
vacuumdb is a utility for cleaning a PostgreSQL database. vacuumdb will also generate internal statistics used by the PostgreSQL query optimizer.

vacuumdb is a wrapper around the SQL command VACUUM. There is no effective difference between vacuuming databases via this utility and via other methods for accessing the server.

Options
vacuumdb accepts the following command-line arguments:

-a
--all
Vacuum all databases.

[-d] dbname
[--dbname] dbname
Specifies the name of the database to be cleaned or analyzed. If this is not specified and -a (or --all) is not used, the database name is read from the environment variable PGDATABASE. If that is not set, the user name specified for the connection is used.

-e
--echo
Echo the commands that vacuumdb generates and sends to the server.

-f
--full
Perform “full” vacuuming.

-q
--quiet
Do not display progress messages.
-t table [ (column [, ...]) ]
--table table [ (column [, ...]) ]

Clean or analyze table only. Column names can be specified only in conjunction with the
--analyze option.

Tip: If you specify columns, you probably have to escape the parentheses from the shell.
(See examples below.)

-v
--verbose

Print detailed information during processing.

-z
--analyze

Calculate statistics for use by the optimizer.

vacuumdb also accepts the following command-line arguments for connection parameters:

-h host
--host host

Specifies the host name of the machine on which the server is running. If the value begins with a
slash, it is used as the directory for the Unix domain socket.

-p port
--port port

Specifies the TCP port or local Unix domain socket file extension on which the server is listening
for connections.

-U username
--username username

User name to connect as.

-W
--password

Force vacuumdb to prompt for a password before connecting to a database.

This option is never essential, since vacuumdb will automatically prompt for a password if the
server demands password authentication. However, vacuumdb will waste a connection attempt
finding out that the server wants a password. In some cases it is worth typing -W to avoid the
extra connection attempt.
Environment

PGDATABASE
PGHOST
PGPORT
PGUSER

Default connection parameters
This utility, like most other PostgreSQL utilities, also uses the environment variables supported by
libpq (see Section 30.12).

Diagnostics
In case of difficulty, see VACUUM and psql for discussions of potential problems and error messages.
The database server must be running at the targeted host. Also, any default connection settings and
environment variables used by the libpq front-end library will apply.

Notes
vacuumdb might need to connect several times to the PostgreSQL server, asking for a password each
time. It is convenient to have a ~/.pgpass file in such cases. See Section 30.13 for more information.

Examples
To clean the database test:

$ vacuumdb test

To clean and analyze for the optimizer a database named bigdb:

$ vacuumdb --analyze bigdb

To clean a single table foo in a database named xyzzy, and analyze a single column bar of the table
for the optimizer:

$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy

See Also
VACUUM
III. PostgreSQL Server Applications

This part contains reference information for PostgreSQL server applications and support utilities. These commands can only be run usefully on the host where the database server resides. Other utility programs are listed in Reference II, *PostgreSQL Client Applications.*
initdb

Name
initdb — create a new PostgreSQL database cluster

Synopsis
initdb [option...] --pgdata | -D directory

Description
initdb creates a new PostgreSQL database cluster. A database cluster is a collection of databases
that are managed by a single server instance.

Creating a database cluster consists of creating the directories in which the database data will live,
generating the shared catalog tables (tables that belong to the whole cluster rather than to any par-
ticular database), and creating the template1 and postgres databases. When you later create a
new database, everything in the template1 database is copied. (Therefore, anything installed in
template1 is automatically copied into each database created later.) The postgres database is a
default database meant for use by users, utilities and third party applications.

Although initdb will attempt to create the specified data directory, it might not have permission if
the parent directory of the desired data directory is root-owned. To initialize in such a setup, create
an empty data directory as root, then use chown to assign ownership of that directory to the database
user account, then su to become the database user to run initdb.

initdb must be run as the user that will own the server process, because the server needs to have
access to the files and directories that initdb creates. Since the server cannot be run as root, you
must not run initdb as root either. (It will in fact refuse to do so.)

initdb initializes the database cluster’s default locale and character set encoding. The collation or-
der (LC_COLLATE) and character set classes (LC_CTYPE, e.g. upper, lower, digit) are fixed for all
databases and cannot be changed. Collation orders other than C or POSIX also have a performance
penalty. For these reasons it is important to choose the right locale when running initdb. The re-
maining locale categories can be changed later when the server is started. All server locale values
(lc_*) can be displayed via SHOW ALL. More details can be found in Section 22.1.

The character set encoding can be set separately for a database when it is created. initdb determines
the encoding for the template1 database, which will serve as the default for all other databases. To
alter the default encoding use the --encoding option. More details can be found in Section 22.2.
Options

-A authmethod
--auth=authmethod

This option specifies the authentication method for local users used in pg_hba.conf. Do not use trust unless you trust all local users on your system. Trust is the default for ease of installation.

-D directory
--pgdata=directory

This option specifies the directory where the database cluster should be stored. This is the only information required by initdb, but you can avoid writing it by setting the PGDATA environment variable, which can be convenient since the database server (postgres) can find the database directory later by the same variable.

-E encoding
--encoding=encoding

Selects the encoding of the template database. This will also be the default encoding of any database you create later, unless you override it there. The default is derived from the locale, or SQL_ASCII if that does not work. The character sets supported by the PostgreSQL server are described in Section 22.2.1.

--locale=locale

Sets the default locale for the database cluster. If this option is not specified, the locale is inherited from the environment that initdb runs in. Locale support is described in Section 22.1.

--lc-collate=locale
--lc-ctype=locale
--lc-messages=locale
--lc-monetary=locale
--lc-numeric=locale
--lc-time=locale

Like --locale, but only sets the locale in the specified category.

--no-locale

Equivalent to --locale=C.

-X directory
--xlogdir=directory

This option specifies the directory where the transaction log should be stored.

-U username
--username=username

Selects the user name of the database superuser. This defaults to the name of the effective user running initdb. It is really not important what the superuser’s name is, but one might choose to keep the customary name postgres, even if the operating system user’s name is different.

-W
--pwprompt

Makes initdb prompt for a password to give the database superuser. If you don’t plan on using password authentication, this is not important. Otherwise you won’t be able to use password authentication until you have a password set up.
initdb

--pwfile=filename

Makes initdb read the database superuser’s password from a file. The first line of the file is taken as the password.

--text-search-config=CFG

Sets the default text search configuration. See default_text_search_config for further information.

Other, less commonly used, parameters are also available:

-d
--debug

Print debugging output from the bootstrap backend and a few other messages of lesser interest for the general public. The bootstrap backend is the program initdb uses to create the catalog tables. This option generates a tremendous amount of extremely boring output.

-L directory

Specifies where initdb should find its input files to initialize the database cluster. This is normally not necessary. You will be told if you need to specify their location explicitly.

-n
--noclean

By default, when initdb determines that an error prevented it from completely creating the database cluster, it removes any files it might have created before discovering that it cannot finish the job. This option inhibits tidying-up and is thus useful for debugging.

Environment

PGDATA

Specifies the directory where the database cluster is to be stored; can be overridden using the -D option.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

See Also

postgres
ipcclean

Name

ipcclean — remove shared memory and semaphores from a failed PostgreSQL server

Synopsis

ipcclean

Description

ipcclean removes all shared memory segments and semaphore sets owned by the current user. It is intended to be used for cleaning up after a crashed PostgreSQL server (postgres). Note that immediately restarting the server will also clean up shared memory and semaphores, so this command is of little real utility.

Only the database administrator should execute this program as it can cause bizarre behavior (i.e., crashes) if run during multiuser execution. If this command is executed while a server is running, the shared memory and semaphores allocated by that server will be deleted, which would have rather severe consequences for that server.

Notes

This script is a hack, but in the many years since it was written, no one has come up with an equally effective and portable solution. Since postgres can now clean up by itself, it is unlikely that ipcclean will be improved upon in the future.

The script makes assumptions about the output format of the ipcs utility which might not be true across different operating systems. Therefore, it might not work on your particular OS. It’s wise to look at the script before trying it.
pg_controldata

Name

pg_controldata — display control information of a PostgreSQL database cluster

Synopsis

pg_controldata [datadir]

Description

pg_controldata prints information initialized during initdb, such as the catalog version and server locale. It also shows information about write-ahead logging and checkpoint processing. This information is cluster-wide, and not specific to any one database.

This utility can only be run by the user who initialized the cluster because it requires read access to the data directory. You can specify the data directory on the command line, or use the environment variable PGDATA.

Environment

PGDATA

    Default data directory location
pg_ctl

Name
pg_ctl — start, stop, or restart a PostgreSQL server

Synopsis

pg_ctl reload [-s] [-D datadir]
pg_ctl status [-D datadir]
pg_ctl kill signal_name process_id
pg_ctl unregister [-N servicename]

Description

pg_ctl is a utility for starting, stopping, or restarting the PostgreSQL backend server (postgres), or displaying the status of a running server. Although the server can be started manually, pg_ctl encapsulates tasks such as redirecting log output and properly detaching from the terminal and process group. It also provides convenient options for controlled shutdown.

In start mode, a new server is launched. The server is started in the background, and standard input is attached to /dev/null. The standard output and standard error are either appended to a log file (if the -l option is used), or redirected to pg_ctl’s standard output (not standard error). If no log file is chosen, the standard output of pg_ctl should be redirected to a file or piped to another process such as a log rotating program like rotatelogs; otherwise postgres will write its output to the controlling terminal (from the background) and will not leave the shell’s process group.

In stop mode, the server that is running in the specified data directory is shut down. Three different shutdown methods can be selected with the -m option: “Smart” mode waits for all the clients to disconnect. This is the default. “Fast” mode does not wait for clients to disconnect. All active transactions are rolled back and clients are forcibly disconnected, then the server is shut down. “Immediate” mode will abort all server processes without a clean shutdown. This will lead to a recovery run on restart.

restart mode effectively executes a stop followed by a start. This allows changing the postgres command-line options.

reload mode simply sends the postgres process a SIGHUP signal, causing it to reread its configuration files (postgresql.conf, pg_hba.conf, etc.). This allows changing of configuration-file options that do not require a complete restart to take effect.

status mode checks whether a server is running in the specified data directory. If it is, the PID and the command line options that were used to invoke it are displayed.
kill mode allows you to send a signal to a specified process. This is particularly valuable for Microsoft Windows which does not have a kill command. Use --help to see a list of supported signal names.

register mode allows you to register a system service on Microsoft Windows.

unregister mode allows you to unregister a system service on Microsoft Windows, previously registered with the register command.

**Options**

-c

Attempt to allow server crashes to produce core files, on platforms where this available, by lifting any soft resource limit placed on them. This is useful in debugging or diagnosing problems by allowing a stack trace to be obtained from a failed server process.

-D datadir

Specifies the file system location of the database files. If this is omitted, the environment variable PGDATA is used.

-l filename

Append the server log output to filename. If the file does not exist, it is created. The umask is set to 077, so access to the log file from other users is disallowed by default.

-m mode

Specifies the shutdown mode. *mode* can be smart, fast, or immediate, or the first letter of one of these three.

-o options

Specifies options to be passed directly to the postgres command.

The options are usually surrounded by single or double quotes to ensure that they are passed through as a group.

-p path

Specifies the location of the postgres executable. By default the postgres executable is taken from the same directory as pg_ctl, or failing that, the hard-wired installation directory. It is not necessary to use this option unless you are doing something unusual and get errors that the postgres executable was not found.

-s

Only print errors, no informational messages.

-t

The number of seconds to wait when waiting for start or shutdown to complete.

-w

Wait for the start or shutdown to complete. The default wait time is 60 seconds. This is the default option for shutdowns. A successful shutdown is indicated by removal of the PID file. For starting up, a successful \texttt{psql -l} indicates success. \texttt{pg_ctl} will attempt to use the proper port for psql. If the environment variable PGPORT exists, that is used. Otherwise, it will see if a port has been set in the postgresql.conf file. If neither of those is used, it will use the default
port that PostgreSQL was compiled with (5432 by default). When waiting, `pg_ctl` will return an accurate exit code based on the success of the startup or shutdown.

`-W`
Do not wait for start or shutdown to complete. This is the default for starts and restarts.

**Options for Windows**

`-N servicename`
Name of the system service to register. The name will be used as both the service name and the display name.

`-P password`
Password for the user to start the service.

`-U username`
User name for the user to start the service. For domain users, use the format `DOMAIN\username`.

**Environment**

`PGDATA`
Default data directory location.

`PGPORT`
Default port for psql (used by the `-w` option).

For additional server variables, see postgres. This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 30.12).

**Files**

`postmaster.pid`
The existence of this file in the data directory is used to help `pg_ctl` determine if the server is currently running or not.

`postmaster.opts.default`
If this file exists in the data directory, `pg_ctl` (in start mode) will pass the contents of the file as options to the `postgres` command, unless overridden by the `-o` option.

`postmaster.opts`
If this file exists in the data directory, `pg_ctl` (in restart mode) will pass the contents of the file as options to `postgres`, unless overridden by the `-o` option. The contents of this file are also displayed in status mode.

`postgresql.conf`
This file, located in the data directory, is parsed to find the proper port to use with psql when the `-w` is given in start mode.
Notes

Waiting for complete start is not a well-defined operation and might fail if access control is set up so that a local client cannot connect without manual interaction (e.g., password authentication). For additional connection variables, see Section 30.12, and for passwords, also see Section 30.13.

Examples

Starting the Server

To start up a server:

$ pg_ctl start

An example of starting the server, blocking until the server has come up is:

$ pg_ctl -w start

For a server using port 5433, and running without fsync, use:

$ pg_ctl -o "-F -p 5433" start

Stopping the Server

$ pg_ctl stop

stops the server. Using the -m switch allows one to control how the backend shuts down.

Restarting the Server

Restarting the server is almost equivalent to stopping the server and starting it again except that pg_ctl saves and reuses the command line options that were passed to the previously running instance. To restart the server in the simplest form, use:

$ pg_ctl restart

To restart server, waiting for it to shut down and to come up:

$ pg_ctl -w restart

To restart using port 5433 and disabling fsync after restarting:

$ pg_ctl -o "-F -p 5433" restart
Showing the Server Status

Here is a sample status output from pg_ctl:

```
$ pg_ctl status
pg_ctl: server is running (pid: 13718)
Command line was:
/usr/local/pgsql/bin/postgres '-D' '/usr/local/pgsql/data' '-p' '5433' '-B' '128'
```

This is the command line that would be invoked in restart mode.

See Also

postgres
pg_resetxlog

Name

pg_resetxlog — reset the write-ahead log and other control information of a PostgreSQL database cluster

Synopsis


Description

pg_resetxlog clears the write-ahead log (WAL) and optionally resets some other control information stored in the pg_control file. This function is sometimes needed if these files have become corrupted. It should be used only as a last resort, when the server will not start due to such corruption.

After running this command, it should be possible to start the server, but bear in mind that the database might contain inconsistent data due to partially-committed transactions. You should immediately dump your data, run initdb, and reload. After reload, check for inconsistencies and repair as needed.

This utility can only be run by the user who installed the server, because it requires read/write access to the data directory. For safety reasons, you must specify the data directory on the command line. pg_resetxlog does not use the environment variable PGDATA.

If pg_resetxlog complains that it cannot determine valid data for pg_control, you can force it to proceed anyway by specifying the -f (force) switch. In this case plausible values will be substituted for the missing data. Most of the fields can be expected to match, but manual assistance might be needed for the next OID, next transaction ID and epoch, next multitransaction ID and offset, WAL starting address, and database locale fields. The first six of these can be set using the switches discussed below. pg_resetxlog’s own environment is the source for its guess at the locale fields; take care that LANG and so forth match the environment that initdb was run in. If you are not able to determine correct values for all these fields, -f can still be used, but the recovered database must be treated with even more suspicion than usual: an immediate dump and reload is imperative. Do not execute any data-modifying operations in the database before you dump, as any such action is likely to make the corruption worse.

The -o, -x, -e, -m, -O, and -l switches allow the next OID, next transaction ID, next transaction ID’s epoch, next multitransaction ID, next multitransaction offset, and WAL starting address values to be set manually. These are only needed when pg_resetxlog is unable to determine appropriate values by reading pg_control. Safe values can be determined as follows:

- A safe value for the next transaction ID (-x) can be determined by looking for the numerically largest file name in the directory pg_clog under the data directory, adding one, and then multiplying by 1048576. Note that the file names are in hexadecimal. It is usually easiest to specify the switch value in hexadecimal too. For example, if 0011 is the largest entry in pg_clog, -x 0x1200000 will work (five trailing zeroes provide the proper multiplier).
- A safe value for the next multitransaction ID (-m) can be determined by looking for the numerically largest file name in the directory pg_multixact/offsets under the data directory, adding one,
and then multiplying by 65536. As above, the file names are in hexadecimal, so the easiest way to do this is to specify the switch value in hexadecimal and add four zeroes.

- A safe value for the next multitransaction offset (\(-O\)) can be determined by looking for the numerically largest file name in the directory `pg_multixact/members` under the data directory, adding one, and then multiplying by 65536. As above, the file names are in hexadecimal, so the easiest way to do this is to specify the switch value in hexadecimal and add four zeroes.

- The WAL starting address (\(-l\)) should be larger than any WAL segment file name currently existing in the directory `pg_xlog` under the data directory. These names are also in hexadecimal and have three parts. The first part is the "timeline ID" and should usually be kept the same. Do not choose a value larger than 255 (0xFF) for the third part; instead increment the second part and reset the third part to 0. For example, if 00000001000000320000004A is the largest entry in `pg_xlog`, \(-l\) 0x1,0x32,0x4B will work; but if the largest entry is 000000010000003A000000FF, choose \(-l\) 0x1,0x33,0x0 or more.

  **Note:** `pg_resetxlog` itself looks at the files in `pg_xlog` and chooses a default \(-l\) setting beyond the last existing file name. Therefore, manual adjustment of \(-l\) should only be needed if you are aware of WAL segment files that are not currently present in `pg_xlog`, such as entries in an offline archive; or if the contents of `pg_xlog` have been lost entirely.

- There is no comparably easy way to determine a next OID that's beyond the largest one in the database, but fortunately it is not critical to get the next-OID setting right.

- The transaction ID epoch is not actually stored anywhere in the database except in the field that is set by `pg_resetxlog`, so any value will work so far as the database itself is concerned. You might need to adjust this value to ensure that replication systems such as Slony-I work correctly — if so, an appropriate value should be obtainable from the state of the downstream replicated database.

The \(-n\) (no operation) switch instructs `pg_resetxlog` to print the values reconstructed from `pg_control` and then exit without modifying anything. This is mainly a debugging tool, but can be useful as a sanity check before allowing `pg_resetxlog` to proceed for real.

**Notes**

This command must not be used when the server is running. `pg_resetxlog` will refuse to start up if it finds a server lock file in the data directory. If the server crashed then a lock file might have been left behind; in that case you can remove the lock file to allow `pg_resetxlog` to run. But before you do so, make doubly certain that there is no server process still alive.
Name

postgres — PostgreSQL database server

Synopsis

postgres [option...]

Description

postgres is the PostgreSQL database server. In order for a client application to access a database it connects (over a network or locally) to a running postgres instance. The postgres instance then starts a separate server process to handle the connection.

One postgres instance always manages the data of exactly one database cluster. A database cluster is a collection of databases that is stored at a common file system location (the “data area”). More than one postgres instance can run on a system at one time, so long as they use different data areas and different communication ports (see below). When postgres starts it needs to know the location of the data area. The location must be specified by the -D option or the PGDATA environment variable; there is no default. Typically, -D or PGDATA points directly to the data area directory created by initdb. Other possible file layouts are discussed in Section 18.2.

By default postgres starts in the foreground and prints log messages to the standard error stream. In practical applications postgres should be started as a background process, perhaps at boot time.

The postgres command can also be called in single-user mode. The primary use for this mode is during bootstrapping by initdb. Sometimes it is used for debugging or disaster recovery (but note that running a single-user server is not truly suitable for debugging the server, since no realistic interprocess communication and locking will happen). When invoked in single-user mode from the shell, the user can enter queries and the results will be printed to the screen, but in a form that is more useful for developers than end users. In the single-user mode, the session user will be set to the user with ID 1, and implicit superuser powers are granted to this user. This user does not actually have to exist, so the single-user mode can be used to manually recover from certain kinds of accidental damage to the system catalogs.

Options

postgres accepts the following command-line arguments. For a detailed discussion of the options consult Chapter 18. You can save typing most of these options by setting up a configuration file. Some (safe) options can also be set from the connecting client in an application-dependent way to apply only for that session. For example, if the environment variable PGOPTIONS is set, then libpq-based clients will pass that string to the server, which will interpret it as postgres command-line options.
General Purpose

-A 0|1

Enables run-time assertion checks, which is a debugging aid to detect programming mistakes. This option is only available if assertions were enabled when PostgreSQL was compiled. If so, the default is on.

-B nbuffers

Sets the number of shared buffers for use by the server processes. The default value of this parameter is chosen automatically by initdb. Specifying this option is equivalent to setting the shared_buffers configuration parameter.

-c name=value

Sets a named run-time parameter. The configuration parameters supported by PostgreSQL are described in Chapter 18. Most of the other command line options are in fact short forms of such a parameter assignment. -c can appear multiple times to set multiple parameters.

-d debug-level

Sets the debug level. The higher this value is set, the more debugging output is written to the server log. Values are from 1 to 5. It is also possible to pass -d 0 for a specific session, which will prevent the server log level of the parent postgres process from being propagated to this session.

-D datadir

Specifies the file system location of the data directory or configuration file(s). See Section 18.2 for details.

-e

Sets the default date style to “European”, that is DMY ordering of input date fields. This also causes the day to be printed before the month in certain date output formats. See Section 8.5 for more information.

-F

Disables fsync calls for improved performance, at the risk of data corruption in the event of a system crash. Specifying this option is equivalent to disabling the fsync configuration parameter. Read the detailed documentation before using this!

-h hostname

Specifies the IP host name or address on which postgres is to listen for TCP/IP connections from client applications. The value can also be a comma-separated list of addresses, or * to specify listening on all available interfaces. An empty value specifies not listening on any IP addresses, in which case only Unix-domain sockets can be used to connect to the server. Defaults to listening only on localhost. Specifying this option is equivalent to setting the listen_addresses configuration parameter.

-i

Allows remote clients to connect via TCP/IP (Internet domain) connections. Without this option, only local connections are accepted. This option is equivalent to setting listen_addresses to * in postgresql.conf or via -h.

This option is deprecated since it does not allow access to the full functionality of listen_addresses. It's usually better to set listen_addresses directly.
-k directory

Specifies the directory of the Unix-domain socket on which postgres is to listen for connections from client applications. The default is normally /tmp, but can be changed at build time.

-l

Enables secure connections using SSL. PostgreSQL must have been compiled with support for SSL for this option to be available. For more information on using SSL, refer to Section 17.8.

-N max-connections

Sets the maximum number of client connections that this server will accept. The default value of this parameter is chosen automatically by initdb. Specifying this option is equivalent to setting the max_connections configuration parameter.

-o extra-options

The command-line-style options specified in extra-options are passed to all server processes started by this postgres process. If the option string contains any spaces, the entire string must be quoted.

The use of this option is obsolete; all command-line options for server processes can be specified directly on the postgres command line.

-p port

Specifies the TCP/IP port or local Unix domain socket file extension on which postgres is to listen for connections from client applications. Defaults to the value of the PGPORT environment variable, or if PGPORT is not set, then defaults to the value established during compilation (normally 5432). If you specify a port other than the default port, then all client applications must specify the same port using either command-line options or PGPORT.

-s

Print time information and other statistics at the end of each command. This is useful for benchmarking or for use in tuning the number of buffers.

-S work_mem

Specifies the amount of memory to be used by internal sorts and hashes before resorting to temporary disk files. See the description of the work_mem configuration parameter in Section 18.4.1.

--name=value

Sets a named run-time parameter; a shorter form of -c.

--describe-config

This option dumps out the server’s internal configuration variables, descriptions, and defaults in tab-delimited COPY format. It is designed primarily for use by administration tools.

Semi-internal Options

The options described here are used mainly for debugging purposes, and in some cases to assist with recovery of severely damaged databases. There should be no reason to use them in a production database setup. They are listed here only for use by PostgreSQL system developers. Furthermore, these options might change or be removed in a future release without notice.
Forbids the use of particular scan and join methods: \texttt{s} and \texttt{i} disable sequential and index scans respectively, while \texttt{n}, \texttt{m}, and \texttt{h} disable nested-loop, merge and hash joins respectively.

Neither sequential scans nor nested-loop joins can be disabled completely; the \texttt{-fs} and \texttt{-fn} options simply discourage the optimizer from using those plan types if it has any other alternative.

\texttt{-n}

This option is for debugging problems that cause a server process to die abnormally. The ordinary strategy in this situation is to notify all other server processes that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant server process could have corrupted some shared state before terminating. This option specifies that \texttt{postgres} will not reinitialize shared data structures. A knowledgeable system programmer can then use a debugger to examine shared memory and semaphore state.

\texttt{-O}

Allows the structure of system tables to be modified. This is used by \texttt{initdb}.

\texttt{-P}

Ignore system indexes when reading system tables (but still update the indexes when modifying the tables). This is useful when recovering from damaged system indexes.

\texttt{-t pa[rser] | pl[anner] | e[xecutor]}

Print timing statistics for each query relating to each of the major system modules. This option cannot be used together with the \texttt{-s} option.

\texttt{-T}

This option is for debugging problems that cause a server process to die abnormally. The ordinary strategy in this situation is to notify all other server processes that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant server process could have corrupted some shared state before terminating. This option specifies that \texttt{postgres} will stop all other server processes by sending the signal \texttt{SIGSTOP}, but will not cause them to terminate. This permits system programmers to collect core dumps from all server processes by hand.

\texttt{-v protocol}

Specifies the version number of the frontend/backend protocol to be used for a particular session. This option is for internal use only.

\texttt{-W seconds}

A delay of this many seconds occurs when a new server process is started, after it conducts the authentication procedure. This is intended to give an opportunity to attach to the server process with a debugger.

\textbf{Options for single-user mode}

The following options only apply to the single-user mode.

\texttt{--single}

Selects the single-user mode. This must be the first argument on the command line.
database

Specifies the name of the database to be accessed. This must be the last argument on the command line. If it is omitted it defaults to the user name.

-E

Echo all commands.

-j

Disables use of newline as a statement delimiter.

-r filename

Send all server log output to filename. In normal multiuser mode, this option is ignored, and stderr is used by all processes.

Environment

PGCLIENTENCODING

Default character encoding used by clients. (The clients can override this individually.) This value can also be set in the configuration file.

PGDATA

Default data directory location

PGDATESTYLE

Default value of the DateStyle run-time parameter. (The use of this environment variable is deprecated.)

PGPORT

Default port (preferably set in the configuration file)

TZ

Server time zone

Diagnostics

A failure message mentioning semget or shmget probably indicates you need to configure your kernel to provide adequate shared memory and semaphores. For more discussion see Section 17.4. You might be able to postpone reconfiguring your kernel by decreasing shared_buffers to reduce the shared memory consumption of PostgreSQL, and/or by reducing max_connections to reduce the semaphore consumption.

A failure message suggesting that another server is already running should be checked carefully, for example by using the command

$ ps ax | grep postgres

or

$ ps -ef | grep postgres
depending on your system. If you are certain that no conflicting server is running, you can remove the
lock file mentioned in the message and try again.

A failure message indicating inability to bind to a port might indicate that that port is already in
use by some non-PostgreSQL process. You might also get this error if you terminate `postgres` and
immediately restart it using the same port; in this case, you must simply wait a few seconds until the
operating system closes the port before trying again. Finally, you might get this error if you specify a
port number that your operating system considers to be reserved. For example, many versions of Unix
consider port numbers under 1024 to be “trusted” and only permit the Unix superuser to access them.

**Notes**

The utility command `pg_ctl` can be used to start and shut down the `postgres` server safely and
comfortably.

If at all possible, do not use SIGKILL to kill the main `postgres` server. Doing so will prevent
`postgres` from freeing the system resources (e.g., shared memory and semaphores) that it holds
before terminating. This might cause problems for starting a fresh `postgres` run.

To terminate the `postgres` server normally, the signals SIGTERM, SIGINT, or SIGQUIT can be used.
The first will wait for all clients to terminate before quitting, the second will forcefully disconnect
all clients, and the third will quit immediately without proper shutdown, resulting in a recovery run
during restart.

The SIGHUP signal will reload the server configuration files. It is also possible to send SIGHUP to an
individual server process, but that is usually not sensible.

To cancel a running query, send the SIGINT signal to the process running that command.

The `postgres` server uses SIGTERM to tell subordinate server processes to quit normally and
SIGQUIT to terminate without the normal cleanup. These signals should not be used by users. It is
also unwise to send SIGKILL to a server process — the main `postgres` process will interpret this as
a crash and will force all the sibling processes to quit as part of its standard crash-recovery procedure.

**Bugs**

The -- options will not work on FreeBSD or OpenBSD. Use -c instead. This is a bug in the affected
operating systems; a future release of PostgreSQL will provide a workaround if this is not fixed.

**Usage**

To start a single-user mode server, use a command like

```bash
postgres --single -D /usr/local/pgsql/data other-options my_database
```

Provide the correct path to the database directory with -D, or make sure that the environment variable
PGDATA is set. Also specify the name of the particular database you want to work in.

Normally, the single-user mode server treats newline as the command entry terminator; there is no
intelligence about semicolons, as there is in psql. To continue a command across multiple lines, you
must type backslash just before each newline except the last one.
But if you use the \-j command line switch, then newline does not terminate command entry. In this case, the server will read the standard input until the end-of-file (EOF) marker, then process the input as a single command string. Backslash-newline is not treated specially in this case.

To quit the session, type EOF (\Control+D, usually). If you’ve used \-j, two consecutive EOFs are needed to exit.

Note that the single-user mode server does not provide sophisticated line-editing features (no command history, for example).

**Examples**

To start *postgres* in the background using default values, type:

```
$ nohup postgres >logfile 2>&1 < /dev/null &
```

To start *postgres* with a specific port:

```
$ postgres -p 1234
```

This command will start up *postgres* communicating through the port 1234. In order to connect to this server using *psql*, you would need to run it as

```
$ psql -p 1234
```

or set the environment variable *PGPORT*:

```
$ export PGPORT=1234
$ psql
```

Named run-time parameters can be set in either of these styles:

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Either form overrides whatever setting might exist for *work_mem* in *postgresql.conf*. Notice that underscores in parameter names can be written as either underscore or dash on the command line. Except for short-term experiments, it’s probably better practice to edit the setting in *postgresql.conf* than to rely on a command-line switch to set a parameter.

**See Also**

initdb, pg_ctl
postmaster

Name
postmaster — PostgreSQL database server

Synopsis
postmaster [option...]

Description
postmaster is a deprecated alias of postgres.

See Also
postgres
VII. Internals

This part contains assorted information that might be of use to PostgreSQL developers.
Chapter 43. Overview of PostgreSQL Internals

Author: This chapter originated as part of Enhancement of the ANSI SQL Implementation of PostgreSQL, Stefan Simkovic’s Master’s Thesis prepared at Vienna University of Technology under the direction of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr.

This chapter gives an overview of the internal structure of the backend of PostgreSQL. After having read the following sections you should have an idea of how a query is processed. This chapter does not aim to provide a detailed description of the internal operation of PostgreSQL, as such a document would be very extensive. Rather, this chapter is intended to help the reader understand the general sequence of operations that occur within the backend from the point at which a query is received, to the point at which the results are returned to the client.

43.1. The Path of a Query

Here we give a short overview of the stages a query has to pass in order to obtain a result.

1. A connection from an application program to the PostgreSQL server has to be established. The application program transmits a query to the server and waits to receive the results sent back by the server.

2. The parser stage checks the query transmitted by the application program for correct syntax and creates a query tree.

3. The rewrite system takes the query tree created by the parser stage and looks for any rules (stored in the system catalogs) to apply to the query tree. It performs the transformations given in the rule bodies.

   One application of the rewrite system is in the realization of views. Whenever a query against a view (i.e. a virtual table) is made, the rewrite system rewrites the user’s query to a query that accesses the base tables given in the view definition instead.

4. The planner/optimizer takes the (rewritten) query tree and creates a query plan that will be the input to the executor.

   It does so by first creating all possible paths leading to the same result. For example if there is an index on a relation to be scanned, there are two paths for the scan. One possibility is a simple sequential scan and the other possibility is to use the index. Next the cost for the execution of each path is estimated and the cheapest path is chosen. The cheapest path is expanded into a complete plan that the executor can use.

5. The executor recursively steps through the plan tree and retrieves rows in the way represented by the plan. The executor makes use of the storage system while scanning relations, performs sorts and joins, evaluates qualifications and finally hands back the rows derived.

In the following sections we will cover each of the above listed items in more detail to give a better understanding of PostgreSQL’s internal control and data structures.
43.2. How Connections are Established

PostgreSQL is implemented using a simple “process per user” client/server model. In this model there is one client process connected to exactly one server process. As we do not know ahead of time how many connections will be made, we have to use a master process that spawns a new server process every time a connection is requested. This master process is called postgres and listens at a specified TCP/IP port for incoming connections. Whenever a request for a connection is detected the postgres process spawns a new server process. The server tasks communicate with each other using semaphores and shared memory to ensure data integrity throughout concurrent data access.

The client process can be any program that understands the PostgreSQL protocol described in Chapter 45. Many clients are based on the C-language library libpq, but several independent implementations of the protocol exist, such as the Java JDBC driver.

Once a connection is established the client process can send a query to the backend (server). The query is transmitted using plain text, i.e. there is no parsing done in the frontend (client). The server parses the query, creates an execution plan, executes the plan and returns the retrieved rows to the client by transmitting them over the established connection.

43.3. The Parser Stage

The parser stage consists of two parts:

- The parser defined in gram.y and scan.l is built using the Unix tools yacc and lex.
- The transformation process does modifications and augmentations to the data structures returned by the parser.

43.3.1. Parser

The parser has to check the query string (which arrives as plain ASCII text) for valid syntax. If the syntax is correct a parse tree is built up and handed back; otherwise an error is returned. The parser and lexer are implemented using the well-known Unix tools yacc and lex.

The lexer is defined in the file scan.l and is responsible for recognizing identifiers, the SQL key words etc. For every key word or identifier that is found, a token is generated and handed to the parser.

The parser is defined in the file gram.y and consists of a set of grammar rules and actions that are executed whenever a rule is fired. The code of the actions (which is actually C code) is used to build up the parse tree.

The file scan.l is transformed to the C source file scan.c using the program lex and gram.y is transformed to gram.c using yacc. After these transformations have taken place a normal C compiler can be used to create the parser. Never make any changes to the generated C files as they will be overwritten the next time lex or yacc is called.

Note: The mentioned transformations and compilations are normally done automatically using the makefiles shipped with the PostgreSQL source distribution.
A detailed description of yacc or the grammar rules given in `gram.y` would be beyond the scope of this paper. There are many books and documents dealing with lex and yacc. You should be familiar with yacc before you start to study the grammar given in `gram.y` otherwise you won’t understand what happens there.

### 43.3.2. Transformation Process

The parser stage creates a parse tree using only fixed rules about the syntactic structure of SQL. It does not make any lookups in the system catalogs, so there is no possibility to understand the detailed semantics of the requested operations. After the parser completes, the transformation process takes the tree handed back by the parser as input and does the semantic interpretation needed to understand which tables, functions, and operators are referenced by the query. The data structure that is built to represent this information is called the *query tree*.

The reason for separating raw parsing from semantic analysis is that system catalog lookups can only be done within a transaction, and we do not wish to start a transaction immediately upon receiving a query string. The raw parsing stage is sufficient to identify the transaction control commands (BEGIN, ROLLBACK, etc), and these can then be correctly executed without any further analysis. Once we know that we are dealing with an actual query (such as SELECT or UPDATE), it is okay to start a transaction if we’re not already in one. Only then can the transformation process be invoked.

The query tree created by the transformation process is structurally similar to the raw parse tree in most places, but it has many differences in detail. For example, a `FuncCall` node in the parse tree represents something that looks syntactically like a function call. This might be transformed to either a `FuncExpr` or `Aggref` node depending on whether the referenced name turns out to be an ordinary function or an aggregate function. Also, information about the actual data types of columns and expression results is added to the query tree.

### 43.4. The PostgreSQL Rule System

PostgreSQL supports a powerful *rule system* for the specification of *views* and ambiguous *view updates*. Originally the PostgreSQL rule system consisted of two implementations:

- The first one worked using *row level* processing and was implemented deep in the *executor*. The rule system was called whenever an individual row had been accessed. This implementation was removed in 1995 when the last official release of the Berkeley Postgres project was transformed into Postgres95.
- The second implementation of the rule system is a technique called *query rewriting*. The *rewrite system* is a module that exists between the *parser stage* and the *planner/optimizer*. This technique is still implemented.

The query rewriter is discussed in some detail in Chapter 36, so there is no need to cover it here. We will only point out that both the input and the output of the rewriter are query trees, that is, there is no change in the representation or level of semantic detail in the trees. Rewriting can be thought of as a form of macro expansion.
43.5. Planner/Optimizer

The task of the planner/optimizer is to create an optimal execution plan. A given SQL query (and hence, a query tree) can be actually executed in a wide variety of different ways, each of which will produce the same set of results. If it is computationally feasible, the query optimizer will examine each of these possible execution plans, ultimately selecting the execution plan that is expected to run the fastest.

Note: In some situations, examining each possible way in which a query can be executed would take an excessive amount of time and memory space. In particular, this occurs when executing queries involving large numbers of join operations. In order to determine a reasonable (not necessarily optimal) query plan in a reasonable amount of time, PostgreSQL uses a Genetic Query Optimizer when the number of joins exceeds a threshold (see geqo_threshold).

The planner’s search procedure actually works with data structures called paths, which are simply cut-down representations of plans containing only as much information as the planner needs to make its decisions. After the cheapest path is determined, a full-fledged plan tree is built to pass to the executor. This represents the desired execution plan in sufficient detail for the executor to run it. In the rest of this section we’ll ignore the distinction between paths and plans.

43.5.1. Generating Possible Plans

The planner/optimizer starts by generating plans for scanning each individual relation (table) used in the query. The possible plans are determined by the available indexes on each relation. There is always the possibility of performing a sequential scan on a relation, so a sequential scan plan is always created. Assume an index is defined on a relation (for example a B-tree index) and a query contains the restriction relation.attribute OPR constant. If relation.attribute happens to match the key of the B-tree index and OPR is one of the operators listed in the index’s operator class, another plan is created using the B-tree index to scan the relation. If there are further indexes present and the restrictions in the query happen to match a key of an index, further plans will be considered. Index scan plans are also generated for indexes that have a sort ordering that can match the query’s ORDER BY clause (if any), or a sort ordering that might be useful for merge joining (see below).

If the query requires joining two or more relations, plans for joining relations are considered after all feasible plans have been found for scanning single relations. The three available join strategies are:

- **nested loop join**: The right relation is scanned once for every row found in the left relation. This strategy is easy to implement but can be very time consuming. (However, if the right relation can be scanned with an index scan, this can be a good strategy. It is possible to use values from the current row of the left relation as keys for the index scan of the right.)

- **merge join**: Each relation is sorted on the join attributes before the join starts. Then the two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is more attractive because each relation has to be scanned only once. The required sorting might be achieved either by an explicit sort step, or by scanning the relation in the proper order using an index on the join key.

- **hash join**: the right relation is first scanned and loaded into a hash table, using its join attributes as hash keys. Next the left relation is scanned and the appropriate values of every row found are used as hash keys to locate the matching rows in the table.
When the query involves more than two relations, the final result must be built up by a tree of join steps, each with two inputs. The planner examines different possible join sequences to find the cheapest one.

If the query uses fewer than geqo_threshold relations, a near-exhaustive search is conducted to find the best join sequence. The planner preferentially considers joins between any two relations for which there exist a corresponding join clause in the WHERE qualification (i.e. for which a restriction like WHERE rel1.attr1=rel2.attr2 exists). Join pairs with no join clause are considered only when there is no other choice, that is, a particular relation has no available join clauses to any other relation. All possible plans are generated for every join pair considered by the planner, and the one that is (estimated to be) the cheapest is chosen.

When geqo_threshold is exceeded, the join sequences considered are determined by heuristics, as described in Chapter 49. Otherwise the process is the same.

The finished plan tree consists of sequential or index scans of the base relations, plus nested-loop, merge, or hash join nodes as needed, plus any auxiliary steps needed, such as sort nodes or aggregate-function calculation nodes. Most of these plan node types have the additional ability to do selection (discarding rows that do not meet a specified boolean condition) and projection (computation of a derived column set based on given column values, that is, evaluation of scalar expressions where needed). One of the responsibilities of the planner is to attach selection conditions from the WHERE clause and computation of required output expressions to the most appropriate nodes of the plan tree.

43.6. Executor

The executor takes the plan handed back by the planner/optimizer and recursively processes it to extract the required set of rows. This is essentially a demand-pull pipeline mechanism. Each time a plan node is called, it must deliver one more row, or report that it is done delivering rows.

To provide a concrete example, assume that the top node is a MergeJoin node. Before any merge can be done two rows have to be fetched (one from each subplan). So the executor recursively calls itself to process the subplans (it starts with the subplan attached to lefttree). The new top node (the top node of the left subplan) is, let’s say, a Sort node and again recursion is needed to obtain an input row. The child node of the Sort might be a SeqScan node, representing actual reading of a table. Execution of this node causes the executor to fetch a row from the table and return it up to the calling node. The Sort node will repeatedly call its child to obtain all the rows to be sorted. When the input is exhausted (as indicated by the child node returning a NULL instead of a row), the Sort code performs the sort, and finally is able to return its first output row, namely the first one in sorted order. It keeps the remaining rows stored so that it can deliver them in sorted order in response to later demands.

The MergeJoin node similarly demands the first row from its right subplan. Then it compares the two rows to see if they can be joined; if so, it returns a join row to its caller. On the next call, or immediately if it cannot join the current pair of inputs, it advances to the next row of one table or the other (depending on how the comparison came out), and again checks for a match. Eventually, one subplan or the other is exhausted, and the MergeJoin node returns NULL to indicate that no more join rows can be formed.

Complex queries can involve many levels of plan nodes, but the general approach is the same: each node computes and returns its next output row each time it is called. Each node is also responsible for applying any selection or projection expressions that were assigned to it by the planner.

The executor mechanism is used to evaluate all four basic SQL query types: SELECT, INSERT, UPDATE, and DELETE. For SELECT, the top-level executor code only needs to send each row re-
turned by the query plan tree off to the client. For INSERT, each returned row is inserted into the target table specified for the INSERT. (A simple INSERT ... VALUES command creates a trivial plan tree consisting of a single Result node, which computes just one result row. But INSERT ... SELECT can demand the full power of the executor mechanism.) For UPDATE, the planner arranges that each computed row includes all the updated column values, plus the TID (tuple ID, or row ID) of the original target row; the executor top level uses this information to create a new updated row and mark the old row deleted. For DELETE, the only column that is actually returned by the plan is the TID, and the executor top level simply uses the TID to visit each target row and mark it deleted.
Chapter 44. System Catalogs

The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL’s system catalogs are regular tables. You can drop and recreate the tables, add columns, insert and update values, and severely mess up your system that way. Normally, one should not change the system catalogs by hand, there are always SQL commands to do that. (For example, CREATE DATABASE inserts a row into the pg_database catalog — and actually creates the database on disk.) There are some exceptions for particularly esoteric operations, such as adding index access methods.

44.1. Overview

Table 44-1 lists the system catalogs. More detailed documentation of each catalog follows below.

Most system catalogs are copied from the template database during database creation and are there-after database-specific. A few catalogs are physically shared across all databases in a cluster; these are noted in the descriptions of the individual catalogs.

Table 44-1. System Catalogs

<table>
<thead>
<tr>
<th>Catalog Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_aggregate</td>
<td>aggregate functions</td>
</tr>
<tr>
<td>pg_am</td>
<td>index access methods</td>
</tr>
<tr>
<td>pg_amop</td>
<td>access method operators</td>
</tr>
<tr>
<td>pg_amproc</td>
<td>access method support procedures</td>
</tr>
<tr>
<td>pg_attrdef</td>
<td>column default values</td>
</tr>
<tr>
<td>pg_attribute</td>
<td>table columns (&quot;attributes&quot;)</td>
</tr>
<tr>
<td>pg_authid</td>
<td>authorization identifiers (roles)</td>
</tr>
<tr>
<td>pg_auth_members</td>
<td>authorization identifier membership relationships</td>
</tr>
<tr>
<td>pg_autovacuum</td>
<td>per-relation autovacuum configuration parameters</td>
</tr>
<tr>
<td>pg_cast</td>
<td>casts (data type conversions)</td>
</tr>
<tr>
<td>pg_class</td>
<td>tables, indexes, sequences, views (&quot;relations&quot;)</td>
</tr>
<tr>
<td>pg_constraint</td>
<td>check constraints, unique constraints, primary key constraints, foreign key constraints</td>
</tr>
<tr>
<td>pg_conversion</td>
<td>encoding conversion information</td>
</tr>
<tr>
<td>pg_database</td>
<td>databases within this database cluster</td>
</tr>
<tr>
<td>pg_depend</td>
<td>dependencies between database objects</td>
</tr>
<tr>
<td>pg_description</td>
<td>descriptions or comments on database objects</td>
</tr>
<tr>
<td>pg_enum</td>
<td>enum label and value definitions</td>
</tr>
<tr>
<td>pg_index</td>
<td>additional index information</td>
</tr>
<tr>
<td>pg_inherits</td>
<td>table inheritance hierarchy</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Catalog Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_language</td>
<td>languages for writing functions</td>
</tr>
<tr>
<td>pg_largeobject</td>
<td>large objects</td>
</tr>
<tr>
<td>pg_listener</td>
<td>asynchronous notification support</td>
</tr>
<tr>
<td>pg_namespace</td>
<td>schemas</td>
</tr>
<tr>
<td>pg_opclass</td>
<td>access method operator classes</td>
</tr>
<tr>
<td>pg_operator</td>
<td>operators</td>
</tr>
<tr>
<td>pg_opfamily</td>
<td>access method operator families</td>
</tr>
<tr>
<td>pg_pltemplate</td>
<td>template data for procedural languages</td>
</tr>
<tr>
<td>pg_proc</td>
<td>functions and procedures</td>
</tr>
<tr>
<td>pg_rewrite</td>
<td>query rewrite rules</td>
</tr>
<tr>
<td>pg_shdepend</td>
<td>dependencies on shared objects</td>
</tr>
<tr>
<td>pg_shdescription</td>
<td>comments on shared objects</td>
</tr>
<tr>
<td>pg_statistic</td>
<td>planner statistics</td>
</tr>
<tr>
<td>pg_tablespace</td>
<td>tables spaces within this database cluster</td>
</tr>
<tr>
<td>pg_trigger</td>
<td>triggers</td>
</tr>
<tr>
<td>pg_ts_config</td>
<td>text search configurations</td>
</tr>
<tr>
<td>pg_ts_config_map</td>
<td>text search configurations’ token mappings</td>
</tr>
<tr>
<td>pg_ts_dict</td>
<td>text search dictionaries</td>
</tr>
<tr>
<td>pg_ts_parser</td>
<td>text search parsers</td>
</tr>
<tr>
<td>pg_ts_template</td>
<td>text search templates</td>
</tr>
<tr>
<td>pg_type</td>
<td>data types</td>
</tr>
</tbody>
</table>

44.2. pg_aggregate

The catalog pg_aggregate stores information about aggregate functions. An aggregate function is a function that operates on a set of values (typically one column from each row that matches a query condition) and returns a single value computed from all these values. Typical aggregate functions are sum, count, and max. Each entry in pg_aggregate is an extension of an entry in pg_proc. The pg_proc entry carries the aggregate’s name, input and output data types, and other information that is similar to ordinary functions.

Table 44-2. pg_aggregate Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggfnoid</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>pg_proc OID of the aggregate function</td>
</tr>
<tr>
<td>aggtransfn</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Transition function</td>
</tr>
<tr>
<td>aggfinalfn</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Final function (zero if none)</td>
</tr>
<tr>
<td>aggsortop</td>
<td>oid</td>
<td>pg_operator.oid</td>
<td>Associated sort operator (zero if none)</td>
</tr>
</tbody>
</table>
New aggregate functions are registered with the `CREATE AGGREGATE` command. See Section 34.10 for more information about writing aggregate functions and the meaning of the transition functions, etc.

### 44.3. `pg_am`

The catalog `pg_am` stores information about index access methods. There is one row for each index access method supported by the system. The contents of this catalog are discussed in detail in Chapter 50.

#### Table 44-3. `pg_am` Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amname</td>
<td>name</td>
<td></td>
<td>Name of the access method</td>
</tr>
<tr>
<td>amstrategies</td>
<td>int2</td>
<td></td>
<td>Number of operator strategies for this access method, or zero if access method does not have a fixed set of operator strategies</td>
</tr>
<tr>
<td>amsupport</td>
<td>int2</td>
<td></td>
<td>Number of support routines for this access method</td>
</tr>
<tr>
<td>amcanorder</td>
<td>bool</td>
<td></td>
<td>Does the access method support ordered scans?</td>
</tr>
<tr>
<td>amcanunique</td>
<td>bool</td>
<td></td>
<td>Does the access method support unique indexes?</td>
</tr>
<tr>
<td>amcanmulticol</td>
<td>bool</td>
<td></td>
<td>Does the access method support multicolumn indexes?</td>
</tr>
</tbody>
</table>
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amoptionalkey</td>
<td>bool</td>
<td></td>
<td>Does the access method support a scan without any constraint for the first index column?</td>
</tr>
<tr>
<td>amindexnulls</td>
<td>bool</td>
<td></td>
<td>Does the access method support null index entries?</td>
</tr>
<tr>
<td>amsearchnulls</td>
<td>bool</td>
<td></td>
<td>Does the access method support IS NULL searches?</td>
</tr>
<tr>
<td>amstorage</td>
<td>bool</td>
<td></td>
<td>Can index storage data type differ from column data type?</td>
</tr>
<tr>
<td>amclusterable</td>
<td>bool</td>
<td></td>
<td>Can an index of this type be clustered on?</td>
</tr>
<tr>
<td>aminsert</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“Insert this tuple” function</td>
</tr>
<tr>
<td>ambeginscan</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“Start new scan” function</td>
</tr>
<tr>
<td>amgettuplet</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“Next valid tuple” function</td>
</tr>
<tr>
<td>amgetmulti</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“Fetch multiple tuples” function</td>
</tr>
<tr>
<td>amrescan</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“Restart this scan” function</td>
</tr>
<tr>
<td>amendscan</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“End this scan” function</td>
</tr>
<tr>
<td>ammarkpos</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“Mark current scan position” function</td>
</tr>
<tr>
<td>amrestrpos</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“Restore marked scan position” function</td>
</tr>
<tr>
<td>ambuild</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>“Build new index” function</td>
</tr>
<tr>
<td>ambulkdelete</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Bulk-delete function</td>
</tr>
<tr>
<td>amvacuumcleanup</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Post-VACUUM cleanup function</td>
</tr>
<tr>
<td>amcostestimate</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Function to estimate cost of an index scan</td>
</tr>
<tr>
<td>amoptions</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Function to parse and validate reloptions for an index</td>
</tr>
</tbody>
</table>
44.4. **pg_amop**

The catalog `pg_amop` stores information about operators associated with access method operator families. There is one row for each operator that is a member of an operator family. An operator can appear in more than one family, but cannot appear in more than one position within a family.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amopfamily</td>
<td>oid</td>
<td>pg_opfamily.oid</td>
<td>The operator family this entry is for</td>
</tr>
<tr>
<td>amoplefttype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Left-hand input data type of operator</td>
</tr>
<tr>
<td>amoprighttype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Right-hand input data type of operator</td>
</tr>
<tr>
<td>amopstrategy</td>
<td>int2</td>
<td></td>
<td>Operator strategy number</td>
</tr>
<tr>
<td>amopreqcheck</td>
<td>bool</td>
<td></td>
<td>Index hit must be rechecked</td>
</tr>
<tr>
<td>amopopr</td>
<td>oid</td>
<td>pg_operator.oid</td>
<td>OID of the operator</td>
</tr>
<tr>
<td>amopmethod</td>
<td>oid</td>
<td>pg_am.oid</td>
<td>Index access method operator family is for</td>
</tr>
</tbody>
</table>

An entry’s `amopmethod` must match the `opfmethod` of its containing operator family (including `amopmethod` here is an intentional denormalization of the catalog structure for performance reasons). Also, `amoplefttype` and `amoprighttype` must match the `oprleft` and `oprright` fields of the referenced `pg_operator` entry.

44.5. **pg_amproc**

The catalog `pg_amproc` stores information about support procedures associated with access method operator families. There is one row for each support procedure belonging to an operator family.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amprocfamily</td>
<td>oid</td>
<td>pg_opfamily.oid</td>
<td>The operator family this entry is for</td>
</tr>
<tr>
<td>amproclefttype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Left-hand input data type of associated operator</td>
</tr>
<tr>
<td>amprocrighttype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Right-hand input data type of associated operator</td>
</tr>
<tr>
<td>amprocnum</td>
<td>int2</td>
<td></td>
<td>Support procedure number</td>
</tr>
<tr>
<td>amproc</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>OID of the procedure</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

The usual interpretation of the amproclefttype and amprocrighttype fields is that they identify the left and right input types of the operator(s) that a particular support procedure supports. For some access methods these match the input data type(s) of the support procedure itself, for others not. There is a notion of “default” support procedures for an index, which are those with amproclefttype and amprocrighttype both equal to the index opclass’s opcintype.

44.6. pg_attrdef

The catalog pg_attrdef stores column default values. The main information about columns is stored in pg_attribute (see below). Only columns that explicitly specify a default value (when the table is created or the column is added) will have an entry here.

Table 44-6. pg_attrdef Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The table this column belongs to</td>
</tr>
<tr>
<td>adnum</td>
<td>int2</td>
<td>pg_attribute.attnum</td>
<td>The number of the column</td>
</tr>
<tr>
<td>adbin</td>
<td>text</td>
<td></td>
<td>The internal representation of the column default value</td>
</tr>
<tr>
<td>adsrc</td>
<td>text</td>
<td></td>
<td>A human-readable representation of the default value</td>
</tr>
</tbody>
</table>

The adsrc field is historical, and is best not used, because it does not track outside changes that might affect the representation of the default value. Reverse-compiling the adbin field (with pg_get_expr for example) is a better way to display the default value.

44.7. pg_attribute

The catalog pg_attribute stores information about table columns. There will be exactly one pg_attribute row for every column in every table in the database. (There will also be attribute entries for indexes, and indeed all objects that have pg_class entries.)

The term attribute is equivalent to column and is used for historical reasons.

Table 44-7. pg_attribute Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The table this column belongs to</td>
</tr>
<tr>
<td>attname</td>
<td>name</td>
<td></td>
<td>The column name</td>
</tr>
<tr>
<td>atttypid</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>The data type of this column</td>
</tr>
</tbody>
</table>
### System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attstattarget</td>
<td>int4</td>
<td><code>attstattarget</code> controls the level of detail of statistics accumulated for this column by <em>ANALYZE</em>. A zero value indicates that no statistics should be collected. A negative value says to use the system default statistics target. The exact meaning of positive values is data type-dependent. For scalar data types, <code>attstattarget</code> is both the target number of “most common values” to collect, and the target number of histogram bins to create.</td>
</tr>
<tr>
<td>attlen</td>
<td>int2</td>
<td>A copy of <code>pg_type.typlen</code> of this column’s type.</td>
</tr>
<tr>
<td>attnum</td>
<td>int2</td>
<td>The number of the column. Ordinary columns are numbered from 1 up. System columns, such as <code>oid</code>, have (arbitrary) negative numbers.</td>
</tr>
<tr>
<td>attndims</td>
<td>int4</td>
<td>Number of dimensions, if the column is an array type; otherwise 0. (Presently, the number of dimensions of an array is not enforced, so any nonzero value effectively means “it’s an array”)</td>
</tr>
<tr>
<td>attcacheoff</td>
<td>int4</td>
<td>Always -1 in storage, but when loaded into a row descriptor in memory this might be updated to cache the offset of the attribute within the row.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>------------</td>
</tr>
<tr>
<td>atttypmod</td>
<td>int4</td>
<td></td>
</tr>
<tr>
<td>attbyval</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>attstorage</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>attalign</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>attnotnull</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>atthasdef</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>attisdropped</td>
<td>bool</td>
<td></td>
</tr>
</tbody>
</table>
### 44.8. `pg_authid`

The catalog `pg_authid` contains information about database authorization identifiers (roles). A role subsumes the concepts of “users” and “groups”. A user is essentially just a role with the `rolcanlogin` flag set. Any role (with or without `rolcanlogin`) can have other roles as members; see `pg_auth_members`.

Since this catalog contains passwords, it must not be publicly readable. `pg_roles` is a publicly readable view on `pg_authid` that blanks out the password field.

Chapter 19 contains detailed information about user and privilege management.

Because user identities are cluster-wide, `pg_authid` is shared across all databases of a cluster: there is only one copy of `pg_authid` per cluster, not one per database.

**Table 44-8. `pg_authid` Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolname</td>
<td>name</td>
<td>Role name</td>
</tr>
<tr>
<td>rolsuper</td>
<td>bool</td>
<td>Role has superuser privileges</td>
</tr>
<tr>
<td>rolinherit</td>
<td>bool</td>
<td>Role automatically inherits privileges of roles it is a member of</td>
</tr>
<tr>
<td>rolcreaterole</td>
<td>bool</td>
<td>Role can create more roles</td>
</tr>
<tr>
<td>rolcatedb</td>
<td>bool</td>
<td>Role can create databases</td>
</tr>
<tr>
<td>rolcateupdate</td>
<td>bool</td>
<td>Role can update system catalogs directly. (Even a superuser cannot do this unless this column is true)</td>
</tr>
</tbody>
</table>
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolcanlogin</td>
<td>bool</td>
<td>Role can log in. That is, this role can be given as the initial session authorization identifier</td>
</tr>
<tr>
<td>rolconnlimit</td>
<td>int4</td>
<td>For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit</td>
</tr>
<tr>
<td>rolpassword</td>
<td>text</td>
<td>Password (possibly encrypted); NULL if none</td>
</tr>
<tr>
<td>rolvaliduntil</td>
<td>timestamptz</td>
<td>Password expiry time (only used for password authentication); NULL if no expiration</td>
</tr>
<tr>
<td>rolconfig</td>
<td>text[]</td>
<td>Session defaults for run-time configuration variables</td>
</tr>
</tbody>
</table>

#### 44.9. pg_auth_members

The catalog `pg_auth_members` shows the membership relations between roles. Any non-circular set of relationships is allowed.

Because user identities are cluster-wide, `pg_auth_members` is shared across all databases of a cluster: there is only one copy of `pg_auth_members` per cluster, not one per database.

**Table 44-9. pg_auth_members Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>roleid</td>
<td>oid</td>
<td><code>pg_authid.oid</code></td>
<td>ID of a role that has a member</td>
</tr>
<tr>
<td>member</td>
<td>oid</td>
<td><code>pg_authid.oid</code></td>
<td>ID of a role that is a member of <code>roleid</code></td>
</tr>
<tr>
<td>grantor</td>
<td>oid</td>
<td><code>pg_authid.oid</code></td>
<td>ID of the role that granted this membership</td>
</tr>
<tr>
<td>admin_option</td>
<td>bool</td>
<td></td>
<td>True if member can grant membership in <code>roleid</code> to others</td>
</tr>
</tbody>
</table>

#### 44.10. pg_autovacuum

The catalog `pg_autovacuum` stores optional per-relation configuration parameters for the autovacuum daemon. If there is an entry here for a particular relation, the given parameters will be used for autovacuuming that table. If no entry is present, the system-wide defaults will be used. For more information about the autovacuum daemon, see Section 23.1.4.
Note: It is likely that \texttt{pg\_autovacuum} will disappear in a future release, with the information instead being kept in \texttt{pg\_class.reloptions} entries.

Table 44-10. \texttt{pg\_autovacuum} Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{vacrelid}</td>
<td>\texttt{oid}</td>
<td>\texttt{pg_class.oid}</td>
<td>The table this entry is for</td>
</tr>
<tr>
<td>\texttt{enabled}</td>
<td>\texttt{bool}</td>
<td></td>
<td>If false, this table will not be autovacuumed, except to prevent transaction ID wraparound</td>
</tr>
<tr>
<td>\texttt{vac_base_thresh}</td>
<td>\texttt{integer}</td>
<td></td>
<td>Minimum number of modified tuples before vacuum</td>
</tr>
<tr>
<td>\texttt{vac_scale_factor}</td>
<td>\texttt{float4}</td>
<td></td>
<td>Multiplier for \texttt{reltuples} to add to \texttt{vac_base_thresh}</td>
</tr>
<tr>
<td>\texttt{anl_base_thresh}</td>
<td>\texttt{integer}</td>
<td></td>
<td>Minimum number of modified tuples before analyze</td>
</tr>
<tr>
<td>\texttt{anl_scale_factor}</td>
<td>\texttt{float4}</td>
<td></td>
<td>Multiplier for \texttt{reltuples} to add to \texttt{anl_base_thresh}</td>
</tr>
<tr>
<td>\texttt{vac_cost_delay}</td>
<td>\texttt{integer}</td>
<td></td>
<td>Custom \texttt{vacuum_cost_delay} parameter</td>
</tr>
<tr>
<td>\texttt{vac_cost_limit}</td>
<td>\texttt{integer}</td>
<td></td>
<td>Custom \texttt{vacuum_cost_limit} parameter</td>
</tr>
<tr>
<td>\texttt{freeze_min_age}</td>
<td>\texttt{integer}</td>
<td></td>
<td>Custom \texttt{vacuum_freeze_min_age} parameter</td>
</tr>
<tr>
<td>\texttt{freeze_max_age}</td>
<td>\texttt{integer}</td>
<td></td>
<td>Custom \texttt{autovacuum_freeze_max_age} parameter</td>
</tr>
</tbody>
</table>

The autovacuum daemon will initiate a \texttt{VACUUM} operation on a particular table when the number of updated or deleted tuples exceeds \texttt{vac\_base\_thresh} plus \texttt{vac\_scale\_factor} times the number of live tuples currently estimated to be in the relation. Similarly, it will initiate an \texttt{ANALYZE} operation when the number of inserted, updated or deleted tuples exceeds \texttt{anl\_base\_thresh} plus \texttt{anl\_scale\_factor} times the number of live tuples currently estimated to be in the relation.

Also, the autovacuum daemon will perform a \texttt{VACUUM} operation to prevent transaction ID wraparound if the table’s \texttt{pg\_class.relfrozenxid} field attains an age of more than \texttt{freeze\_max\_age} transactions, whether the table has been changed or not, even if \texttt{pg\_autovacuum\_enabled} is set to false for it. The system will launch autovacuum to perform such \texttt{VACUUM}s even if autovacuum is otherwise disabled. See Section 23.1.3 for more about wraparound prevention.
Any of the numerical fields can contain -1 (or indeed any negative value) to indicate that the system-wide default should be used for this particular value. Observe that the `vac_cost_delay` variable inherits its default value from the `autovacuum_vacuum_cost_delay` configuration parameter, or from `vacuum_cost_delay` if the former is set to a negative value. The same applies to `vac_cost_limit`. Also, `autovacuum` will ignore attempts to set a per-table `freeze_max_age` larger than the system-wide setting (it can only be set smaller), and the `freeze_min_age` value will be limited to half the system-wide `autovacuum_freeze_max_age` setting. Note that while you can set `freeze_max_age` very small, or even zero, this is usually unwise since it will force frequent vacuuming.

### 44.11. `pg_cast`

The catalog `pg_cast` stores data type conversion paths, both built-in paths and those defined with `CREATE CAST`.

It should be noted that `pg_cast` does not represent every type conversion that the system knows how to perform; only those that cannot be deduced from some generic rule. For example, casting between a domain and its base type is not explicitly represented in `pg_cast`. Another important exception is that “I/O conversion casts”, those performed using a data type’s own I/O functions to convert to or from `text` or other string types, are not explicitly represented in `pg_cast`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>castsource</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>OID of the source data type</td>
</tr>
<tr>
<td>casttarget</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>OID of the target data type</td>
</tr>
<tr>
<td>castfunc</td>
<td>oid</td>
<td>pg_proc.oid</td>
<td>The OID of the function to use to perform this cast. Zero is stored if the data types are binary compatible (that is, no run-time operation is needed to perform the cast)</td>
</tr>
<tr>
<td>castcontext</td>
<td>char</td>
<td></td>
<td>Indicates what contexts the cast can be invoked in. e means only as an explicit cast (using <code>CAST</code> or <code>::</code> syntax), a means implicitly in assignment to a target column, as well as explicitly. i means implicitly in expressions, as well as the other cases</td>
</tr>
</tbody>
</table>
The cast functions listed in `pg_cast` must always take the cast source type as their first argument type, and return the cast destination type as their result type. A cast function can have up to three arguments. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise.

It is legitimate to create a `pg_cast` entry in which the source and target types are the same, if the associated function takes more than one argument. Such entries represent “length coercion functions” that coerce values of the type to be legal for a particular type modifier value.

When a `pg_cast` entry has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

### 44.12. `pg_class`

The catalog `pg_class` catalogs tables and most everything else that has columns or is otherwise similar to a table. This includes indexes (but see also `pg_index`), sequences, views, composite types, and TOAST tables; see `relkind`. Below, when we mean all of these kinds of objects we speak of “relations”. Not all columns are meaningful for all relation types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relname</td>
<td>name</td>
<td></td>
<td>Name of the table, index, view, etc.</td>
</tr>
<tr>
<td>relnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this relation</td>
</tr>
<tr>
<td>reltype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>The OID of the data type that corresponds to this table’s row type, if any (zero for indexes, which have no <code>pg_type</code> entry)</td>
</tr>
<tr>
<td>relowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the relation</td>
</tr>
<tr>
<td>relam</td>
<td>oid</td>
<td>pg_am.oid</td>
<td>If this is an index, the access method used (B-tree, hash, etc.)</td>
</tr>
<tr>
<td>relfilenode</td>
<td>oid</td>
<td></td>
<td>Name of the on-disk file of this relation; 0 if none</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>reltablespace</td>
<td>oid</td>
<td>pg_tablespace.oid</td>
<td>The tablespace in which this relation is stored. If zero, the database’s default tablespace is implied. (Not meaningful if the relation has no on-disk file.)</td>
</tr>
<tr>
<td>relpages</td>
<td>int4</td>
<td></td>
<td>Size of the on-disk representation of this table in pages (of size BLCKSZ). This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX.</td>
</tr>
<tr>
<td>reltuples</td>
<td>float4</td>
<td></td>
<td>Number of rows in the table. This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX.</td>
</tr>
<tr>
<td>reltoastrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>OID of the TOAST table associated with this table, 0 if none. The TOAST table stores large attributes “out of line” in a secondary table.</td>
</tr>
<tr>
<td>reltoastidxid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>For a TOAST table, the OID of its index. 0 if not a TOAST table.</td>
</tr>
<tr>
<td>relhasindex</td>
<td>bool</td>
<td></td>
<td>True if this is a table and it has (or recently had) any indexes. This is set by CREATE INDEX, but not cleared immediately by DROP INDEX. VACUUM clears relhasindex if it finds the table has no indexes.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>relissshared</td>
<td>bool</td>
<td>True if this table is shared across all databases in the cluster. Only certain system catalogs (such as pg_database) are shared</td>
<td></td>
</tr>
<tr>
<td>relkind</td>
<td>char</td>
<td>( r = ) ordinary table, ( i = ) index, ( S = ) sequence, ( v = ) view, ( c = ) composite type, ( t = ) TOAST table</td>
<td></td>
</tr>
<tr>
<td>relnatts</td>
<td>int2</td>
<td>Number of user columns in the relation (system columns not counted). There must be this many corresponding entries in pg_attribute. See also pg_attribute.attnum</td>
<td></td>
</tr>
<tr>
<td>relchecks</td>
<td>int2</td>
<td>Number of check constraints on the table; see pg_constraint catalog</td>
<td></td>
</tr>
<tr>
<td>reltriggers</td>
<td>int2</td>
<td>Number of triggers on the table; see pg_trigger catalog</td>
<td></td>
</tr>
<tr>
<td>relukeys</td>
<td>int2</td>
<td>Unused (not the number of unique keys)</td>
<td></td>
</tr>
<tr>
<td>relfkeys</td>
<td>int2</td>
<td>Unused (not the number of foreign keys on the table)</td>
<td></td>
</tr>
<tr>
<td>relrefs</td>
<td>int2</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>relhasoids</td>
<td>bool</td>
<td>True if we generate an OID for each row of the relation</td>
<td></td>
</tr>
<tr>
<td>relhaspkey</td>
<td>bool</td>
<td>True if the table has (or once had) a primary key</td>
<td></td>
</tr>
<tr>
<td>relhasrules</td>
<td>bool</td>
<td>True if table has rules; see pg_rewrite catalog</td>
<td></td>
</tr>
<tr>
<td>relhassubclass</td>
<td>bool</td>
<td>True if table has (or once had) any inheritance children</td>
<td></td>
</tr>
</tbody>
</table>
44.13. pg_constraint

The catalog pg_constraint stores check, primary key, unique, and foreign key constraints on tables. (Column constraints are not treated specially. Every column constraint is equivalent to some table constraint.) Not-null constraints are represented in the pg_attribute catalog.

Check constraints on domains are stored here, too.

Table 44-13. pg_constraint Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conname</td>
<td>name</td>
<td></td>
<td>Constraint name (not necessarily unique!)</td>
</tr>
<tr>
<td>connamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this constraint</td>
</tr>
<tr>
<td>connamespace</td>
<td>oid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cotype</td>
<td>char</td>
<td></td>
<td>c = check constraint, f = foreign key constraint, p = primary key constraint, u = unique constraint</td>
</tr>
</tbody>
</table>
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condeferrable</td>
<td>bool</td>
<td></td>
<td>Is the constraint deferrable?</td>
</tr>
<tr>
<td>condeferred</td>
<td>bool</td>
<td></td>
<td>Is the constraint deferred by default?</td>
</tr>
<tr>
<td>conrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The table this constraint is on; 0 if not a table constraint</td>
</tr>
<tr>
<td>contypid</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>The domain this constraint is on; 0 if not a domain constraint</td>
</tr>
<tr>
<td>confrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>If a foreign key, the referenced table; else 0</td>
</tr>
<tr>
<td>confupdtype</td>
<td>char</td>
<td></td>
<td>Foreign key update action code</td>
</tr>
<tr>
<td>confdeltype</td>
<td>char</td>
<td></td>
<td>Foreign key deletion action code</td>
</tr>
<tr>
<td>confmatchtype</td>
<td>char</td>
<td></td>
<td>Foreign key match type</td>
</tr>
<tr>
<td>conkey</td>
<td>int2[]</td>
<td>pg_attribute.attnum</td>
<td>If a table constraint, list of columns which the constraint constrains</td>
</tr>
<tr>
<td>confkey</td>
<td>int2[]</td>
<td>pg_attribute.attnum</td>
<td>If a foreign key, list of the referenced columns</td>
</tr>
<tr>
<td>compfeqop</td>
<td>oid[]</td>
<td>pg_operator.oid</td>
<td>If a foreign key, list of the equality operators for PK = FK comparisons</td>
</tr>
<tr>
<td>coneqop</td>
<td>oid[]</td>
<td>pg_operator.oid</td>
<td>If a foreign key, list of the equality operators for PK = PK comparisons</td>
</tr>
<tr>
<td>confseqop</td>
<td>oid[]</td>
<td>pg_operator.oid</td>
<td>If a foreign key, list of the equality operators for FK = FK comparisons</td>
</tr>
<tr>
<td>conbin</td>
<td>text</td>
<td></td>
<td>If a check constraint, an internal representation of the expression</td>
</tr>
<tr>
<td>consrc</td>
<td>text</td>
<td></td>
<td>If a check constraint, a human-readable representation of the expression</td>
</tr>
</tbody>
</table>

**Note:** `consrc` is not updated when referenced objects change; for example, it won’t track renaming of columns. Rather than relying on this field, it’s best to use `pg_get_constraintdef()` to
extract the definition of a check constraint.

Note: `pg_class.relchecks` needs to agree with the number of check-constraint entries found in this table for each relation.

### 44.14. pg_conversion

The catalog `pg_conversion` describes the available encoding conversion procedures. See `CREATE CONVERSION` for more information.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conname</td>
<td>name</td>
<td></td>
<td>Conversion name (unique within a namespace)</td>
</tr>
<tr>
<td>connamespace</td>
<td>oid</td>
<td><code>pg_namespace.oid</code></td>
<td>The OID of the namespace that contains this conversion</td>
</tr>
<tr>
<td>conowner</td>
<td>oid</td>
<td><code>pg_authid.oid</code></td>
<td>Owner of the conversion</td>
</tr>
<tr>
<td>conforencoding</td>
<td>int4</td>
<td></td>
<td>Source encoding ID</td>
</tr>
<tr>
<td>contoencoding</td>
<td>int4</td>
<td></td>
<td>Destination encoding ID</td>
</tr>
<tr>
<td>conproc</td>
<td>regproc</td>
<td><code>pg_proc.oid</code></td>
<td>Conversion procedure</td>
</tr>
<tr>
<td>condefault</td>
<td>bool</td>
<td></td>
<td>True if this is the default conversion</td>
</tr>
</tbody>
</table>

### 44.15. pg_database

The catalog `pg_database` stores information about the available databases. Databases are created with the `CREATE DATABASE` command. Consult Chapter 20 for details about the meaning of some of the parameters.

Unlike most system catalogs, `pg_database` is shared across all databases of a cluster: there is only one copy of `pg_database` per cluster, not one per database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datname</td>
<td>name</td>
<td></td>
<td>Database name</td>
</tr>
</tbody>
</table>
## Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datdba</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the database, usually the user who created it</td>
</tr>
<tr>
<td>encoding</td>
<td>int4</td>
<td></td>
<td>Character encoding for this database (pg_encoding_to_char() can translate this number to the encoding name)</td>
</tr>
<tr>
<td>datistemplate</td>
<td>bool</td>
<td></td>
<td>If true then this database can be used in the TEMPLATE clause of CREATE DATABASE to create a new database as a clone of this one</td>
</tr>
<tr>
<td>datallowconn</td>
<td>bool</td>
<td></td>
<td>If false then no one can connect to this database. This is used to protect the template0 database from being altered</td>
</tr>
<tr>
<td>datconnlimit</td>
<td>int4</td>
<td></td>
<td>Sets maximum number of concurrent connections that can be made to this database. -1 means no limit</td>
</tr>
<tr>
<td>datlastsysoid</td>
<td>oid</td>
<td></td>
<td>Last system OID in the database; useful particularly to pg_dump</td>
</tr>
<tr>
<td>datfrozenxid</td>
<td>xid</td>
<td></td>
<td>All transaction IDs before this one have been replaced with a permanent (&quot;frozen&quot;) transaction ID in this database. This is used to track whether the database needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. It is the minimum of the per-table pg_class.relfrozenxid values</td>
</tr>
</tbody>
</table>
**Chapter 44. System Catalogs**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dattablespace</td>
<td>oid</td>
<td>pg_tablespace.oid</td>
<td>The default tablespace for the database. Within this database, all tables for which pg_class.reltablespace is zero will be stored in this tablespace; in particular, all the non-shared system catalogs will be there</td>
</tr>
<tr>
<td>datconfig</td>
<td>text[]</td>
<td></td>
<td>Session defaults for run-time configuration variables</td>
</tr>
<tr>
<td>datacl</td>
<td>aclitem[]</td>
<td></td>
<td>Access privileges; see GRANT and REVOKE for details</td>
</tr>
</tbody>
</table>

### 44.16. pg_depend

The catalog `pg_depend` records the dependency relationships between database objects. This information allows DROP commands to find which other objects must be dropped by DROP CASCADE or prevent dropping in the DROP RESTRICT case.

See also `pg_shdepend`, which performs a similar function for dependencies involving objects that are shared across a database cluster.

**Table 44-16. pg_depend Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>classid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The OID of the system catalog the dependent object is in</td>
</tr>
<tr>
<td>objid</td>
<td>oid</td>
<td>any OID column</td>
<td>The OID of the specific dependent object</td>
</tr>
<tr>
<td>objsubid</td>
<td>int4</td>
<td></td>
<td>For a table column, this is the column number (the objid and classid refer to the table itself). For all other object types, this column is zero</td>
</tr>
<tr>
<td>refclassid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The OID of the system catalog the referenced object is in</td>
</tr>
<tr>
<td>refobjid</td>
<td>oid</td>
<td>any OID column</td>
<td>The OID of the specific referenced object</td>
</tr>
</tbody>
</table>
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refobjsubid</td>
<td>int4</td>
<td></td>
<td>For a table column, this is the column number (the refobjid and refclassid refer to the table itself). For all other object types, this column is zero.</td>
</tr>
<tr>
<td>deptype</td>
<td>char</td>
<td></td>
<td>A code defining the specific semantics of this dependency relationship; see text</td>
</tr>
</tbody>
</table>

In all cases, a `pg_depend` entry indicates that the referenced object cannot be dropped without also dropping the dependent object. However, there are several subflavors identified by `deptype`:

**DEPENDENCY_NORMAL (n)**

A normal relationship between separately-created objects. The dependent object can be dropped without affecting the referenced object. The referenced object can only be dropped by specifying `CASCADE`, in which case the dependent object is dropped, too. Example: a table column has a normal dependency on its data type.

**DEPENDENCY_AUTO (a)**

The dependent object can be dropped separately from the referenced object, and should be automatically dropped (regardless of `RESTRICT` or `CASCADE` mode) if the referenced object is dropped. Example: a named constraint on a table is made autodependent on the table, so that it will go away if the table is dropped.

**DEPENDENCY_INTERNAL (i)**

The dependent object was created as part of creation of the referenced object, and is really just a part of its internal implementation. A `DROP` of the dependent object will be disallowed outright (we’ll tell the user to issue a `DROP` against the referenced object, instead). A `DROP` of the referenced object will be propagated through to drop the dependent object whether `CASCADE` is specified or not. Example: a trigger that’s created to enforce a foreign-key constraint is made internally dependent on the constraint’s `pg_constraint` entry.

**DEPENDENCY_PIN (p)**

There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by `initdb`. The columns for the dependent object contain zeroes.

Other dependency flavors might be needed in future.

**44.17. pg_description**

The catalog `pg_description` stores optional descriptions (comments) for each database object. Descriptions can be manipulated with the `COMMENT` command and viewed with psql’s `\d` commands. Descriptions of many built-in system objects are provided in the initial contents of `pg_description`.

See also `pg_shdescription`, which performs a similar function for descriptions involving objects that are shared across a database cluster.
44.18. pg_enum

The pg_enum catalog contains entries matching enum types to their associated values and labels. The internal representation of a given enum value is actually the OID of its associated row in pg_enum. The OIDs for a particular enum type are guaranteed to be ordered in the way the type should sort, but there is no guarantee about the ordering of OIDs of unrelated enum types.

Table 44-18. pg_enum Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enumtypid</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>The OID of the pg_type entry owning this enum value</td>
</tr>
<tr>
<td>enumlabel</td>
<td>name</td>
<td></td>
<td>The textual label for this enum value</td>
</tr>
</tbody>
</table>

44.19. pg_index

The catalog pg_index contains part of the information about indexes. The rest is mostly in pg_class.

Table 44-19. pg_index Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>indexrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The OID of the pg_class entry for this index</td>
</tr>
<tr>
<td>indrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The OID of the pg_class entry for the table this index is for</td>
</tr>
<tr>
<td>indnatts</td>
<td>int2</td>
<td></td>
<td>The number of columns in the index (duplicates pg_class.relnatts)</td>
</tr>
<tr>
<td>indisunique</td>
<td>bool</td>
<td></td>
<td>If true, this is a unique index</td>
</tr>
<tr>
<td>indisprimary</td>
<td>bool</td>
<td></td>
<td>If true, this index represents the primary key of the table (indisunique should always be true when this is true)</td>
</tr>
<tr>
<td>indisclustered</td>
<td>bool</td>
<td></td>
<td>If true, the table was last clustered on this index</td>
</tr>
<tr>
<td>indisvalid</td>
<td>bool</td>
<td></td>
<td>If true, the index is currently valid for queries. False means the index is possibly incomplete: it must still be modified by INSERT/UPDATE operations, but it cannot safely be used for queries. If it is unique, the uniqueness property is not true either</td>
</tr>
<tr>
<td>indcheckxmin</td>
<td>bool</td>
<td></td>
<td>If true, queries must not use the index until the xmin of this pg_index row is below their TransactionXmin event horizon, because the table may contain broken HOT chains with incompatible rows that they can see</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>indisready</td>
<td>bool</td>
<td></td>
<td>If true, the index is currently ready for inserts. False means the index must be ignored by INSERT/UPDATE operations</td>
</tr>
<tr>
<td>indkey</td>
<td>int2vector</td>
<td>pg_attribute.attnum</td>
<td>This is an array of indnatts values that indicate which table columns this index indexes. For example a value of 1 3 would mean that the first and the third table columns make up the index key. A zero in this array indicates that the corresponding index attribute is an expression over the table columns, rather than a simple column reference</td>
</tr>
<tr>
<td>indclass</td>
<td>oidvector</td>
<td>pg_opclass.oid</td>
<td>For each column in the index key, this contains the OID of the operator class to use. See pg_opclass for details</td>
</tr>
<tr>
<td>indoption</td>
<td>int2vector</td>
<td></td>
<td>This is an array of indnatts values that store per-column flag bits. The meaning of the bits is defined by the index’s access method</td>
</tr>
<tr>
<td>indexprs</td>
<td>text</td>
<td></td>
<td>Expression trees (in nodeToString() representation) for index attributes that are not simple column references. This is a list with one element for each zero entry in indkey. NULL if all index attributes are simple references</td>
</tr>
</tbody>
</table>

1292
### 44.20. pg_inherits

The catalog `pg_inherits` records information about table inheritance hierarchies. There is one entry for each direct child table in the database. (Indirect inheritance can be determined by following chains of entries.)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inhrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The OID of the child table</td>
</tr>
<tr>
<td>inhparent</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The OID of the parent table</td>
</tr>
<tr>
<td>inhseqno</td>
<td>int4</td>
<td></td>
<td>If there is more than one direct parent for a child table (multiple inheritance), this number tells the order in which the inherited columns are to be arranged. The count starts at 1</td>
</tr>
</tbody>
</table>

### 44.21. pg_language

The catalog `pg_language` registers languages in which you can write functions or stored procedures. See `CREATE LANGUAGE` and Chapter 37 for more information about language handlers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lanname</td>
<td>name</td>
<td></td>
<td>Name of the language</td>
</tr>
<tr>
<td>lanowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the language</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>lanispl</td>
<td>bool</td>
<td></td>
<td>This is false for internal languages (such as SQL) and true for user-defined languages. Currently, pg_dump still uses this to determine which languages need to be dumped, but this might be replaced by a different mechanism in the future.</td>
</tr>
<tr>
<td>lanpltrusted</td>
<td>bool</td>
<td></td>
<td>True if this is a trusted language, which means that it is believed not to grant access to anything outside the normal SQL execution environment. Only superusers can create functions in untrusted languages.</td>
</tr>
<tr>
<td>lanplcallfoid</td>
<td>oid</td>
<td>pg_proc.oid</td>
<td>For noninternal languages this references the language handler, which is a special function that is responsible for executing all functions that are written in the particular language.</td>
</tr>
<tr>
<td>lanvalidator</td>
<td>oid</td>
<td>pg_proc.oid</td>
<td>This references a language validator function that is responsible for checking the syntax and validity of new functions when they are created. Zero if no validator is provided.</td>
</tr>
<tr>
<td>lanacl</td>
<td>aclitem[]</td>
<td></td>
<td>Access privileges; see <code>GRANT</code> and <code>REVOKE</code> for details.</td>
</tr>
</tbody>
</table>

### 44.22. `pg_largeobject`

The catalog `pg_largeobject` holds the data making up “large objects”. A large object is identified
by an OID assigned when it is created. Each large object is broken into segments or “pages” small enough to be conveniently stored as rows in \textit{pg\_largeobject}. The amount of data per page is defined to be \texttt{LOBLKSIZE} (which is currently \texttt{BLCKSZ/4}, or typically 2 kB).

### Table 44-22. \texttt{pg\_largeobject} Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{loid}</td>
<td>\texttt{oid}</td>
<td>Identifier of the large object that includes this page</td>
</tr>
<tr>
<td>\texttt{pageno}</td>
<td>\texttt{int4}</td>
<td>Page number of this page within its large object (counting from zero)</td>
</tr>
<tr>
<td>\texttt{data}</td>
<td>\texttt{bytea}</td>
<td>Actual data stored in the large object. This will never be more than \texttt{LOBLKSIZE} bytes and might be less</td>
</tr>
</tbody>
</table>

Each row of \texttt{pg\_largeobject} holds data for one page of a large object, beginning at byte offset \((\texttt{pageno} \times \texttt{LOBLKSIZE})\) within the object. The implementation allows sparse storage: pages might be missing, and might be shorter than \texttt{LOBLKSIZE} bytes even if they are not the last page of the object. Missing regions within a large object read as zeroes.

### 44.23. \texttt{pg\_listener}

The catalog \texttt{pg\_listener} supports the \texttt{LISTEN} and \texttt{NOTIFY} commands. A listener creates an entry in \texttt{pg\_listener} for each notification name it is listening for. A notifier scans \texttt{pg\_listener} and updates each matching entry to show that a notification has occurred. The notifier also sends a signal (using the PID recorded in the table) to awaken the listener from sleep.

### Table 44-23. \texttt{pg\_listener} Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{relname}</td>
<td>\texttt{name}</td>
<td></td>
<td>Notify condition name. (The name need not match any actual relation in the database; the name \texttt{relname} is historical.)</td>
</tr>
<tr>
<td>\texttt{listenerpid}</td>
<td>\texttt{int4}</td>
<td></td>
<td>PID of the server process that created this entry</td>
</tr>
<tr>
<td>\texttt{notification}</td>
<td>\texttt{int4}</td>
<td></td>
<td>Zero if no event is pending for this listener. If an event is pending, the PID of the server process that sent the notification</td>
</tr>
</tbody>
</table>
44.24. pg_namespace

The catalog pg_namespace stores namespaces. A namespace is the structure underlying SQL schemas: each namespace can have a separate collection of relations, types, etc. without name conflicts.

Table 44-24. pg_namespace Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nspname</td>
<td>name</td>
<td></td>
<td>Name of the namespace</td>
</tr>
<tr>
<td>nspowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the namespace</td>
</tr>
<tr>
<td>nspacl</td>
<td>aclitem[]</td>
<td></td>
<td>Access privileges; see GRANT and REVOKE for details</td>
</tr>
</tbody>
</table>

44.25. pg_opclass

The catalog pg_opclass defines index access method operator classes. Each operator class defines semantics for index columns of a particular data type and a particular index access method. An operator class essentially specifies that a particular operator family is applicable to a particular indexable column data type. The set of operators from the family that are actually usable with the indexed column are whichever ones accept the column’s data type as their lefthand input.

Operator classes are described at length in Section 34.14.

Table 44-25. pg_opclass Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcmethod</td>
<td>oid</td>
<td>pg_sm.oid</td>
<td>Index access method operator class is for</td>
</tr>
<tr>
<td>opcname</td>
<td>name</td>
<td></td>
<td>Name of this operator class</td>
</tr>
<tr>
<td>opcnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>Namespace of this operator class</td>
</tr>
<tr>
<td>opcowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the operator class</td>
</tr>
<tr>
<td>opcfamily</td>
<td>oid</td>
<td>pg_opfamily.oid</td>
<td>Operator family containing the operator class</td>
</tr>
<tr>
<td>opcintype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Data type that the operator class indexes</td>
</tr>
<tr>
<td>opcdefault</td>
<td>bool</td>
<td></td>
<td>True if this operator class is the default for opcintype</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opckeytype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Type of data stored in index, or zero if same as opcintype</td>
</tr>
</tbody>
</table>

An operator class’s opcmethod must match the opfmethod of its containing operator family. Also, there must be no more than one pg_opclass row having opcdefault true for any given combination of opcmethod and opcintype.

44.26. pg_operator

The catalog pg_operator stores information about operators. See CREATE OPERATOR and Section 34.12 for more information.

Table 44-26. pg_operator Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oprname</td>
<td>name</td>
<td></td>
<td>Name of the operator</td>
</tr>
<tr>
<td>oprnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this operator</td>
</tr>
<tr>
<td>oprowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the operator</td>
</tr>
<tr>
<td>oprkind</td>
<td>char</td>
<td></td>
<td>b = infix (“both”), l = prefix (“left”), r = postfix (“right”)</td>
</tr>
<tr>
<td>oprcanmerge</td>
<td>bool</td>
<td></td>
<td>This operator supports merge joins</td>
</tr>
<tr>
<td>oprcanhash</td>
<td>bool</td>
<td></td>
<td>This operator supports hash joins</td>
</tr>
<tr>
<td>oprleft</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Type of the left operand</td>
</tr>
<tr>
<td>oprright</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Type of the right operand</td>
</tr>
<tr>
<td>oprresult</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Type of the result</td>
</tr>
<tr>
<td>oprcom</td>
<td>oid</td>
<td>pg_operator.oid</td>
<td>Commutator of this operator, if any</td>
</tr>
<tr>
<td>oprnegate</td>
<td>oid</td>
<td>pg_operator.oid</td>
<td>Negator of this operator, if any</td>
</tr>
<tr>
<td>oprcode</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Function that implements this operator</td>
</tr>
<tr>
<td>oprrest</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Restriction selectivity estimation function for this operator</td>
</tr>
<tr>
<td>oprjoin</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Join selectivity estimation function for this operator</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

Unused column contain zeroes. For example, oprleft is zero for a prefix operator.

### 44.27. pg_opfamily

The catalog `pg_opfamily` defines operator families. Each operator family is a collection of operators and associated support routines that implement the semantics specified for a particular index access method. Furthermore, the operators in a family are all “compatible”, in a way that is specified by the access method. The operator family concept allows cross-data-type operators to be used with indexes and to be reasoned about using knowledge of access method semantics.

Operator families are described at length in Section 34.14.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opfmethod</td>
<td>oid</td>
<td>pg_am.oid</td>
<td>Index access method operator family is for</td>
</tr>
<tr>
<td>opfname</td>
<td>name</td>
<td></td>
<td>Name of this operator family</td>
</tr>
<tr>
<td>opfnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>Namespace of this operator family</td>
</tr>
<tr>
<td>opfowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the operator family</td>
</tr>
</tbody>
</table>

The majority of the information defining an operator family is not in its `pg_opfamily` row, but in the associated rows in `pg_amop`, `pg_amproc`, and `pg_opclass`.

### 44.28. pg_pltemplate

The catalog `pg_pltemplate` stores “template” information for procedural languages. A template for a language allows the language to be created in a particular database by a simple `CREATE LANGUAGE` command, with no need to specify implementation details.

Unlike most system catalogs, `pg_pltemplate` is shared across all databases of a cluster: there is only one copy of `pg_pltemplate` per cluster, not one per database. This allows the information to be accessible in each database as it is needed.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmplname</td>
<td>name</td>
<td>Name of the language this template is for</td>
</tr>
<tr>
<td>tmpltrusted</td>
<td>boolean</td>
<td>True if language is considered trusted</td>
</tr>
<tr>
<td>tmpldbcreate</td>
<td>boolean</td>
<td>True if language may be created by a database owner</td>
</tr>
<tr>
<td>tmplhandler</td>
<td>text</td>
<td>Name of call handler function</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmplvalidator</td>
<td>text</td>
<td>Name of validator function, or NULL if none</td>
</tr>
<tr>
<td>tmpllibrary</td>
<td>text</td>
<td>Path of shared library that implements language</td>
</tr>
<tr>
<td>tmplacl</td>
<td>aclitem[]</td>
<td>Access privileges for template (not yet used)</td>
</tr>
</tbody>
</table>

There are not currently any commands that manipulate procedural language templates; to change the built-in information, a superuser must modify the table using ordinary `INSERT`, `DELETE`, or `UPDATE` commands. It is likely that a future release of PostgreSQL will offer commands to change the entries in a cleaner fashion.

When implemented, the `tmplacl` field will provide access control for the template itself (i.e., the right to create a language using it), not for the languages created from the template.

### 44.29. `pg_proc`

The catalog `pg_proc` stores information about functions (or procedures). See `CREATE FUNCTION` and Section 34.3 for more information.

The table contains data for aggregate functions as well as plain functions. If `proisagg` is true, there should be a matching row in `pg_aggregate`.

#### Table 44-29. `pg_proc` Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proname</td>
<td>name</td>
<td></td>
<td>Name of the function</td>
</tr>
<tr>
<td>pronamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this function</td>
</tr>
<tr>
<td>proowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the function</td>
</tr>
<tr>
<td>prolang</td>
<td>oid</td>
<td>pg_language.oid</td>
<td>Implementation language or call interface of this function</td>
</tr>
<tr>
<td>procost</td>
<td>float4</td>
<td></td>
<td>Estimated execution cost (in units of <code>cpu_operator_cost</code>); if <code>proretset</code>, this is cost per row returned</td>
</tr>
<tr>
<td>prorows</td>
<td>float4</td>
<td></td>
<td>Estimated number of result rows (zero if not <code>proretset</code>)</td>
</tr>
<tr>
<td>proisagg</td>
<td>bool</td>
<td></td>
<td>Function is an aggregate function</td>
</tr>
<tr>
<td>prosecdif</td>
<td>bool</td>
<td></td>
<td>Function is a security definer (i.e., a &quot;setuid&quot; function)</td>
</tr>
</tbody>
</table>
## Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proisstrict</td>
<td>bool</td>
<td></td>
<td>Function returns null if any call argument is null. In that case the function won’t actually be called at all. Functions that are not “strict” must be prepared to handle null inputs</td>
</tr>
<tr>
<td>proretset</td>
<td>bool</td>
<td></td>
<td>Function returns a set (i.e., multiple values of the specified data type)</td>
</tr>
<tr>
<td>provolatile</td>
<td>char</td>
<td></td>
<td>provolatile tells whether the function’s result depends only on its input arguments, or is affected by outside factors. It is i for “immutable” functions, which always deliver the same result for the same inputs. It is s for “stable” functions, whose results (for fixed inputs) do not change within a scan. It is v for “volatile” functions, whose results might change at any time. (Use v also for functions with side-effects, so that calls to them cannot get optimized away.)</td>
</tr>
<tr>
<td>pronargs</td>
<td>int2</td>
<td></td>
<td>Number of arguments</td>
</tr>
<tr>
<td>prorettype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>Data type of the return value</td>
</tr>
<tr>
<td>proargtypes</td>
<td>oidvector</td>
<td>pg_type.oid</td>
<td>An array with the data types of the function arguments. This includes only input arguments (including INOUT arguments), and thus represents the call signature of the function</td>
</tr>
</tbody>
</table>
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proallargtypes</td>
<td>oid[]</td>
<td>pg_type.oid</td>
<td>An array with the data types of the function arguments. This includes all arguments (including OUT and INOUT arguments); however, if all the arguments are IN arguments, this field will be null. Note that subscripting is 1-based, whereas for historical reasons proargtypes is subscripted from 0</td>
</tr>
<tr>
<td>proargmodes</td>
<td>char[]</td>
<td></td>
<td>An array with the modes of the function arguments, encoded as i for IN arguments, o for OUT arguments, b for INOUT arguments. If all the arguments are IN arguments, this field will be null. Note that subscripts correspond to positions of proallargtypes not proargtypes</td>
</tr>
<tr>
<td>proargnames</td>
<td>text[]</td>
<td></td>
<td>An array with the names of the function arguments. Arguments without a name are set to empty strings in the array. If none of the arguments have a name, this field will be null. Note that subscripts correspond to positions of proallargtypes not proargtypes</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prosrc</td>
<td>text</td>
<td></td>
<td>This tells the function handler how to invoke the function. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending on the implementation language/call convention</td>
</tr>
<tr>
<td>probin</td>
<td>bytea</td>
<td></td>
<td>Additional information about how to invoke the function. Again, the interpretation is language-specific</td>
</tr>
<tr>
<td>proconfig</td>
<td>text[]</td>
<td></td>
<td>Function’s local settings for run-time configuration variables</td>
</tr>
<tr>
<td>proacl</td>
<td>aclitem[]</td>
<td></td>
<td>Access privileges; see GRANT and REVOKE for details</td>
</tr>
</tbody>
</table>

For compiled functions, both built-in and dynamically loaded, prosrc contains the function’s C-language name (link symbol). For all other currently-known language types, prosrc contains the function’s source text. probin is unused except for dynamically-loaded C functions, for which it gives the name of the shared library file containing the function.

44.30. pg_rewrite

The catalog pg_rewrite stores rewrite rules for tables and views.

Table 44-30. pg_rewrite Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rulename</td>
<td>name</td>
<td></td>
<td>Rule name</td>
</tr>
<tr>
<td>ev_class</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The table this rule is for</td>
</tr>
<tr>
<td>ev_attr</td>
<td>int2</td>
<td></td>
<td>The column this rule is for (currently, always zero to indicate the whole table)</td>
</tr>
</tbody>
</table>
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ev_type</td>
<td>char</td>
<td></td>
<td>Event type that the rule is for: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE</td>
</tr>
<tr>
<td>ev_enabled</td>
<td>char</td>
<td></td>
<td>Controls in which session_replication_role modes the rule fires. O = rule fires in “origin” and “local” modes, D = rule is disabled, R = rule fires in “replica” mode, A = rule fires always.</td>
</tr>
<tr>
<td>is_instead</td>
<td>bool</td>
<td></td>
<td>True if the rule is an INSTEAD rule</td>
</tr>
<tr>
<td>ev_qual</td>
<td>text</td>
<td></td>
<td>Expression tree (in the form of a nodeToString() representation) for the rule’s qualifying condition</td>
</tr>
<tr>
<td>ev_action</td>
<td>text</td>
<td></td>
<td>Query tree (in the form of a nodeToString() representation) for the rule’s action</td>
</tr>
</tbody>
</table>

**Note:** pg_class.relhasrules must be true if a table has any rules in this catalog.

### 44.31. pg_shdepend

The catalog pg_shdepend records the dependency relationships between database objects and shared objects, such as roles. This information allows PostgreSQL to ensure that those objects are unreferenced before attempting to delete them.

See also pg_depend, which performs a similar function for dependencies involving objects within a single database.

Unlike most system catalogs, pg_shdepend is shared across all databases of a cluster: there is only one copy of pg_shdepend per cluster, not one per database.

#### Table 44-31. pg_shdepend Columns

| Name      | Type    | References | Description |
|-----------|---------|------------|-------------|-------------|
|           |         |            |             |             |
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbid</td>
<td>oid</td>
<td>pg_database.oid</td>
<td>The OID of the database the dependent object is in, or zero for a shared object</td>
</tr>
<tr>
<td>classid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The OID of the system catalog the dependent object is in</td>
</tr>
<tr>
<td>objid</td>
<td>oid</td>
<td>any OID column</td>
<td>The OID of the specific dependent object</td>
</tr>
<tr>
<td>refclassid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The OID of the system catalog the referenced object is in (must be a shared catalog)</td>
</tr>
<tr>
<td>refobjid</td>
<td>oid</td>
<td>any OID column</td>
<td>The OID of the specific referenced object</td>
</tr>
<tr>
<td>deptype</td>
<td>char</td>
<td></td>
<td>A code defining the specific semantics of this dependency relationship; see text</td>
</tr>
</tbody>
</table>

In all cases, a `pg_shdepend` entry indicates that the referenced object cannot be dropped without also dropping the dependent object. However, there are several subflavors identified by `deptype`:

**SHARED_DEPENDENCY_OWNER (o)**

The referenced object (which must be a role) is the owner of the dependent object.

**SHARED_DEPENDENCY_ACL (a)**

The referenced object (which must be a role) is mentioned in the ACL (access control list, i.e., privileges list) of the dependent object. (A `SHARED_DEPENDENCY_ACL` entry is not made for the owner of the object, since the owner will have a `SHARED_DEPENDENCY_OWNER` entry anyway.)

**SHARED_DEPENDENCY_PIN (p)**

There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by `initdb`. The columns for the dependent object contain zeroes.

Other dependency flavors might be needed in future. Note in particular that the current definition only supports roles as referenced objects.

#### 44.32. pg_shdescription

The catalog `pg_shdescription` stores optional descriptions (comments) for shared database objects. Descriptions can be manipulated with the `COMMENT` command and viewed with `psql`'s `	d` commands.

See also `pg_description`, which performs a similar function for descriptions involving objects within a single database.
Unlike most system catalogs, `pg_shdescription` is shared across all databases of a cluster: there is only one copy of `pg_shdescription` per cluster, not one per database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objoid</td>
<td>oid</td>
<td>any OID column</td>
<td>The OID of the object this description pertains to</td>
</tr>
<tr>
<td>classoid</td>
<td>oid</td>
<td><code>pg_class.oid</code></td>
<td>The OID of the system catalog this object appears in</td>
</tr>
<tr>
<td>description</td>
<td>text</td>
<td></td>
<td>Arbitrary text that serves as the description of this object</td>
</tr>
</tbody>
</table>

**44.33. pg_statistic**

The catalog `pg_statistic` stores statistical data about the contents of the database. Entries are created by `ANALYZE` and subsequently used by the query planner. There is one entry for each table column that has been analyzed. Note that all the statistical data is inherently approximate, even assuming that it is up-to-date.

`pg_statistic` also stores statistical data about the values of index expressions. These are described as if they were actual data columns; in particular, `starelid` references the index. No entry is made for an ordinary non-expression index column, however, since it would be redundant with the entry for the underlying table column.

Since different kinds of statistics might be appropriate for different kinds of data, `pg_statistic` is designed not to assume very much about what sort of statistics it stores. Only extremely general statistics (such as nullness) are given dedicated columns in `pg_statistic`. Everything else is stored in “slots”, which are groups of associated columns whose content is identified by a code number in one of the slot’s columns. For more information see `src/include/catalog/pg_statistic.h`.

`pg_statistic` should not be readable by the public, since even statistical information about a table’s contents might be considered sensitive. (Example: minimum and maximum values of a salary column might be quite interesting.) `pg_stats` is a publicly readable view on `pg_statistic` that only exposes information about those tables that are readable by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>starelid</td>
<td>oid</td>
<td><code>pg_class.oid</code></td>
<td>The table or index that the described column belongs to</td>
</tr>
<tr>
<td>staattnum</td>
<td>int2</td>
<td><code>pg_attribute.attnum</code></td>
<td>The number of the described column</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>stanullfrac</td>
<td>float4</td>
<td></td>
<td>The fraction of the column’s entries that are null</td>
</tr>
<tr>
<td>stawidth</td>
<td>int4</td>
<td></td>
<td>The average stored width, in bytes, of nonnull entries</td>
</tr>
<tr>
<td>stadistinct</td>
<td>float4</td>
<td></td>
<td>The number of distinct nonnull data values in the column. A value greater than zero is the actual number of distinct values. A value less than zero is the negative of a fraction of the number of rows in the table (for example, a column in which values appear about twice on the average could be represented by <code>stadistinct = -0.5</code>). A zero value means the number of distinct values is unknown</td>
</tr>
<tr>
<td>stakindN</td>
<td>int2</td>
<td></td>
<td>A code number indicating the kind of statistics stored in the Nth “slot” of the <code>pg_statistic</code> row</td>
</tr>
<tr>
<td>staopN</td>
<td>oid</td>
<td><code>pg_operator.oid</code></td>
<td>An operator used to derive the statistics stored in the Nth “slot”. For example, a histogram slot would show the <code>&lt;</code> operator that defines the sort order of the data</td>
</tr>
<tr>
<td>stanumbersN</td>
<td>float4[]</td>
<td></td>
<td>Numerical statistics of the appropriate kind for the Nth “slot”, or NULL if the slot kind does not involve numerical values</td>
</tr>
</tbody>
</table>
### 44.34. `pg_tablespace`

The catalog `pg_tablespace` stores information about the available tablespaces. Tables can be placed in particular tablespaces to aid administration of disk layout.

Unlike most system catalogs, `pg_tablespace` is shared across all databases of a cluster: there is only one copy of `pg_tablespace` per cluster, not one per database.

**Table 44-34. `pg_tablespace` Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spcname</td>
<td>name</td>
<td></td>
<td>Tablespace name</td>
</tr>
<tr>
<td>spcowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the tablespace, usually the user who created it</td>
</tr>
<tr>
<td>spclocation</td>
<td>text</td>
<td></td>
<td>Location (directory path) of the tablespace</td>
</tr>
<tr>
<td>spcacl</td>
<td>aclitem[]</td>
<td></td>
<td>Access privileges; see <code>GRANT</code> and <code>REVOKE</code> for details</td>
</tr>
</tbody>
</table>

### 44.35. `pg_trigger`

The catalog `pg_trigger` stores triggers on tables. See `CREATE TRIGGER` for more information.

**Table 44-35. `pg_trigger` Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The table this trigger is on</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>tgname</td>
<td>name</td>
<td></td>
<td>Trigger name (must be unique among triggers of same table)</td>
</tr>
<tr>
<td>tgfoid</td>
<td>oid</td>
<td>pg_proc.oid</td>
<td>The function to be called</td>
</tr>
<tr>
<td>tgtype</td>
<td>int2</td>
<td></td>
<td>Bit mask identifying trigger conditions</td>
</tr>
<tr>
<td>tgenabled</td>
<td>char</td>
<td></td>
<td>Controls in which session_replication_role modes the trigger fires.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$O$ = trigger fires in “origin” and “local” modes, $D$ = trigger is disabled,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$R$ = trigger fires in “replica” mode, $A$ = trigger fires always.</td>
</tr>
<tr>
<td>tgisconstraint</td>
<td>bool</td>
<td></td>
<td>True if trigger is a “constraint trigger”</td>
</tr>
<tr>
<td>tgconstname</td>
<td>name</td>
<td></td>
<td>Constraint name, if a constraint trigger</td>
</tr>
<tr>
<td>tgconstrrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>The table referenced by a referential integrity constraint</td>
</tr>
<tr>
<td>tgconstraint</td>
<td>oid</td>
<td>pg_constraint.oid</td>
<td>The pg_constraint entry owning the trigger, if any</td>
</tr>
<tr>
<td>tgdeferrable</td>
<td>bool</td>
<td></td>
<td>True if constraint trigger is deferrable</td>
</tr>
<tr>
<td>tginitdeferred</td>
<td>bool</td>
<td></td>
<td>True if constraint trigger is initially deferred</td>
</tr>
<tr>
<td>tgnargs</td>
<td>int2</td>
<td></td>
<td>Number of argument strings passed to trigger function</td>
</tr>
<tr>
<td>tgattr</td>
<td>int2vector</td>
<td></td>
<td>Currently unused</td>
</tr>
<tr>
<td>tgargs</td>
<td>bytea</td>
<td></td>
<td>Argument strings to pass to trigger, each NULL-terminated</td>
</tr>
</tbody>
</table>

**Note:** When `tgconstraint` is nonzero, `tgisconstraint` must be true, and `tgconstname`, `tgconstrrelid`, `tgdeferrable`, `tginitdeferred` are redundant with the referenced `pg_constraint` entry. The reason we keep these fields is that we support “stand-alone” constraint triggers with no corresponding `pg_constraint` entry.

**Note:** `pg_class.reltriggers` needs to agree with the number of triggers found in this table.
Chapter 44. System Catalogs

each relation.

### 44.36. pg_ts_config

The `pg_ts_config` catalog contains entries representing text search configurations. A configuration specifies a particular text search parser and a list of dictionaries to use for each of the parser’s output token types. The parser is shown in the `pg_ts_config` entry, but the token-to-dictionary mapping is defined by subsidiary entries in `pg_ts_config_map`.

PostgreSQL’s text search features are described at length in Chapter 12.

**Table 44-36. pg_ts_config Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfgname</td>
<td>name</td>
<td></td>
<td>Text search configuration name</td>
</tr>
<tr>
<td>cfgnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this configuration</td>
</tr>
<tr>
<td>cfgowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the configuration</td>
</tr>
<tr>
<td>cfgparser</td>
<td>oid</td>
<td>pg_ts_parser.oid</td>
<td>The OID of the text search parser for this configuration</td>
</tr>
</tbody>
</table>

### 44.37. pg_ts_config_map

The `pg_ts_config_map` catalog contains entries showing which text search dictionaries should be consulted, and in what order, for each output token type of each text search configuration’s parser.

PostgreSQL’s text search features are described at length in Chapter 12.

**Table 44-37. pg_ts_config_map Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapcfg</td>
<td>oid</td>
<td>pg_ts_config.oid</td>
<td>The OID of the <code>pg_ts_config</code> entry owning this map entry</td>
</tr>
<tr>
<td>maptokentype</td>
<td>integer</td>
<td></td>
<td>A token type emitted by the configuration’s parser</td>
</tr>
<tr>
<td>mapseqno</td>
<td>integer</td>
<td></td>
<td>Order in which to consult this entry (lower mapseqnos first)</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapdict</td>
<td>oid</td>
<td>pg_ts_dict.oid</td>
<td>The OID of the text search dictionary to consult</td>
</tr>
</tbody>
</table>

44.38. pg_ts_dict

The pg_ts_dict catalog contains entries defining text search dictionaries. A dictionary depends on a text search template, which specifies all the implementation functions needed; the dictionary itself provides values for the user-settable parameters supported by the template. This division of labor allows dictionaries to be created by unprivileged users. The parameters are specified by a text string dictinitoption, whose format and meaning vary depending on the template.

PostgreSQL’s text search features are described at length in Chapter 12.

Table 44-38. pg_ts_dict Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictname</td>
<td>name</td>
<td></td>
<td>Text search dictionary name</td>
</tr>
<tr>
<td>dictnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this dictionary</td>
</tr>
<tr>
<td>dictowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the dictionary</td>
</tr>
<tr>
<td>dicttemplate</td>
<td>oid</td>
<td>pg_ts_template.oid</td>
<td>The OID of the text search template for this dictionary</td>
</tr>
<tr>
<td>dictinitoption</td>
<td>text</td>
<td></td>
<td>Initialization option string for the template</td>
</tr>
</tbody>
</table>

44.39. pg_ts_parser

The pg_ts_parser catalog contains entries defining text search parsers. A parser is responsible for splitting input text into lexemes and assigning a token type to each lexeme. Since a parser must be implemented by C-language-level functions, creation of new parsers is restricted to database superusers.

PostgreSQL’s text search features are described at length in Chapter 12.

Table 44-39. pg_ts_parser Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prsname</td>
<td>name</td>
<td></td>
<td>Text search parser name</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prsnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this parser</td>
</tr>
<tr>
<td>prsstart</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>OID of the parser’s startup function</td>
</tr>
<tr>
<td>prstoken</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>OID of the parser’s next-token function</td>
</tr>
<tr>
<td>prsend</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>OID of the parser’s shutdown function</td>
</tr>
<tr>
<td>prsheadline</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>OID of the parser’s headline function</td>
</tr>
<tr>
<td>prslextype</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>OID of the parser’s lextype function</td>
</tr>
</tbody>
</table>

44.40. pg_ts_template

The pg_ts_template catalog contains entries defining text search templates. A template is the implementation skeleton for a class of text search dictionaries. Since a template must be implemented by C-language-level functions, creation of new templates is restricted to database superusers.

PostgreSQL’s text search features are described at length in Chapter 12.

Table 44-40. pg_ts_template Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmplname</td>
<td>name</td>
<td></td>
<td>Text search template name</td>
</tr>
<tr>
<td>tmplnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this template</td>
</tr>
<tr>
<td>tmplinit</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>OID of the template’s initialization function</td>
</tr>
<tr>
<td>tmpflexize</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>OID of the template’s lexize function</td>
</tr>
</tbody>
</table>

44.41. pg_type

The catalog pg_type stores information about data types. Base types (scalar types) are created with CREATE TYPE, and domains with CREATE DOMAIN. A composite type is automatically created for each table in the database, to represent the row structure of the table. It is also possible to create composite types with CREATE TYPE AS.

Table 44-41. pg_type Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>typname</td>
<td>name</td>
<td></td>
<td>Data type name</td>
</tr>
<tr>
<td>typnamespace</td>
<td>oid</td>
<td>pg_namespace.oid</td>
<td>The OID of the namespace that contains this type</td>
</tr>
<tr>
<td>typowner</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>Owner of the type</td>
</tr>
<tr>
<td>typlen</td>
<td>int2</td>
<td></td>
<td>For a fixed-size type, typlen is the number of bytes in the internal representation of the type. But for a variable-length type, typlen is negative. -1 indicates a “varlena” type (one that has a length word). -2 indicates a null-terminated C string.</td>
</tr>
<tr>
<td>typbyval</td>
<td>bool</td>
<td></td>
<td>typbyval determines whether internal routines pass a value of this type by value or by reference. typbyval had better be false if typlen is not 1, 2, or 4 (or 8 on machines where Datum is 8 bytes). Variable-length types are always passed by reference. Note that typbyval can be false even if the length would allow pass-by-value; this is currently true for type float4, for example.</td>
</tr>
<tr>
<td>typtype</td>
<td>char</td>
<td></td>
<td>typtype is b for a base type, c for a composite type (e.g., a table’s row type), d for a domain, e for an enum type, or p for a pseudo-type. See also typrelid and typbasetype.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>typisdefined</td>
<td>bool</td>
<td></td>
<td>True if the type is defined, false if this is a placeholder entry for a not-yet-defined type. When typisdefined is false, nothing except the type name, namespace, and OID can be relied on.</td>
</tr>
<tr>
<td>typdelim</td>
<td>char</td>
<td></td>
<td>Character that separates two values of this type when parsing array input. Note that the delimiter is associated with the array element data type, not the array data type.</td>
</tr>
<tr>
<td>typrelid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>If this is a composite type (see typtype), then this column points to the pg_class entry that defines the corresponding table. (For a free-standing composite type, the pg_class entry doesn’t really represent a table, but it is needed anyway for the type’s pg_attribute entries to link to.) Zero for non-composite types</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>typelem</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>If typelem is not 0 then it identifies another row in pg_type. The current type can then be subscripted like an array yielding values of type typelem. A “true” array type is variable length (typlen = -1), but some fixed-length (typlen &gt; 0) types also have nonzero typelem, for example name and point. If a fixed-length type has a typelem then its internal representation must be some number of values of the typelem data type with no other data. Variable-length array types have a header defined by the array subroutines.</td>
</tr>
<tr>
<td>typarray</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>If typarray is not 0 then it identifies another row in pg_type, which is the “true” array type having this type as element.</td>
</tr>
<tr>
<td>typinput</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Input conversion function (text format)</td>
</tr>
<tr>
<td>typoutput</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Output conversion function (text format)</td>
</tr>
<tr>
<td>typreceive</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Input conversion function (binary format), or 0 if none</td>
</tr>
<tr>
<td>typsend</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Output conversion function (binary format), or 0 if none</td>
</tr>
<tr>
<td>typmodin</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Type modifier input function, or 0 if type does not support modifiers</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>---------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>typmodout</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Type modifier output function, or 0 to use the standard format</td>
</tr>
<tr>
<td>typanalyze</td>
<td>regproc</td>
<td>pg_proc.oid</td>
<td>Custom ANALYZE function, or 0 to use the standard function</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| typalign | char  |            | typalign is the alignment required when storing a value of this type. It applies to storage on disk as well as most representations of the value inside PostgreSQL. When multiple values are stored consecutively, such as in the representation of a complete row on disk, padding is inserted before a datum of this type so that it begins on the specified boundary. The alignment reference is the beginning of the first datum in the sequence. Possible values are:  
  • \( c = \text{char} \) alignment, i.e., no alignment needed.  
  • \( s = \text{short} \) alignment (2 bytes on most machines).  
  • \( i = \text{int} \) alignment (4 bytes on most machines).  
  • \( d = \text{double} \) alignment (8 bytes on many machines, but by no means all).  

**Note:** For types used in system tables, it is critical that the size and alignment defined in `pg_type` agree with the way that the compiler will lay out the column in a structure representing a table row.
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>typstorage</td>
<td>char</td>
<td></td>
<td>typstorage tells for varlena types (those with <code>typlen = -1</code>) if the type is prepared for toasting and what the default strategy for attributes of this type should be. Possible values are · p: Value must always be stored plain. · e: Value can be stored in a “secondary” relation (if relation has one, see <code>pg_class.reltoastrelid</code>). · m: Value can be stored compressed inline. · x: Value can be stored compressed inline or stored in “secondary” storage. Note that m columns can also be moved out to secondary storage, but only as a last resort (e and x columns are moved first).</td>
</tr>
<tr>
<td>typnull</td>
<td>bool</td>
<td></td>
<td>typnotnull represents a not-null constraint on a type. Used for domains only.</td>
</tr>
<tr>
<td>typbasetype</td>
<td>oid</td>
<td>pg_type.oid</td>
<td>If this is a domain (see <code>typtype</code>), then <code>typbasetype</code> identifies the type that this one is based on. Zero if this type is not a domain</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>typtypmod</td>
<td>int4</td>
<td></td>
<td>Domains use <code>typtypmod</code> to record the <code>typmod</code> to be applied to their base type (-1 if base type does not use a <code>typmod</code>). -1 if this type is not a domain</td>
</tr>
<tr>
<td>typndims</td>
<td>int4</td>
<td></td>
<td><code>typndims</code> is the number of array dimensions for a domain that is an array (that is, <code>typbasetype</code> is an array type; the domain’s <code>typelem</code> will match the base type’s <code>typelem</code>). Zero for types other than domains over array types</td>
</tr>
<tr>
<td>typdefaultbin</td>
<td>text</td>
<td></td>
<td>If <code>typdefaultbin</code> is not null, it is the <code>nodeToString()</code> representation of a default expression for the type. This is only used for domains</td>
</tr>
<tr>
<td>typdefault</td>
<td>text</td>
<td></td>
<td><code>typdefault</code> is null if the type has no associated default value. If <code>typdefaultbin</code> is not null, <code>typdefault</code> must contain a human-readable version of the default expression represented by <code>typdefaultbin</code>. If <code>typdefaultbin</code> is null and <code>typdefault</code> is not, then <code>typdefault</code> is the external representation of the type’s default value, which might be fed to the type’s input converter to produce a constant</td>
</tr>
</tbody>
</table>
44.42. System Views

In addition to the system catalogs, PostgreSQL provides a number of built-in views. Some system views provide convenient access to some commonly used queries on the system catalogs. Other views provide access to internal server state.

The information schema (Chapter 33) provides an alternative set of views which overlap the functionality of the system views. Since the information schema is SQL-standard whereas the views described here are PostgreSQL-specific, it’s usually better to use the information schema if it provides all the information you need.

Table 44-42 lists the system views described here. More detailed documentation of each view follows below. There are some additional views that provide access to the results of the statistics collector; they are described in Table 26-1.

Except where noted, all the views described here are read-only.

Table 44-42. System Views

<table>
<thead>
<tr>
<th>View Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_cursors</td>
<td>open cursors</td>
</tr>
<tr>
<td>pg_group</td>
<td>groups of database users</td>
</tr>
<tr>
<td>pg_indexes</td>
<td>indexes</td>
</tr>
<tr>
<td>pg_locks</td>
<td>currently held locks</td>
</tr>
<tr>
<td>pg_prepared_statements</td>
<td>prepared statements</td>
</tr>
<tr>
<td>pg_prepared_xacts</td>
<td>prepared transactions</td>
</tr>
<tr>
<td>pg_roles</td>
<td>database roles</td>
</tr>
<tr>
<td>pg_rules</td>
<td>rules</td>
</tr>
<tr>
<td>pg_settings</td>
<td>parameter settings</td>
</tr>
<tr>
<td>pg_shadow</td>
<td>database users</td>
</tr>
<tr>
<td>pg_stats</td>
<td>planner statistics</td>
</tr>
<tr>
<td>pg_tables</td>
<td>tables</td>
</tr>
<tr>
<td>pg_timezone_abbrevs</td>
<td>time zone abbreviations</td>
</tr>
<tr>
<td>pg_timezone_names</td>
<td>time zone names</td>
</tr>
<tr>
<td>pg_user</td>
<td>database users</td>
</tr>
<tr>
<td>pg_views</td>
<td>views</td>
</tr>
</tbody>
</table>

44.43. pg_cursors

The pg_cursors view lists the cursors that are currently available. Cursors can be defined in several ways:

- via the DECLARE statement in SQL
- via the Bind message in the frontend/backend protocol, as described in Section 45.2.3
- via the Server Programming Interface (SPI), as described in Section 42.1
The \texttt{pg\_cursors} view displays cursors created by any of these means. Cursors only exist for the duration of the transaction that defines them, unless they have been declared \texttt{WITH HOLD}. Therefore non-holdable cursors are only present in the view until the end of their creating transaction.

\textbf{Note:} Cursors are used internally to implement some of the components of PostgreSQL, such as procedural languages. Therefore, the \texttt{pg\_cursors} view might include cursors that have not been explicitly created by the user.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Name} & \textbf{Type} & \textbf{Description} \\
\hline
name & text & The name of the cursor \\
statement & text & The verbatim query string submitted to declare this cursor \\
is\_holdable & boolean & \texttt{true} if the cursor is holdable (that is, it can be accessed after the transaction that declared the cursor has committed); \texttt{false} otherwise \\
is\_binary & boolean & \texttt{true} if the cursor was declared \texttt{BINARY}; \texttt{false} otherwise \\
is\_scrollable & boolean & \texttt{true} if the cursor is scrollable (that is, it allows rows to be retrieved in a nonsequential manner); \texttt{false} otherwise \\
creation\_time & timestamptz & The time at which the cursor was declared \\
\hline
\end{tabular}
\caption{\texttt{pg\_cursors} Columns}
\end{table}

The \texttt{pg\_cursors} view is read only.

\section*{44.44. pg\_group}

The view \texttt{pg\_group} exists for backwards compatibility: it emulates a catalog that existed in PostgreSQL before version 8.1. It shows the names and members of all roles that are marked as not \texttt{rol\_can\_login}, which is an approximation to the set of roles that are being used as groups.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|}
\hline
\textbf{Name} & \textbf{Type} & \textbf{References} & \textbf{Description} \\
\hline
groname & name & \texttt{pg\_authid.rolname} & Name of the group \\
groSID & oid & \texttt{pg\_authid.oid} & ID of this group \\
grolist & oid[] & \texttt{pg\_authid.oid} & An array containing the IDs of the roles in this group \\
\hline
\end{tabular}
\caption{\texttt{pg\_group} Columns}
\end{table}
Chapter 44. System Catalogs

44.45. pg_indexes

The view pg_indexes provides access to useful information about each index in the database.

Table 44-45. pg_indexes Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schemaname</td>
<td>name</td>
<td>pg_namespace.nspname</td>
<td>Name of schema containing table and index</td>
</tr>
<tr>
<td>tablename</td>
<td>name</td>
<td>pg_class.relname</td>
<td>Name of table the index is for</td>
</tr>
<tr>
<td>indexname</td>
<td>name</td>
<td>pg_class.relname</td>
<td>Name of index</td>
</tr>
<tr>
<td>tablespace</td>
<td>name</td>
<td>pg_tablespace.spcname</td>
<td>Name of tablespace containing index (NULL if default for database)</td>
</tr>
<tr>
<td>indexdef</td>
<td>text</td>
<td></td>
<td>Index definition (a reconstructed CREATE INDEX command)</td>
</tr>
</tbody>
</table>

44.46. pg_locks

The view pg_locks provides access to information about the locks held by open transactions within the database server. See Chapter 13 for more discussion of locking.

pg_locks contains one row per active lockable object, requested lock mode, and relevant transaction. Thus, the same lockable object might appear many times, if multiple transactions are holding or waiting for locks on it. However, an object that currently has no locks on it will not appear at all.

There are several distinct types of lockable objects: whole relations (e.g., tables), individual pages of relations, individual tuples of relations, transaction IDs (both virtual and permanent IDs), and general database objects (identified by class OID and object OID, in the same way as in pg_description or pg_depend). Also, the right to extend a relation is represented as a separate lockable object.

Table 44-46. pg_locks Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locktype</td>
<td>text</td>
<td></td>
<td>type of the lockable object: relation, extend, page, tuple, transactionid, virtualxid, object, userlock, or advisory</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>database</td>
<td>oid</td>
<td>pg_database.oid</td>
<td>OID of the database in which the object exists, or zero if the object is a shared object, or NULL if the object is a transaction ID</td>
</tr>
<tr>
<td>relation</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>OID of the relation, or NULL if the object is not a relation or part of a relation</td>
</tr>
<tr>
<td>page</td>
<td>integer</td>
<td></td>
<td>Page number within the relation, or NULL if the object is not a tuple or relation page</td>
</tr>
<tr>
<td>tuple</td>
<td>smallint</td>
<td></td>
<td>Tuple number within the page, or NULL if the object is not a tuple</td>
</tr>
<tr>
<td>virtualxid</td>
<td>text</td>
<td></td>
<td>Virtual ID of a transaction, or NULL if the object is not a virtual transaction ID</td>
</tr>
<tr>
<td>transactionid</td>
<td>xid</td>
<td></td>
<td>ID of a transaction, or NULL if the object is not a transaction ID</td>
</tr>
<tr>
<td>classid</td>
<td>oid</td>
<td>pg_class.oid</td>
<td>OID of the system catalog containing the object, or NULL if the object is not a general database object</td>
</tr>
<tr>
<td>objid</td>
<td>oid</td>
<td>any OID column</td>
<td>OID of the object within its system catalog, or NULL if the object is not a general database object</td>
</tr>
<tr>
<td>objsubid</td>
<td>smallint</td>
<td></td>
<td>For a table column, this is the column number (the classid and objid refer to the table itself). For all other object types, this column is zero. NULL if the object is not a general database object</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>virtualtransaction</td>
<td>text</td>
<td></td>
<td>Virtual ID of the transaction that is holding or awaiting this lock</td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td></td>
<td>Process ID of the server process holding or awaiting this lock. NULL if the lock is held by a prepared transaction</td>
</tr>
<tr>
<td>mode</td>
<td>text</td>
<td></td>
<td>Name of the lock mode held or desired by this process (see Section 13.3.1)</td>
</tr>
<tr>
<td>granted</td>
<td>boolean</td>
<td></td>
<td>True if lock is held, false if lock is awaited</td>
</tr>
</tbody>
</table>

granted is true in a row representing a lock held by the indicated transaction. False indicates that this transaction is currently waiting to acquire this lock, which implies that some other transaction is holding a conflicting lock mode on the same lockable object. The waiting transaction will sleep until the other lock is released (or a deadlock situation is detected). A single transaction can be waiting to acquire at most one lock at a time.

Every transaction holds an exclusive lock on its virtual transaction ID for its entire duration. If a permanent ID is assigned to the transaction (which normally happens only if the transaction changes the state of the database), it also holds an exclusive lock on its permanent transaction ID until it ends. When one transaction finds it necessary to wait specifically for another transaction, it does so by attempting to acquire share lock on the other transaction ID (either virtual or permanent ID depending on the situation). That will succeed only when the other transaction terminates and releases its locks.

Although tuples are a lockable type of object, information about row-level locks is stored on disk, not in memory, and therefore row-level locks normally do not appear in this view. If a transaction is waiting for a row-level lock, it will usually appear in the view as waiting for the permanent transaction ID of the current holder of that row lock.

Advisory locks can be acquired on keys consisting of either a single bigint value or two integer values. A bigint key is displayed with its high-order half in the classid column, its low-order half in the objid column, and objsubid equal to 1. Integer keys are displayed with the first key in the classid column, the second key in the objid column, and objsubid equal to 2. The actual meaning of the keys is up to the user. Advisory locks are local to each database, so the database column is meaningful for an advisory lock.

When the pg_locks view is accessed, the internal lock manager data structures are momentarily locked, and a copy is made for the view to display. This ensures that the view produces a consistent set of results, while not blocking normal lock manager operations longer than necessary. Nonetheless there could be some impact on database performance if this view is frequently accessed.

pg_locks provides a global view of all locks in the database cluster, not only those relevant to the current database. Although its relation column can be joined against pg_class.oid to identify locked relations, this will only work correctly for relations in the current database (those for which the database column is either the current database’s OID or zero).

The pid column can be joined to the procpid column of the pg_stat_activity view to get...
more information on the session holding or waiting to hold each lock. Also, if you are using prepared transactions, the `transaction` column can be joined to the `transaction` column of the `pg_prepared_xacts` view to get more information on prepared transactions that hold locks. (A prepared transaction can never be waiting for a lock, but it continues to hold the locks it acquired while running.)

### 44.47. pg_prepared_statements

The `pg_prepared_statements` view displays all the prepared statements that are available in the current session. See `PREPARE` for more information about prepared statements.

`pg_prepared_statements` contains one row for each prepared statement. Rows are added to the view when a new prepared statement is created and removed when a prepared statement is released (for example, via the `DEALLOCATE` command).

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>text</td>
<td>The identifier of the prepared statement</td>
</tr>
<tr>
<td>statement</td>
<td>text</td>
<td>The query string submitted by the client to create this prepared statement. For prepared statements created via SQL, this is the <code>PREPARE</code> statement submitted by the client. For prepared statements created via the frontend/backend protocol, this is the text of the prepared statement itself</td>
</tr>
<tr>
<td>prepare_time</td>
<td>timestamptz</td>
<td>The time at which the prepared statement was created</td>
</tr>
<tr>
<td>parameter_types</td>
<td>regtype[]</td>
<td>The expected parameter types for the prepared statement in the form of an array of <code>regtype</code>. The OID corresponding to an element of this array can be obtained by casting the <code>regtype</code> value to <code>oid</code></td>
</tr>
<tr>
<td>from_sql</td>
<td>boolean</td>
<td><code>true</code> if the prepared statement was created via the <code>PREPARE</code> SQL statement; <code>false</code> if the statement was prepared via the frontend/backend protocol</td>
</tr>
</tbody>
</table>

The `pg_prepared_statements` view is read only.
44.48. *pg_prepared_xacts*

The view `pg_prepared_xacts` displays information about transactions that are currently prepared for two-phase commit (see `PREPARE TRANSACTION` for details).

`pg_prepared_xacts` contains one row per prepared transaction. An entry is removed when the transaction is committed or rolled back.

**Table 44-48. pg_prepared_xacts Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction</td>
<td>xid</td>
<td></td>
<td>Numeric transaction identifier of the prepared transaction</td>
</tr>
<tr>
<td>gid</td>
<td>text</td>
<td></td>
<td>Global transaction identifier that was assigned to the transaction</td>
</tr>
<tr>
<td>prepared</td>
<td>timestamp with time zone</td>
<td></td>
<td>Time at which the transaction was prepared for commit</td>
</tr>
<tr>
<td>owner</td>
<td>name</td>
<td><code>pg_authid.rolname</code></td>
<td>Name of the user that executed the transaction</td>
</tr>
<tr>
<td>database</td>
<td>name</td>
<td><code>pg_database.datname</code></td>
<td>Name of the database in which the transaction was executed</td>
</tr>
</tbody>
</table>

When the `pg_prepared_xacts` view is accessed, the internal transaction manager data structures are momentarily locked, and a copy is made for the view to display. This ensures that the view produces a consistent set of results, while not blocking normal operations longer than necessary. Nonetheless there could be some impact on database performance if this view is frequently accessed.

44.49. *pg_roles*

The view `pg_roles` provides access to information about database roles. This is simply a publicly readable view of `pg_authid` that blanks out the password field.

This view explicitly exposes the OID column of the underlying table, since that is needed to do joins to other catalogs.

**Table 44-49. pg_roles Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolname</td>
<td>name</td>
<td></td>
<td>Role name</td>
</tr>
<tr>
<td>rolsuper</td>
<td>bool</td>
<td></td>
<td>Role has superuser privileges</td>
</tr>
</tbody>
</table>
### Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolinherit</td>
<td>bool</td>
<td></td>
<td>Role automatically inherits privileges of roles it is a member of</td>
</tr>
<tr>
<td>rolcreaterole</td>
<td>bool</td>
<td></td>
<td>Role can create more roles</td>
</tr>
<tr>
<td>rolcreatedb</td>
<td>bool</td>
<td></td>
<td>Role can create databases</td>
</tr>
<tr>
<td>rolcatupdate</td>
<td>bool</td>
<td></td>
<td>Role can update system catalogs directly. (Even a superuser cannot do this unless this column is true.)</td>
</tr>
<tr>
<td>rolcanlogin</td>
<td>bool</td>
<td></td>
<td>Role can log in. That is, this role can be given as the initial session authorization identifier</td>
</tr>
<tr>
<td>rolconnlimit</td>
<td>int4</td>
<td></td>
<td>For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit</td>
</tr>
<tr>
<td>rolpassword</td>
<td>text</td>
<td></td>
<td>Not the password (always reads as ********)</td>
</tr>
<tr>
<td>rolvaliduntil</td>
<td>timestamp</td>
<td></td>
<td>Password expiry time (only used for password authentication); NULL if no expiration</td>
</tr>
<tr>
<td>rolconfig</td>
<td>text[]</td>
<td></td>
<td>Session defaults for run-time configuration variables</td>
</tr>
<tr>
<td>oid</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>ID of role</td>
</tr>
</tbody>
</table>

#### 44.50. pg_rules

The view `pg_rules` provides access to useful information about query rewrite rules.

**Table 44.50. pg_rules Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schemaname</td>
<td>name</td>
<td>pg_namespace.nspname</td>
<td>Name of schema containing table</td>
</tr>
</tbody>
</table>
Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tablename</td>
<td>name</td>
<td>pg_class.relname</td>
<td>Name of table the rule is for</td>
</tr>
<tr>
<td>rulename</td>
<td>name</td>
<td>pg_rewrite.rulename</td>
<td>Name of rule</td>
</tr>
<tr>
<td>definition</td>
<td>text</td>
<td></td>
<td>Rule definition (a reconstructed creation command)</td>
</tr>
</tbody>
</table>

The `pg_rules` view excludes the ON SELECT rules of views; those can be seen in `pg_views`.

### 44.51. pg_settings

The view `pg_settings` provides access to run-time parameters of the server. It is essentially an alternative interface to the `SHOW` and `SET` commands. It also provides access to some facts about each parameter that are not directly available from `SHOW`, such as minimum and maximum values.

**Table 44-51. pg_settings Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>text</td>
<td>Run-time configuration parameter name</td>
</tr>
<tr>
<td>setting</td>
<td>text</td>
<td>Current value of the parameter</td>
</tr>
<tr>
<td>unit</td>
<td>text</td>
<td>Implicit unit of the parameter</td>
</tr>
<tr>
<td>category</td>
<td>text</td>
<td>Logical group of the parameter</td>
</tr>
<tr>
<td>short_desc</td>
<td>text</td>
<td>A brief description of the parameter</td>
</tr>
<tr>
<td>extra_desc</td>
<td>text</td>
<td>Additional, more detailed, information about the parameter</td>
</tr>
<tr>
<td>context</td>
<td>text</td>
<td>Context required to set the parameter’s value</td>
</tr>
<tr>
<td>vartype</td>
<td>text</td>
<td>Parameter type (bool, integer, real, or string)</td>
</tr>
<tr>
<td>source</td>
<td>text</td>
<td>Source of the current parameter value</td>
</tr>
<tr>
<td>min_val</td>
<td>text</td>
<td>Minimum allowed value of the parameter (NULL for non-numeric values)</td>
</tr>
<tr>
<td>max_val</td>
<td>text</td>
<td>Maximum allowed value of the parameter (NULL for non-numeric values)</td>
</tr>
</tbody>
</table>

The `pg_settings` view cannot be inserted into or deleted from, but it can be updated. An `UPDATE` applied to a row of `pg_settings` is equivalent to executing the `SET` command on that named parameter. The change only affects the value used by the current session. If an `UPDATE` is issued within a transaction that is later aborted, the effects of the `UPDATE` command disappear when the transaction
is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of
the session, unless overridden by another `UPDATE` or `SET`.

### 44.52. pg_shadow

The view `pg_shadow` exists for backwards compatibility: it emulates a catalog that existed in Post-
greSQL before version 8.1. It shows properties of all roles that are marked as `rolcanlogin`.

The name stems from the fact that this table should not be readable by the public since it contains
passwords. `pg_user` is a publicly readable view on `pg_shadow` that blanks out the password field.

**Table 44-52. pg_shadow Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>usename</td>
<td>name</td>
<td><code>pg_authid.rolname</code></td>
<td>User name</td>
</tr>
<tr>
<td>usesysid</td>
<td>oid</td>
<td><code>pg_authid.oid</code></td>
<td>ID of this user</td>
</tr>
<tr>
<td>usecreatedb</td>
<td>bool</td>
<td></td>
<td>User can create databases</td>
</tr>
<tr>
<td>usesuper</td>
<td>bool</td>
<td></td>
<td>User is a superuser</td>
</tr>
<tr>
<td>usecatupd</td>
<td>bool</td>
<td></td>
<td>User can update system catalogs. (Even a superuser cannot do this unless this column is true.)</td>
</tr>
<tr>
<td>passwd</td>
<td>text</td>
<td></td>
<td>Password (possibly encrypted)</td>
</tr>
<tr>
<td>valuntil</td>
<td>abstime</td>
<td></td>
<td>Password expiry time (only used for password authentication)</td>
</tr>
<tr>
<td>useconfig</td>
<td>text[]</td>
<td></td>
<td>Session defaults for run-time configuration variables</td>
</tr>
</tbody>
</table>

### 44.53. pg_stats

The view `pg_stats` provides access to the information stored in the `pg_statistic` catalog. This view
allows access only to rows of `pg_statistic` that correspond to tables the user has permission to read, and therefore it is safe to allow public read access to this view.

`pg_stats` is also designed to present the information in a more readable format than the underlying
catalog — at the cost that its schema must be extended whenever new slot types are defined for
`pg_statistic`.

**Table 44-53. pg_stats Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>schemaname</td>
<td>name</td>
<td>pg_namespace.nsname</td>
<td>Name of schema containing table</td>
</tr>
<tr>
<td>tablename</td>
<td>name</td>
<td>pg_class.relname</td>
<td>Name of table</td>
</tr>
<tr>
<td>attname</td>
<td>name</td>
<td>pg_attribute.attname</td>
<td>Name of the column described by this row</td>
</tr>
<tr>
<td>null_frac</td>
<td>real</td>
<td></td>
<td>Fraction of column entries that are null</td>
</tr>
<tr>
<td>avg_width</td>
<td>integer</td>
<td></td>
<td>Average width in bytes of column’s entries</td>
</tr>
<tr>
<td>n_distinct</td>
<td>real</td>
<td></td>
<td>If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows. (The negated form is used when ANALYZE believes that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values.) For example, -1 indicates a unique column in which the number of distinct values is the same as the number of rows.</td>
</tr>
<tr>
<td>most_common_vals</td>
<td>anyarray</td>
<td></td>
<td>A list of the most common values in the column. (NULL if no values seem to be more common than any others.)</td>
</tr>
<tr>
<td>most_common_freqs</td>
<td>real[]</td>
<td></td>
<td>A list of the frequencies of the most common values, i.e., number of occurrences of each divided by total number of rows. (NULL when most_common_vals is.)</td>
</tr>
</tbody>
</table>
## Chapter 44. System Catalogs

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram_bounds</td>
<td>anyarray</td>
<td></td>
<td>A list of values that divide the column’s values into groups of approximately equal population. The values in most_common_vals, if present, are omitted from this histogram calculation. (This column is NULL if the column data type does not have a &lt; operator or if the most_common_vals list accounts for the entire population.)</td>
</tr>
<tr>
<td>correlation</td>
<td>real</td>
<td></td>
<td>Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk. (This column is NULL if the column data type does not have a &lt; operator.)</td>
</tr>
</tbody>
</table>

The maximum number of entries in the most_common_vals and histogram_bounds arrays can be set on a column-by-column basis using the ALTER TABLE SET STATISTICS command, or globally by setting the default_statistics_target run-time parameter.

### 44.54. pg_tables

The view pg_tables provides access to useful information about each table in the database.

#### Table 44-54. pg_tables Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schemaname</td>
<td>name</td>
<td>pg_namespace.nspname</td>
<td>Name of schema containing table</td>
</tr>
</tbody>
</table>
44.55. \texttt{pg\_timezone\_abbrevs}

The view \texttt{pg\_timezone\_abbrevs} provides a list of time zone abbreviations that are currently recognized by the datetime input routines. The contents of this view change when the timezone\_abbreviations run-time parameter is modified.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbrev</td>
<td>text</td>
<td>Time zone abbreviation</td>
</tr>
<tr>
<td>utc_offset</td>
<td>interval</td>
<td>Offset from UTC (positive means east of Greenwich)</td>
</tr>
<tr>
<td>is_dst</td>
<td>boolean</td>
<td>True if this is a daylight-savings abbreviation</td>
</tr>
</tbody>
</table>

44.56. \texttt{pg\_timezone\_names}

The view \texttt{pg\_timezone\_names} provides a list of time zone names that are recognized by \texttt{SET TIMEZONE}, along with their associated abbreviations, UTC offsets, and daylight-savings status. Unlike the abbreviations shown in \texttt{pg\_timezone\_abbrevs}, many of these names imply a set of daylight-savings transition date rules. Therefore, the associated information changes across local DST boundaries. The displayed information is computed based on the current value of \texttt{CURRENT\_TIMESTAMP}.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>text</td>
<td>Time zone name</td>
</tr>
<tr>
<td>abbrev</td>
<td>text</td>
<td>Time zone abbreviation</td>
</tr>
</tbody>
</table>
### 44.57. pg_user

The view `pg_user` provides access to information about database users. This is simply a publicly readable view of `pg_shadow` that blanks out the password field.

**Table 44-57. pg_user Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>usename</td>
<td>name</td>
<td>User name</td>
</tr>
<tr>
<td>usesysid</td>
<td>int4</td>
<td>User ID (arbitrary number used to reference this user)</td>
</tr>
<tr>
<td>usecreatedb</td>
<td>bool</td>
<td>User can create databases</td>
</tr>
<tr>
<td>usesuper</td>
<td>bool</td>
<td>User is a superuser</td>
</tr>
<tr>
<td>usecatupd</td>
<td>bool</td>
<td>User can update system catalogs. (Even a superuser cannot do this unless this column is true.)</td>
</tr>
<tr>
<td>passwd</td>
<td>text</td>
<td>Not the password (always reads as ********)</td>
</tr>
<tr>
<td>valuntil</td>
<td>abstime</td>
<td>Password expiry time (only used for password authentication)</td>
</tr>
<tr>
<td>useconfig</td>
<td>text[]</td>
<td>Session defaults for run-time configuration variables</td>
</tr>
</tbody>
</table>

### 44.58. pg_views

The view `pg_views` provides access to useful information about each view in the database.

**Table 44-58. pg_views Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schemaname</td>
<td>name</td>
<td><code>pg_namespace.nspname</code></td>
<td>Name of schema containing view</td>
</tr>
<tr>
<td>viewname</td>
<td>name</td>
<td><code>pg_class.relname</code></td>
<td>Name of view</td>
</tr>
<tr>
<td>viewowner</td>
<td>name</td>
<td><code>pg_authid.rolname</code></td>
<td>Name of view’s owner</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>References</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>definition</td>
<td>text</td>
<td></td>
<td>View definition (a reconstructed SELECT query)</td>
</tr>
</tbody>
</table>
Chapter 45. Frontend/Backend Protocol

PostgreSQL uses a message-based protocol for communication between frontends and backends (clients and servers). The protocol is supported over TCP/IP and also over Unix-domain sockets. Port number 5432 has been registered with IANA as the customary TCP port number for servers supporting this protocol, but in practice any non-privileged port number can be used.

This document describes version 3.0 of the protocol, implemented in PostgreSQL 7.4 and later. For descriptions of the earlier protocol versions, see previous releases of the PostgreSQL documentation. A single server can support multiple protocol versions. The initial startup-request message tells the server which protocol version the client is attempting to use, and then the server follows that protocol if it is able.

Higher level features built on this protocol (for example, how libpq passes certain environment variables when the connection is established) are covered elsewhere.

In order to serve multiple clients efficiently, the server launches a new “backend” process for each client. In the current implementation, a new child process is created immediately after an incoming connection is detected. This is transparent to the protocol, however. For purposes of the protocol, the terms “backend” and “server” are interchangeable; likewise “frontend” and “client” are interchangeable.

45.1. Overview

The protocol has separate phases for startup and normal operation. In the startup phase, the frontend opens a connection to the server and authenticates itself to the satisfaction of the server. (This might involve a single message, or multiple messages depending on the authentication method being used.) If all goes well, the server then sends status information to the frontend, and finally enters normal operation. Except for the initial startup-request message, this part of the protocol is driven by the server.

During normal operation, the frontend sends queries and other commands to the backend, and the backend sends back query results and other responses. There are a few cases (such as NOTIFY) wherein the backend will send unsolicited messages, but for the most part this portion of a session is driven by frontend requests.

Termination of the session is normally by frontend choice, but can be forced by the backend in certain cases. In any case, when the backend closes the connection, it will roll back any open (incomplete) transaction before exiting.

Within normal operation, SQL commands can be executed through either of two sub-protocols. In the “simple query” protocol, the frontend just sends a textual query string, which is parsed and immediately executed by the backend. In the ”extended query” protocol, processing of queries is separated into multiple steps: parsing, binding of parameter values, and execution. This offers flexibility and performance benefits, at the cost of extra complexity.

Normal operation has additional sub-protocols for special operations such as COPY.
45.1.1. Messaging Overview

All communication is through a stream of messages. The first byte of a message identifies the message type, and the next four bytes specify the length of the rest of the message (this length count includes itself, but not the message-type byte). The remaining contents of the message are determined by the message type. For historical reasons, the very first message sent by the client (the startup message) has no initial message-type byte.

To avoid losing synchronization with the message stream, both servers and clients typically read an entire message into a buffer (using the byte count) before attempting to process its contents. This allows easy recovery if an error is detected while processing the contents. In extreme situations (such as not having enough memory to buffer the message), the receiver can use the byte count to determine how much input to skip before it resumes reading messages.

Conversely, both servers and clients must take care never to send an incomplete message. This is commonly done by marshaling the entire message in a buffer before beginning to send it. If a communications failure occurs partway through sending or receiving a message, the only sensible response is to abandon the connection, since there is little hope of recovering message-boundary synchronization.

45.1.2. Extended Query Overview

In the extended-query protocol, execution of SQL commands is divided into multiple steps. The state retained between steps is represented by two types of objects: prepared statements and portals. A prepared statement represents the result of parsing, semantic analysis, and (optionally) planning of a textual query string. A prepared statement is not necessarily ready to execute, because it might lack specific values for parameters. A portal represents a ready-to-execute or already-partially-executed statement, with any missing parameter values filled in. (For SELECT statements, a portal is equivalent to an open cursor, but we choose to use a different term since cursors don’t handle non-SELECT statements.)

The overall execution cycle consists of a parse step, which creates a prepared statement from a textual query string; a bind step, which creates a portal given a prepared statement and values for any needed parameters; and an execute step that runs a portal’s query. In the case of a query that returns rows (SELECT, SHOW, etc), the execute step can be told to fetch only a limited number of rows, so that multiple execute steps might be needed to complete the operation.

The backend can keep track of multiple prepared statements and portals (but note that these exist only within a session, and are never shared across sessions). Existing prepared statements and portals are referenced by names assigned when they were created. In addition, an “unnamed” prepared statement and portal exist. Although these behave largely the same as named objects, operations on them are optimized for the case of executing a query only once and then discarding it, whereas operations on named objects are optimized on the expectation of multiple uses.

45.1.3. Formats and Format Codes

Data of a particular data type might be transmitted in any of several different formats. As of PostgreSQL 7.4 the only supported formats are “text” and “binary”, but the protocol makes provision for future extensions. The desired format for any value is specified by a format code. Clients can specify a format code for each transmitted parameter value and for each column of a query result. Text has format code zero, binary has format code one, and all other format codes are reserved for future definition.
Chapter 45. Frontend/Backend Protocol

The text representation of values is whatever strings are produced and accepted by the input/output conversion functions for the particular data type. In the transmitted representation, there is no trailing null character; the frontend must add one to received values if it wants to process them as C strings. (The text format does not allow embedded nulls, by the way.)

Binary representations for integers use network byte order (most significant byte first). For other data types consult the documentation or source code to learn about the binary representation. Keep in mind that binary representations for complex data types might change across server versions; the text format is usually the more portable choice.

45.2. Message Flow

This section describes the message flow and the semantics of each message type. (Details of the exact representation of each message appear in Section 45.4.) There are several different sub-protocols depending on the state of the connection: start-up, query, function call, COPY, and termination. There are also special provisions for asynchronous operations (including notification responses and command cancellation), which can occur at any time after the start-up phase.

45.2.1. Start-Up

To begin a session, a frontend opens a connection to the server and sends a startup message. This message includes the names of the user and of the database the user wants to connect to; it also identifies the particular protocol version to be used. (Optionally, the startup message can include additional settings for run-time parameters.) The server then uses this information and the contents of its configuration files (such as pg_hba.conf) to determine whether the connection is provisionally acceptable, and what additional authentication is required (if any).

The server then sends an appropriate authentication request message, to which the frontend must reply with an appropriate authentication response message (such as a password). For all authentication methods except GSSAPI and SSPI, there is at most one request and one response. In some methods, no response at all is needed from the frontend, and so no authentication request occurs. For GSSAPI and SSPI, multiple exchanges of packets may be needed to complete the authentication.

The authentication cycle ends with the server either rejecting the connection attempt (ErrorResponse), or sending AuthenticationOk.

The possible messages from the server in this phase are:

ErrorResponse

The connection attempt has been rejected. The server then immediately closes the connection.

AuthenticationOk

The authentication exchange is successfully completed.

AuthenticationKerberosV5

The frontend must now take part in a Kerberos V5 authentication dialog (not described here, part of the Kerberos specification) with the server. If this is successful, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.
AuthenticationCleartextPassword

The frontend must now send a PasswordMessage containing the password in clear-text form. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationCryptPassword

The frontend must now send a PasswordMessage containing the password encrypted via crypt(3), using the 2-character salt specified in the AuthenticationCryptPassword message. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationMD5Password

The frontend must now send a PasswordMessage containing the password encrypted via MD5, using the 4-character salt specified in the AuthenticationMD5Password message. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationSCMCredential

This response is only possible for local Unix-domain connections on platforms that support SCM credential messages. The frontend must issue an SCM credential message and then send a single data byte. (The contents of the data byte are uninteresting; it’s only used to ensure that the server waits long enough to receive the credential message.) If the credential is acceptable, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationGSS

The frontend must now initiate a GSSAPI negotiation. The frontend will send a PasswordMessage with the first part of the GSSAPI data stream in response to this. If further messages are needed, the server will respond with AuthenticationGSSContinue.

AuthenticationSSPI

The frontend must now initiate a SSPI negotiation. The frontend will send a PasswordMessage with the first part of the SSPI data stream in response to this. If further messages are needed, the server will respond with AuthenticationGSSContinue.

AuthenticationGSSContinue

This message contains the response data from the previous step of GSSAPI or SSPI negotiation (AuthenticationGSS, AuthenticationSSPI or a previous AuthenticationGSSContinue). If the GSSAPI or SSPI data in this message indicates more data is needed to complete the authentication, the frontend must send that data as another PasswordMessage. If GSSAPI or SSPI authentication is completed by this message, the server will next send AuthenticationOk to indicate successful authentication or ErrorResponse to indicate failure.

If the frontend does not support the authentication method requested by the server, then it should immediately close the connection.

After having received AuthenticationOk, the frontend must wait for further messages from the server. In this phase a backend process is being started, and the frontend is just an interested bystander. It is still possible for the startup attempt to fail (ErrorResponse), but in the normal case the backend will send some ParameterStatus messages, BackendKeyData, and finally ReadyForQuery.

During this phase the backend will attempt to apply any additional run-time parameter settings that were given in the startup message. If successful, these values become session defaults. An error causes ErrorResponse and exit.
The possible messages from the backend in this phase are:

**BackendKeyData**

This message provides secret-key data that the frontend must save if it wants to be able to issue cancel requests later. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

**ParameterStatus**

This message informs the frontend about the current (initial) setting of backend parameters, such as client_encoding or DateStyle. The frontend can ignore this message, or record the settings for its future use; see Section 45.2.6 for more details. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

**ReadyForQuery**

Start-up is completed. The frontend can now issue commands.

**ErrorResponse**

Start-up failed. The connection is closed after sending this message.

**NoticeResponse**

A warning message has been issued. The frontend should display the message but continue listening for ReadyForQuery or ErrorResponse.

The ReadyForQuery message is the same one that the backend will issue after each command cycle. Depending on the coding needs of the frontend, it is reasonable to consider ReadyForQuery as starting a command cycle, or to consider ReadyForQuery as ending the start-up phase and each subsequent command cycle.

### 45.2.2. Simple Query

A simple query cycle is initiated by the frontend sending a Query message to the backend. The message includes an SQL command (or commands) expressed as a text string. The backend then sends one or more response messages depending on the contents of the query command string, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it can safely send a new command. (It is not actually necessary for the frontend to wait for ReadyForQuery before issuing another command, but the frontend must then take responsibility for figuring out what happens if the earlier command fails and already-issued later commands succeed.)

The possible response messages from the backend are:

**CommandComplete**

An SQL command completed normally.

**CopyInResponse**

The backend is ready to copy data from the frontend to a table; see Section 45.2.5.

**CopyOutResponse**

The backend is ready to copy data from a table to the frontend; see Section 45.2.5.
RowDescription
Indicates that rows are about to be returned in response to a SELECT, FETCH, etc query. The contents of this message describe the column layout of the rows. This will be followed by a DataRow message for each row being returned to the frontend.

DataRow
One of the set of rows returned by a SELECT, FETCH, etc query.

EmptyQueryResponse
An empty query string was recognized.

ErrorResponse
An error has occurred.

ReadyForQuery
Processing of the query string is complete. A separate message is sent to indicate this because the query string might contain multiple SQL commands. (CommandComplete marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse
A warning message has been issued in relation to the query. Notices are in addition to other responses, i.e., the backend will continue processing the command.

The response to a SELECT query (or other queries that return row sets, such as EXPLAIN or SHOW) normally consists of RowDescription, zero or more DataRow messages, and then CommandComplete. COPY to or from the frontend invokes special protocol as described in Section 45.2.5. All other query types normally produce only a CommandComplete message.

Since a query string could contain several queries (separated by semicolons), there might be several such response sequences before the backend finishes processing the query string. ReadyForQuery is issued when the entire string has been processed and the backend is ready to accept a new query string.

If a completely empty (no contents other than whitespace) query string is received, the response is EmptyQueryResponse followed by ReadyForQuery.

In the event of an error, ErrorResponse is issued followed by ReadyForQuery. All further processing of the query string is aborted by ErrorResponse (even if more queries remained in it). Note that this might occur partway through the sequence of messages generated by an individual query.

In simple Query mode, the format of retrieved values is always text, except when the given command is a FETCH from a cursor declared with the BINARY option. In that case, the retrieved values are in binary format. The format codes given in the RowDescription message tell which format is being used.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message. See also Section 45.2.6 concerning messages that the backend might generate due to outside events.

Recommended practice is to code frontends in a state-machine style that will accept any message type at any time that it could make sense, rather than wiring in assumptions about the exact sequence of messages.
45.2.3. Extended Query

The extended query protocol breaks down the above-described simple query protocol into multiple steps. The results of preparatory steps can be re-used multiple times for improved efficiency. Furthermore, additional features are available, such as the possibility of supplying data values as separate parameters instead of having to insert them directly into a query string.

In the extended protocol, the frontend first sends a Parse message, which contains a textual query string, optionally some information about data types of parameter placeholders, and the name of a destination prepared-statement object (an empty string selects the unnamed prepared statement). The response is either ParseComplete or ErrorResponse. Parameter data types can be specified by OID; if not given, the parser attempts to infer the data types in the same way as it would do for untyped literal string constants.

Note: A parameter data type can be left unspecified by setting it to zero, or by making the array of parameter type OIDs shorter than the number of parameter symbols \( \$n \) used in the query string. Another special case is that a parameter’s type can be specified as void (that is, the OID of the void pseudotype). This is meant to allow parameter symbols to be used for function parameters that are actually OUT parameters. Ordinarily there is no context in which a void parameter could be used, but if such a parameter symbol appears in a function’s parameter list, it is effectively ignored. For example, a function call such as \( \text{foo}() \) could match a function with two IN and two OUT arguments, if \( \$3 \) and \( \$4 \) are specified as having type void.

Note: The query string contained in a Parse message cannot include more than one SQL statement; else a syntax error is reported. This restriction does not exist in the simple-query protocol, but it does exist in the extended protocol, because allowing prepared statements or portals to contain multiple commands would complicate the protocol unduly.

If successfully created, a named prepared-statement object lasts till the end of the current session, unless explicitly destroyed. An unnamed prepared statement lasts only until the next Parse statement specifying the unnamed statement as destination is issued. (Note that a simple Query message also destroys the unnamed statement.) Named prepared statements must be explicitly closed before they can be redefined by a Parse message, but this is not required for the unnamed statement. Named prepared statements can also be created and accessed at the SQL command level, using PREPARE and EXECUTE.

Once a prepared statement exists, it can be readied for execution using a Bind message. The Bind message gives the name of the source prepared statement (empty string denotes the unnamed prepared statement), the name of the destination portal (empty string denotes the unnamed portal), and the values to use for any parameter placeholders present in the prepared statement. The supplied parameter set must match those needed by the prepared statement. (If you declared any void parameters in the Parse message, pass NULL values for them in the Bind message.) Bind also specifies the format to use for any data returned by the query; the format can be specified overall, or per-column. The response is either BindComplete or ErrorResponse.

Note: The choice between text and binary output is determined by the format codes given in Bind, regardless of the SQL command involved. The BINARY attribute in cursor declarations is irrelevant when using extended query protocol.
Chapter 45. Frontend/Backend Protocol

Query planning for named prepared-statement objects occurs when the Parse message is processed. If a query will be repeatedly executed with different parameters, it might be beneficial to send a single Parse message containing a parameterized query, followed by multiple Bind and Execute messages. This will avoid replanning the query on each execution.

The unnamed prepared statement is likewise planned during Parse processing if the Parse message defines no parameters. But if there are parameters, query planning occurs during Bind processing instead. This allows the planner to make use of the actual values of the parameters provided in the Bind message when planning the query.

Note: Query plans generated from a parameterized query might be less efficient than query plans generated from an equivalent query with actual parameter values substituted. The query planner cannot make decisions based on actual parameter values (for example, index selectivity) when planning a parameterized query assigned to a named prepared-statement object. This possible penalty is avoided when using the unnamed statement, since it is not planned until actual parameter values are available. The cost is that planning must occur afresh for each Bind, even if the query stays the same.

If successfully created, a named portal object lasts till the end of the current transaction, unless explicitly destroyed. An unnamed portal is destroyed at the end of the transaction, or as soon as the next Bind statement specifying the unnamed portal as destination is issued. (Note that a simple Query message also destroys the unnamed portal.) Named portals must be explicitly closed before they can be redefined by a Bind message, but this is not required for the unnamed portal. Named portals can also be created and accessed at the SQL command level, using DECLARE CURSOR and FETCH.

Once a portal exists, it can be executed using an Execute message. The Execute message specifies the portal name (empty string denotes the unnamed portal) and a maximum result-row count (zero meaning “fetch all rows”). The result-row count is only meaningful for portals containing commands that return row sets; in other cases the command is always executed to completion, and the row count is ignored. The possible responses to Execute are the same as those described above for queries issued via simple query protocol, except that Execute doesn’t cause ReadyForQuery or RowDescription to be issued.

If Execute terminates before completing the execution of a portal (due to reaching a nonzero result-row count), it will send a PortalSuspended message; the appearance of this message tells the frontend that another Execute should be issued against the same portal to complete the operation. The CommandComplete message indicating completion of the source SQL command is not sent until the portal’s execution is completed. Therefore, an Execute phase is always terminated by the appearance of exactly one of these messages: CommandComplete, EmptyQueryResponse (if the portal was created from an empty query string), ErrorResponse, or PortalSuspended.

At completion of each series of extended-query messages, the frontend should issue a Sync message. This parameterless message causes the backend to close the current transaction if it’s not inside a BEGIN/COMMIT transaction block (“close” meaning to commit if no error, or roll back if error). Then a ReadyForQuery response is issued. The purpose of Sync is to provide a resynchronization point for error recovery. When an error is detected while processing any extended-query message, the backend issues ErrorResponse, then reads and discards messages until a Sync is reached, then issues ReadyForQuery and returns to normal message processing. (But note that no skipping occurs if an error is detected while processing Sync — this ensures that there is one and only one ReadyForQuery sent for each Sync.)

Note: Sync does not cause a transaction block opened with BEGIN to be closed. It is possible to detect this situation since the ReadyForQuery message includes transaction status information.
In addition to these fundamental, required operations, there are several optional operations that can be used with extended-query protocol.

The Describe message (portal variant) specifies the name of an existing portal (or an empty string for the unnamed portal). The response is a RowDescription message describing the rows that will be returned by executing the portal; or a NoData message if the portal does not contain a query that will return rows; or ErrorResponse if there is no such portal.

The Describe message (statement variant) specifies the name of an existing prepared statement (or an empty string for the unnamed prepared statement). The response is a ParameterDescription message describing the parameters needed by the statement, followed by a RowDescription message describing the rows that will be returned when the statement is eventually executed (or a NoData message if the statement will not return rows). ErrorResponse is issued if there is no such prepared statement. Note that since Bind has not yet been issued, the formats to be used for returned columns are not yet known to the backend; the format code fields in the RowDescription message will be zeroes in this case.

**Tip:** In most scenarios the frontend should issue one or the other variant of Describe before issuing Execute, to ensure that it knows how to interpret the results it will get back.

The Close message closes an existing prepared statement or portal and releases resources. It is not an error to issue Close against a nonexistent statement or portal name. The response is normally CloseComplete, but could be ErrorResponse if some difficulty is encountered while releasing resources. Note that closing a prepared statement implicitly closes any open portals that were constructed from that statement.

The Flush message does not cause any specific output to be generated, but forces the backend to deliver any data pending in its output buffers. A Flush must be sent after any extended-query command except Sync, if the frontend wishes to examine the results of that command before issuing more commands. Without Flush, messages returned by the backend will be combined into the minimum possible number of packets to minimize network overhead.

**Note:** The simple Query message is approximately equivalent to the series Parse, Bind, portal Describe, Execute, Close, Sync, using the unnamed prepared statement and portal objects and no parameters. One difference is that it will accept multiple SQL statements in the query string, automatically performing the bind/describe/execute sequence for each one in succession. Another difference is that it will not return ParseComplete, BindComplete, CloseComplete, or NoData messages.

### 45.2.4. Function Call
The Function Call sub-protocol allows the client to request a direct call of any function that exists in the database's `pg_proc` system catalog. The client must have execute permission for the function.

**Note:** The Function Call sub-protocol is a legacy feature that is probably best avoided in new code. Similar results can be accomplished by setting up a prepared statement that does `SELECT function($1, ...)`. The Function Call cycle can then be replaced with Bind/Execute.
Chapter 45. Frontend/Backend Protocol

A Function Call cycle is initiated by the frontend sending a FunctionCall message to the backend. The backend then sends one or more response messages depending on the results of the function call, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it can safely send a new query or function call.

The possible response messages from the backend are:

ErrorResponse
An error has occurred.

FunctionCallResponse
The function call was completed and returned the result given in the message. (Note that the Function Call protocol can only handle a single scalar result, not a row type or set of results.)

ReadyForQuery
Processing of the function call is complete. ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse
A warning message has been issued in relation to the function call. Notices are in addition to other responses, i.e., the backend will continue processing the command.

45.2.5. COPY Operations

The COPY command allows high-speed bulk data transfer to or from the server. Copy-in and copy-out operations each switch the connection into a distinct sub-protocol, which lasts until the operation is completed.

Copy-in mode (data transfer to the server) is initiated when the backend executes a COPY FROM STDIN SQL statement. The backend sends a CopyInResponse message to the frontend. The frontend should then send zero or more CopyData messages, forming a stream of input data. (The message boundaries are not required to have anything to do with row boundaries, although that is often a reasonable choice.) The frontend can terminate the copy-in mode by sending either a CopyDone message (allowing successful termination) or a CopyFail message (which will cause the COPY SQL statement to fail with an error). The backend then reverts to the command-processing mode it was in before the COPY started, which will be either simple or extended query protocol. It will next send either CommandComplete (if successful) or ErrorResponse (if not).

In the event of a backend-detected error during copy-in mode (including receipt of a CopyFail message), the backend will issue an ErrorResponse message. If the COPY command was issued via an extended-query message, the backend will now discard frontend messages until a Sync message is received, then it will issue ReadyForQuery and return to normal processing. If the COPY command was issued in a simple Query message, the rest of that message is discarded and ReadyForQuery is issued. In either case, any subsequent CopyData, CopyDone, or CopyFail messages issued by the frontend will simply be dropped.

The backend will ignore Flush and Sync messages received during copy-in mode. Receipt of any other non-copy message type constitutes an error that will abort the copy-in state as described above. (The exception for Flush and Sync is for the convenience of client libraries that always send Flush or Sync after an Execute message, without checking whether the command to be executed is a COPY FROM STDIN.)
Copy-out mode (data transfer from the server) is initiated when the backend executes a \texttt{COPY TO STDOUT} SQL statement. The backend sends a CopyOutResponse message to the frontend, followed by zero or more CopyData messages (always one per row), followed by CopyDone. The backend then reverts to the command-processing mode it was in before the \texttt{COPY} started, and sends CommandComplete. The frontend cannot abort the transfer (except by closing the connection or issuing a Cancel request), but it can discard unwanted CopyData and CopyDone messages.

In the event of a backend-detected error during copy-out mode, the backend will issue an ErrorResponse message and revert to normal processing. The frontend should treat receipt of ErrorResponse as terminating the copy-out mode.

It is possible for NoticeResponse and ParameterStatus messages to be interspersed between CopyData messages; frontends must handle these cases, and should be prepared for other asynchronous message types as well (see Section 45.2.6). Otherwise, any message type other than CopyData or CopyDone may be treated as terminating copy-out mode.

The CopyInResponse and CopyOutResponse messages include fields that inform the frontend of the number of columns per row and the format codes being used for each column. (As of the present implementation, all columns in a given \texttt{COPY} operation will use the same format, but the message design does not assume this.)

### 45.2.6. Asynchronous Operations

There are several cases in which the backend will send messages that are not specifically prompted by the frontend’s command stream. Frontends must be prepared to deal with these messages at any time, even when not engaged in a query. At minimum, one should check for these cases before beginning to read a query response.

It is possible for NoticeResponse messages to be generated due to outside activity; for example, if the database administrator commands a “fast” database shutdown, the backend will send a NoticeResponse indicating this fact before closing the connection. Accordingly, frontends should always be prepared to accept and display NoticeResponse messages, even when the connection is nominally idle.

ParameterStatus messages will be generated whenever the active value changes for any of the parameters the backend believes the frontend should know about. Most commonly this occurs in response to a \texttt{SET} SQL command executed by the frontend, and this case is effectively synchronous — but it is also possible for parameter status changes to occur because the administrator changed a configuration file and then sent the SIGHUP signal to the server. Also, if a \texttt{SET} command is rolled back, an appropriate ParameterStatus message will be generated to report the current effective value.

At present there is a hard-wired set of parameters for which ParameterStatus will be generated: they are \texttt{server_version}, \texttt{server_encoding}, \texttt{client_encoding}, \texttt{is_superuser}, \texttt{session_authorization}, \texttt{DateStyle}, \texttt{TimeZone}, \texttt{integer_datetimes}, and \texttt{standard_conforming_strings}. (\texttt{server_encoding}, \texttt{TimeZone}, and \texttt{integer_datetimes} were not reported by releases before 8.0; \texttt{standard_conforming_strings} was not reported by releases before 8.1.) Note that \texttt{server_version}, \texttt{server_encoding} and \texttt{integer_datetimes} are pseudo-parameters that cannot change after startup. This set might change in the future, or even become configurable. Accordingly, a frontend should simply ignore ParameterStatus for parameters that it does not understand or care about.

If a frontend issues a \texttt{LISTEN} command, then the backend will send a NotificationResponse message (not to be confused with NoticeResponse!) whenever a \texttt{NOTIFY} command is executed for the same notification name.
Chapter 45. Frontend/Backend Protocol

Note: At present, NotificationResponse can only be sent outside a transaction, and thus it will not occur in the middle of a command-response series, though it might occur just before ReadyFor-Query. It is unwise to design frontend logic that assumes that, however. Good practice is to be able to accept NotificationResponse at any point in the protocol.

45.2.7. Cancelling Requests in Progress

During the processing of a query, the frontend might request cancellation of the query. The cancel request is not sent directly on the open connection to the backend for reasons of implementation efficiency: we don’t want to have the backend constantly checking for new input from the frontend during query processing. Cancel requests should be relatively infrequent, so we make them slightly cumbersome in order to avoid a penalty in the normal case.

To issue a cancel request, the frontend opens a new connection to the server and sends a CancelRequest message, rather than the StartupMessage message that would ordinarily be sent across a new connection. The server will process this request and then close the connection. For security reasons, no direct reply is made to the cancel request message.

A CancelRequest message will be ignored unless it contains the same key data (PID and secret key) passed to the frontend during connection start-up. If the request matches the PID and secret key for a currently executing backend, the processing of the current query is aborted. (In the existing implementation, this is done by sending a special signal to the backend process that is processing the query.)

The cancellation signal might or might not have any effect — for example, if it arrives after the backend has finished processing the query, then it will have no effect. If the cancellation is effective, it results in the current command being terminated early with an error message.

The upshot of all this is that for reasons of both security and efficiency, the frontend has no direct way to tell whether a cancel request has succeeded. It must continue to wait for the backend to respond to the query. Issuing a cancel simply improves the odds that the current query will finish soon, and improves the odds that it will fail with an error message instead of succeeding.

Since the cancel request is sent across a new connection to the server and not across the regular frontend/backend communication link, it is possible for the cancel request to be issued by any process, not just the frontend whose query is to be canceled. This might provide additional flexibility when building multiple-process applications. It also introduces a security risk, in that unauthorized persons might try to cancel queries. The security risk is addressed by requiring a dynamically generated secret key to be supplied in cancel requests.

45.2.8. Termination

The normal, graceful termination procedure is that the frontend sends a Terminate message and immediately closes the connection. On receipt of this message, the backend closes the connection and terminates.

In rare cases (such as an administrator-commanded database shutdown) the backend might disconnect without any frontend request to do so. In such cases the backend will attempt to send an error or notice message giving the reason for the disconnection before it closes the connection.

Other termination scenarios arise from various failure cases, such as core dump at one end or the other, loss of the communications link, loss of message-boundary synchronization, etc. If either frontend or
backend sees an unexpected closure of the connection, it should clean up and terminate. The frontend has the option of launching a new backend by recontacting the server if it doesn’t want to terminate itself. Closing the connection is also advisable if an unrecognizable message type is received, since this probably indicates loss of message-boundary sync.

For either normal or abnormal termination, any open transaction is rolled back, not committed. One should note however that if a frontend disconnects while a non-SELECT query is being processed, the backend will probably finish the query before noticing the disconnection. If the query is outside any transaction block (BEGIN ... COMMIT sequence) then its results might be committed before the disconnection is recognized.

### 45.2.9. SSL Session Encryption

If PostgreSQL was built with SSL support, frontend/backend communications can be encrypted using SSL. This provides communication security in environments where attackers might be able to capture the session traffic. For more information on encrypting PostgreSQL sessions with SSL, see Section 17.8.

To initiate an SSL-encrypted connection, the frontend initially sends an SSLRequest message rather than a StartupMessage. The server then responds with a single byte containing $S$ or $N$, indicating that it is willing or unwilling to perform SSL, respectively. The frontend might close the connection at this point if it is dissatisfied with the response. To continue after $S$, perform an SSL startup handshake (not described here, part of the SSL specification) with the server. If this is successful, continue with sending the usual StartupMessage. In this case the StartupMessage and all subsequent data will be SSL-encrypted. To continue after $N$, send the usual StartupMessage and proceed without encryption.

The frontend should also be prepared to handle an ErrorMessage response to SSLRequest from the server. This would only occur if the server predates the addition of SSL support to PostgreSQL. In this case the connection must be closed, but the frontend might choose to open a fresh connection and proceed without requesting SSL.

An initial SSLRequest can also be used in a connection that is being opened to send a CancelRequest message.

While the protocol itself does not provide a way for the server to force SSL encryption, the administrator can configure the server to reject unencrypted sessions as a byproduct of authentication checking.

### 45.3. Message Data Types

This section describes the base data types used in messages.

**Int$n$(i)**

An $n$-bit integer in network byte order (most significant byte first). If $i$ is specified it is the exact value that will appear, otherwise the value is variable. Eg. Int16, Int32(42).

**Int$n$[k]**

An array of $k$ $n$-bit integers, each in network byte order. The array length $k$ is always determined by an earlier field in the message. Eg. Int16[M].
String($s$)
A null-terminated string (C-style string). There is no specific length limitation on strings. If $s$ is specified it is the exact value that will appear, otherwise the value is variable. Eg. String, String("user").

Note: There is no predefined limit on the length of a string that can be returned by the backend. Good coding strategy for a frontend is to use an expandable buffer so that anything that fits in memory can be accepted. If that's not feasible, read the full string and discard trailing characters that don't fit into your fixed-size buffer.

Byte$n$($c$)
Exactly $n$ bytes. If the field width $n$ is not a constant, it is always determinable from an earlier field in the message. If $c$ is specified it is the exact value. Eg. Byte2, Byte1(\n’).

45.4. Message Formats
This section describes the detailed format of each message. Each is marked to indicate that it can be sent by a frontend (F), a backend (B), or both (F & B). Notice that although each message includes a byte count at the beginning, the message format is defined so that the message end can be found without reference to the byte count. This aids validity checking. (The CopyData message is an exception, because it forms part of a data stream; the contents of any individual CopyData message cannot be interpretable on their own.)

AuthenticationOk (B)
Byte1(‘R’)
Identifies the message as an authentication request.
Int32(8)
Length of message contents in bytes, including self.
Int32(0)
Specifies that the authentication was successful.

AuthenticationKerberosV5 (B)
Byte1(‘R’)
Identifies the message as an authentication request.
Int32(8)
Length of message contents in bytes, including self.
Int32(2)
Specifies that Kerberos V5 authentication is required.
AuthenticationClearPassword (B)

Byte1('R')
Identifies the message as an authentication request.
Int32(8)
Length of message contents in bytes, including self.
Int32(3)
Specifies that a clear-text password is required.

AuthenticationCryptPassword (B)

Byte1('R')
Identifies the message as an authentication request.
Int32(10)
Length of message contents in bytes, including self.
Int32(4)
Specifies that a crypt()-encrypted password is required.
Byte2
The salt to use when encrypting the password.

AuthenticationMD5Password (B)

Byte1('R')
Identifies the message as an authentication request.
Int32(12)
Length of message contents in bytes, including self.
Int32(5)
Specifies that an MD5-encrypted password is required.
Byte4
The salt to use when encrypting the password.

AuthenticationSCMCredential (B)

Byte1('R')
Identifies the message as an authentication request.
Int32(8)
Length of message contents in bytes, including self.
Int32(6)
  Specifies that an SCM credentials message is required.

AuthenticationGSS (B)
  
  Byte1(‘R’)
  Identifies the message as an authentication request.
  Int32(8)
  Length of message contents in bytes, including self.
  Int32(7)
  Specifies that GSSAPI authentication is required.

AuthenticationSSPI (B)
  
  Byte1(‘R’)
  Identifies the message as an authentication request.
  Int32(8)
  Length of message contents in bytes, including self.
  Int32(9)
  Specifies that SSPI authentication is required.

AuthenticationGSSContinue (B)
  
  Byte1(‘R’)
  Identifies the message as an authentication request.
  Int32
  Length of message contents in bytes, including self.
  Int32(8)
  Specifies that this message contains GSSAPI or SSPI data.
  Byte
  GSSAPI or SSPI authentication data.

BackendKeyData (B)
  
  Byte1(‘K’)
  Identifies the message as cancellation key data. The frontend must save these values if it wishes to be able to issue CancelRequest messages later.
  Int32(12)
  Length of message contents in bytes, including self.
Int32
The process ID of this backend.

Int32
The secret key of this backend.

Bind (F)

Byte1(‘B’)
Identifies the message as a Bind command.

Int32
Length of message contents in bytes, including self.

String
The name of the destination portal (an empty string selects the unnamed portal).

String
The name of the source prepared statement (an empty string selects the unnamed prepared statement).

Int16
The number of parameter format codes that follow (denoted $C$ below). This can be zero to indicate that there are no parameters or that the parameters all use the default format (text); or one, in which case the specified format code is applied to all parameters; or it can equal the actual number of parameters.

Int16[$C$]
The parameter format codes. Each must presently be zero (text) or one (binary).

Int16
The number of parameter values that follow (possibly zero). This must match the number of parameters needed by the query.

Next, the following pair of fields appear for each parameter:

Int32
The length of the parameter value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL parameter value. No value bytes follow in the NULL case.

Byte$n$
The value of the parameter, in the format indicated by the associated format code. $n$ is the above length.

After the last parameter, the following fields appear:

Int16
The number of result-column format codes that follow (denoted $R$ below). This can be zero to indicate that there are no result columns or that the result columns should all use the default format (text); or one, in which case the specified format code is applied to all result columns (if any); or it can equal the actual number of result columns of the query.
Int16
The result-column format codes. Each must presently be zero (text) or one (binary).

BindComplete (B)

Byte1(’2’)  
Identifies the message as a Bind-complete indicator.

Int32(4)  
Length of message contents in bytes, including self.

CancelRequest (F)

Int32(16)  
Length of message contents in bytes, including self.

Int32(80877102)  
The cancel request code. The value is chosen to contain \texttt{1234} in the most significant 16 bits, and \texttt{5678} in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

Int32  
The process ID of the target backend.

Int32  
The secret key for the target backend.

Close (F)

Byte1(’C’)  
Identifies the message as a Close command.

Int32  
Length of message contents in bytes, including self.

Byte1  
’s’ to close a prepared statement; or ’p’ to close a portal.

String  
The name of the prepared statement or portal to close (an empty string selects the unnamed prepared statement or portal).

CloseComplete (B)

Byte1(’3’)  
Identifies the message as a Close-complete indicator.
Chapter 45. Frontend/Backend Protocol

Int32(4)
Length of message contents in bytes, including self.

CommandComplete (B)

Byte1(‘C’)
Identifies the message as a command-completed response.

Int32
Length of message contents in bytes, including self.

String
The command tag. This is usually a single word that identifies which SQL command was completed.

For an INSERT command, the tag is INSERT oid rows, where rows is the number of rows inserted. oid is the object ID of the inserted row if rows is 1 and the target table has OIDs; otherwise oid is 0.

For a DELETE command, the tag is DELETE rows where rows is the number of rows deleted.

For an UPDATE command, the tag is UPDATE rows where rows is the number of rows updated.

For a MOVE command, the tag is MOVE rows where rows is the number of rows the cursor’s position has been changed by.

For a FETCH command, the tag is FETCH rows where rows is the number of rows that have been retrieved from the cursor.

For a COPY command, the tag is COPY rows where rows is the number of rows copied.
(Note: the row count appears only in PostgreSQL 8.2 and later.)

CopyData (F & B)

Byte1(‘d’)
Identifies the message as COPY data.

Int32
Length of message contents in bytes, including self.

Byte
Data that forms part of a COPY data stream. Messages sent from the backend will always correspond to single data rows, but messages sent by frontends might divide the data stream arbitrarily.

CopyDone (F & B)

Byte1(‘c’)
Identifies the message as a COPY-complete indicator.
Chapter 45. Frontend/Backend Protocol

Int32(4)
Length of message contents in bytes, including self.

CopyFail (F)

Byte1(‘f’)
Identifies the message as a COPY-failure indicator.
Int32
Length of message contents in bytes, including self.
String
An error message to report as the cause of failure.

CopyInResponse (B)

Byte1(‘G’)
Identifies the message as a Start Copy In response. The frontend must now send copy-in
data (if not prepared to do so, send a CopyFail message).
Int32
Length of message contents in bytes, including self.
Int8
0 indicates the overall COPY format is textual (rows separated by newlines, columns sepa-
rated by separator characters, etc). 1 indicates the overall copy format is binary (similar to
DataRow format). See COPY for more information.
Int16
The number of columns in the data to be copied (denoted $N$ below).
Int16[$N$]
The format codes to be used for each column. Each must presently be zero (text) or one
(binary). All must be zero if the overall copy format is textual.

CopyOutResponse (B)

Byte1(‘H’)
Identifies the message as a Start Copy Out response. This message will be followed by
copy-out data.
Int32
Length of message contents in bytes, including self.
Int8
0 indicates the overall COPY format is textual (rows separated by newlines, columns sepa-
rated by separator characters, etc). 1 indicates the overall copy format is binary (similar to
DataRow format). See COPY for more information.
CHAPTER 45. FRONTEND/BACKEND PROTOCOL

Int16
The number of columns in the data to be copied (denoted \( N \) below).

Int16\([N]\)
The format codes to be used for each column. Each must presently be zero (text) or one (binary). All must be zero if the overall copy format is textual.

DataRow (B)

Byte1(‘D’)
Identifies the message as a data row.

Int32
Length of message contents in bytes, including self.

Int16
The number of column values that follow (possibly zero).

Next, the following pair of fields appear for each column:

Int32
The length of the column value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL column value. No value bytes follow in the NULL case.

Byte\( n \)
The value of the column, in the format indicated by the associated format code. \( n \) is the above length.

Describe (F)

Byte1(‘D’)
Identifies the message as a Describe command.

Int32
Length of message contents in bytes, including self.

Byte1
‘S’ to describe a prepared statement; or ‘P’ to describe a portal.

String
The name of the prepared statement or portal to describe (an empty string selects the unnamed prepared statement or portal).

EmptyQueryResponse (B)

Byte1(‘I’)
Identifies the message as a response to an empty query string. (This substitutes for CommandComplete.)
Chapter 45. Frontend/Backend Protocol

Int32(4)

Length of message contents in bytes, including self.

ErrorResponse (B)

Byte1(’E’)

Identifies the message as an error.

Int32

Length of message contents in bytes, including self.

The message body consists of one or more identified fields, followed by a zero byte as a terminator. Fields can appear in any order. For each field there is the following:

Byte1

A code identifying the field type; if zero, this is the message terminator and no string follows. The presently defined field types are listed in Section 45.5. Since more field types might be added in future, frontends should silently ignore fields of unrecognized type.

String

The field value.

Execute (F)

Byte1(’E’)

Identifies the message as an Execute command.

Int32

Length of message contents in bytes, including self.

String

The name of the portal to execute (an empty string selects the unnamed portal).

Int32

Maximum number of rows to return, if portal contains a query that returns rows (ignored otherwise). Zero denotes “no limit”.

Flush (F)

Byte1(’H’)

Identifies the message as a Flush command.

Int32(4)

Length of message contents in bytes, including self.
Chapter 45. Frontend/Backend Protocol

FunctionCall (F)

Byte1('F')

Identifies the message as a function call.

Int32

Length of message contents in bytes, including self.

Int32

Specifies the object ID of the function to call.

Int16

The number of argument format codes that follow (denoted \( C \) below). This can be zero to indicate that there are no arguments or that the arguments all use the default format (text); or one, in which case the specified format code is applied to all arguments; or it can equal the actual number of arguments.

Int16[\( C \)]

The argument format codes. Each must presently be zero (text) or one (binary).

Int16

Specifies the number of arguments being supplied to the function.

Next, the following pair of fields appear for each argument:

Int32

The length of the argument value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL argument value. No value bytes follow in the NULL case.

Byte\( n \)

The value of the argument, in the format indicated by the associated format code. \( n \) is the above length.

After the last argument, the following field appears:

Int16

The format code for the function result. Must presently be zero (text) or one (binary).

FunctionCallResponse (B)

Byte1('V')

Identifies the message as a function call result.

Int32

Length of message contents in bytes, including self.

Int32

The length of the function result value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL function result. No value bytes follow in the NULL case.
Byte $n$

The value of the function result, in the format indicated by the associated format code. $n$ is the above length.

NoData (B)

Byte1('n')

Identifies the message as a no-data indicator.

Int32(4)

Length of message contents in bytes, including self.

NoticeResponse (B)

Byte1('N')

Identifies the message as a notice.

Int32

Length of message contents in bytes, including self.

The message body consists of one or more identified fields, followed by a zero byte as a terminator. Fields can appear in any order. For each field there is the following:

Byte1

A code identifying the field type; if zero, this is the message terminator and no string follows. The presently defined field types are listed in Section 45.5. Since more field types might be added in future, frontends should silently ignore fields of unrecognized type.

String

The field value.

NotificationResponse (B)

Byte1('A')

Identifies the message as a notification response.

Int32

Length of message contents in bytes, including self.

Int32

The process ID of the notifying backend process.

String

The name of the condition that the notify has been raised on.

String

Additional information passed from the notifying process. (Currently, this feature is unimplemented so the field is always an empty string.)
ParameterDescription (B)

Byte1('t')
   Identifies the message as a parameter description.
Int32
   Length of message contents in bytes, including self.
Int16
   The number of parameters used by the statement (can be zero).
Then, for each parameter, there is the following:

Int32
   Specifies the object ID of the parameter data type.

ParameterStatus (B)

Byte1('S')
   Identifies the message as a run-time parameter status report.
Int32
   Length of message contents in bytes, including self.
String
   The name of the run-time parameter being reported.
String
   The current value of the parameter.

Parse (F)

Byte1('P')
   Identifies the message as a Parse command.
Int32
   Length of message contents in bytes, including self.
String
   The name of the destination prepared statement (an empty string selects the unnamed prepared statement).
String
   The query string to be parsed.
Int16
   The number of parameter data types specified (can be zero). Note that this is not an indication of the number of parameters that might appear in the query string, only the number that the frontend wants to prespecify types for.
Then, for each parameter, there is the following:
Int32

Specifies the object ID of the parameter data type. Placing a zero here is equivalent to leaving the type unspecified.

ParseComplete (B)

Byte1('1')

Identifies the message as a Parse-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

PasswordMessage (F)

Byte1('p')

Identifies the message as a password response. Note that this is also used for GSSAPI and SSPI response messages (which is really a design error, since the contained data is not a null-terminated string in that case, but can be arbitrary binary data).

Int32

Length of message contents in bytes, including self.

String

The password (encrypted, if requested).

PortalSuspended (B)

Byte1('s')

Identifies the message as a portal-suspended indicator. Note this only appears if an Execute message’s row-count limit was reached.

Int32(4)

Length of message contents in bytes, including self.

Query (F)

Byte1('Q')

Identifies the message as a simple query.

Int32

Length of message contents in bytes, including self.

String

The query string itself.
ReadyForQuery (B)

Byte1(‘Z’)
Identifies the message type. ReadyForQuery is sent whenever the backend is ready for a
new query cycle.

Int32(5)
Length of message contents in bytes, including self.

Byte1
Current backend transaction status indicator. Possible values are ‘I’ if idle (not in a transac-
tion block); ‘T’ if in a transaction block; or ‘E’ if in a failed transaction block (queries will
be rejected until block is ended).

RowDescription (B)

Byte1(‘T’)
Identifies the message as a row description.

Int32
Length of message contents in bytes, including self.

Int16
Specifies the number of fields in a row (can be zero).

Then, for each field, there is the following:

String
The field name.

Int32
If the field can be identified as a column of a specific table, the object ID of the table;
otherwise zero.

Int16
If the field can be identified as a column of a specific table, the attribute number of the
column; otherwise zero.

Int32
The object ID of the field’s data type.

Int16
The data type size (see pg_type.typlen). Note that negative values denote variable-width
types.

Int32
The type modifier (see pg_attribute.attypmod). The meaning of the modifier is type-
specific.
Chapter 45. Frontend/Backend Protocol

Int16

The format code being used for the field. Currently will be zero (text) or one (binary). In a RowDescription returned from the statement variant of Describe, the format code is not yet known and will always be zero.

SSLRequest (F)

Int32(8)

Length of message contents in bytes, including self.

Int32(80877103)

The SSL request code. The value is chosen to contain 1234 in the most significant 16 bits, and 5679 in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

StartupMessage (F)

Int32

Length of message contents in bytes, including self.

Int32(196608)

The protocol version number. The most significant 16 bits are the major version number (3 for the protocol described here). The least significant 16 bits are the minor version number (0 for the protocol described here).

The protocol version number is followed by one or more pairs of parameter name and value strings. A zero byte is required as a terminator after the last name/value pair. Parameters can appear in any order. user is required, others are optional. Each parameter is specified as:

String

The parameter name. Currently recognized names are:

user

The database user name to connect as. Required; there is no default.

database

The database to connect to. Defaults to the user name.

options

Command-line arguments for the backend. (This is deprecated in favor of setting individual run-time parameters.)

In addition to the above, any run-time parameter that can be set at backend start time might be listed. Such settings will be applied during backend start (after parsing the command-line options if any). The values will act as session defaults.

String

The parameter value.
45.5. Error and Notice Message Fields

This section describes the fields that can appear in ErrorResponse and NoticeResponse messages. Each field type has a single-byte identification token. Note that any given field type should appear at most once per message.

S

Severity: the field contents are ERROR, FATAL, or PANIC (in an error message), or WARNING, NOTICE, DEBUG, INFO, or LOG (in a notice message), or a localized translation of one of these. Always present.

C

Code: the SQLSTATE code for the error (see Appendix A). Not localizable. Always present.

M

Message: the primary human-readable error message. This should be accurate but terse (typically one line). Always present.

D

Detail: an optional secondary error message carrying more detail about the problem. Might run to multiple lines.

H

Hint: an optional suggestion what to do about the problem. This is intended to differ from Detail in that it offers advice (potentially inappropriate) rather than hard facts. Might run to multiple lines.

P

Position: the field value is a decimal ASCII integer, indicating an error cursor position as an index into the original query string. The first character has index 1, and positions are measured in characters not bytes.
Chapter 45. Frontend/Backend Protocol

Internal position: this is defined the same as the \( p \) field, but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client. The \( q \) field will always appear when this field appears.

Internal query: the text of a failed internally-generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

Where: an indication of the context in which the error occurred. Presently this includes a call stack traceback of active procedural language functions and internally-generated queries. The trace is one entry per line, most recent first.

File: the file name of the source-code location where the error was reported.

Line: the line number of the source-code location where the error was reported.

Routine: the name of the source-code routine reporting the error.

The client is responsible for formatting displayed information to meet its needs; in particular it should break long lines as needed. Newline characters appearing in the error message fields should be treated as paragraph breaks, not line breaks.

45.6. Summary of Changes since Protocol 2.0

This section provides a quick checklist of changes, for the benefit of developers trying to update existing client libraries to protocol 3.0.

The initial startup packet uses a flexible list-of-strings format instead of a fixed format. Notice that session default values for run-time parameters can now be specified directly in the startup packet. (Actually, you could do that before using the \( \text{options} \) field, but given the limited width of \( \text{options} \) and the lack of any way to quote whitespace in the values, it wasn’t a very safe technique.)

All messages now have a length count immediately following the message type byte (except for startup packets, which have no type byte). Also note that \( \text{PasswordMessage} \) now has a type byte.

\( \text{ErrorResponse} \) and \( \text{NoticeResponse} \) (‘E’ and ‘N’) messages now contain multiple fields, from which the client code can assemble an error message of the desired level of verbosity. Note that individual fields will typically not end with a newline, whereas the single string sent in the older protocol always did.

The ReadyForQuery (‘Z’) message includes a transaction status indicator.

The distinction between BinaryRow and DataRow message types is gone; the single DataRow message type serves for returning data in all formats. Note that the layout of DataRow has changed to make it easier to parse. Also, the representation of binary values has changed: it is no longer directly tied to the server’s internal representation.

There is a new “extended query” sub-protocol, which adds the frontend message types Parse, Bind, Execute, Describe, Close, Flush, and Sync, and the backend message types ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData, and CloseComplete. Existing clients do
not have to concern themselves with this sub-protocol, but making use of it might allow improvements in performance or functionality.

COPY data is now encapsulated into CopyData and CopyDone messages. There is a well-defined way to recover from errors during COPY. The special "\." last line is not needed anymore, and is not sent during COPY OUT. (It is still recognized as a terminator during COPY IN, but its use is deprecated and will eventually be removed.) Binary COPY is supported. The CopyInResponse and CopyOutResponse messages include fields indicating the number of columns and the format of each column.

The layout of FunctionCall and FunctionCallResponse messages has changed. FunctionCall can now support passing NULL arguments to functions. It also can handle passing parameters and retrieving results in either text or binary format. There is no longer any reason to consider FunctionCall a potential security hole, since it does not offer direct access to internal server data representations.

The backend sends ParameterStatus ("S") messages during connection startup for all parameters it considers interesting to the client library. Subsequently, a ParameterStatus message is sent whenever the active value changes for any of these parameters.

The RowDescription ("T") message carries new table OID and column number fields for each column of the described row. It also shows the format code for each column.

The CursorResponse ("P") message is no longer generated by the backend.

The NotificationResponse ("A") message has an additional string field, which is presently empty but might someday carry additional data passed from the NOTIFY event sender.

The EmptyQueryResponse ("I") message used to include an empty string parameter; this has been removed.
Chapter 46. PostgreSQL Coding Conventions

46.1. Formatting

Source code formatting uses 4 column tab spacing, with tabs preserved (i.e. tabs are not expanded to spaces). Each logical indentation level is one additional tab stop. Layout rules (brace positioning, etc) follow BSD conventions.

While submitted patches do not absolutely have to follow these formatting rules, it’s a good idea to do so. Your code will get run through pgindent, so there’s no point in making it look nice under some other set of formatting conventions.

The src/tools directory contains sample settings files that can be used with the emacs, xemacs or vim editors to help ensure that they format code according to these conventions.

The text browsing tools more and less can be invoked as:

```
more -x4
less -x4
```

to make them show tabs appropriately.

46.2. Reporting Errors Within the Server

Error, warning, and log messages generated within the server code should be created using ereport, or its older cousin elog. The use of this function is complex enough to require some explanation.

There are two required elements for every message: a severity level (ranging from DEBUG to PANIC) and a primary message text. In addition there are optional elements, the most common of which is an error identifier code that follows the SQL spec’s SQLSTATE conventions. ereport itself is just a shell function, that exists mainly for the syntactic convenience of making message generation look like a function call in the C source code. The only parameter accepted directly by ereport is the severity level. The primary message text and any optional message elements are generated by calling auxiliary functions, such as errmsg, within the ereport call.

A typical call to ereport might look like this:

```
ereport (ERROR,
           (errcode (ERRCODE_DIVISION_BY_ZERO),
                    errmsg("division by zero")));
```

This specifies error severity level ERROR (a run-of-the-mill error). The errcode call specifies the SQLSTATE error code using a macro defined in src/include/utils/errcodes.h. The errmsg call provides the primary message text. Notice the extra set of parentheses surrounding the auxiliary function calls — these are annoying but syntactically necessary.

Here is a more complex example:

```
ereport (ERROR,
           (errcode (ERRCODE_AMBIGUOUS_FUNCTION),
                    errmsg("function %s is not unique"),
                    errmsg("the other function is %s")),
```
func_signature_string(funcname, nargs, actual_arg_types)),
errhint("Unable to choose a best candidate function. "
"You might need to add explicit typecasts.");

This illustrates the use of format codes to embed run-time values into a message text. Also, an optional “hint” message is provided.

The available auxiliary routines for ereport are:

- `errcode(sqlerrcode)` specifies the SQLSTATE error identifier code for the condition. If this routine is not called, the error identifier defaults to ERRCODE_INTERNAL_ERROR when the error severity level is ERROR or higher, ERRCODE_WARNING when the error level is WARNING, otherwise (for NOTICE and below) ERRCODE_SUCCESSFUL_COMPLETION. While these defaults are often convenient, always think whether they are appropriate before omitting the errcode() call.

- `errmsg(const char *msg, ...)` specifies the primary error message text, and possibly run-time values to insert into it. Insertions are specified by sprintf-style format codes. In addition to the standard format codes accepted by sprintf, the format code %m can be used to insert the error message returned by strerror for the current value of errno. %m does not require any corresponding entry in the parameter list for errmsg. Note that the message string will be run through gettext for possible localization before format codes are processed.

- `errmsg_internal(const char *msg, ...)` is the same as errmsg, except that the message string will not be translated nor included in the internationalization message dictionary. This should be used for “cannot happen” cases that are probably not worth expending translation effort on.

- `errdetail(const char *msg, ...)` supplies an optional “detail” message; this is to be used when there is additional information that seems inappropriate to put in the primary message. The message string is processed in just the same way as for errmsg.

- `errhint(const char *msg, ...)` supplies an optional “hint” message; this is to be used when offering suggestions about how to fix the problem, as opposed to factual details about what went wrong. The message string is processed in just the same way as for errmsg.

- `errcontext(const char *msg, ...)` is not normally called directly from an ereport message site; rather it is used in error_context_stack callback functions to provide information about the context in which an error occurred, such as the current location in a PL function. The message string is processed in just the same way as for errmsg. Unlike the other auxiliary functions, this can be called more than once per ereport call; the successive strings thus supplied are concatenated with separating newlines.

- `errposition(int cursorpos)` specifies the textual location of an error within a query string. Currently it is only useful for errors detected in the lexical and syntactic analysis phases of query processing.

- `errcode_for_file_access()` is a convenience function that selects an appropriate SQLSTATE error identifier for a failure in a file-access-related system call. It uses the saved errno to determine which error code to generate. Usually this should be used in combination with %m in the primary error message text.

- `errcode_for_socket_access()` is a convenience function that selects an appropriate SQL-STATE error identifier for a failure in a socket-related system call.

1. That is, the value that was current when the ereport call was reached; changes of errno within the auxiliary reporting routines will not affect it. That would not be true if you were to write strerror(errno) explicitly in errmsg's parameter list; accordingly, do not do so.
errhidestmt(bool hide_stmt) can be called to specify suppression of the STATEMENT: portion of a message in the postmaster log. Generally this is appropriate if the message text includes the current statement already.

There is an older function elog that is still heavily used. An elog call:

elog(level, "format string", ...);

is exactly equivalent to:

erreport(level, (errmsg_internal("format string", ...)));

Notice that the SQLSTATE error code is always defaulted, and the message string is not subject to translation. Therefore, elog should be used only for internal errors and low-level debug logging. Any message that is likely to be of interest to ordinary users should go through ereport. Nonetheless, there are enough internal “cannot happen” error checks in the system that elog is still widely used; it is preferred for those messages for its notational simplicity.

Advice about writing good error messages can be found in Section 46.3.

### 46.3. Error Message Style Guide

This style guide is offered in the hope of maintaining a consistent, user-friendly style throughout all the messages generated by PostgreSQL.

#### 46.3.1. What goes where

The primary message should be short, factual, and avoid reference to implementation details such as specific function names. “Short” means “should fit on one line under normal conditions”. Use a detail message if needed to keep the primary message short, or if you feel a need to mention implementation details such as the particular system call that failed. Both primary and detail messages should be factual. Use a hint message for suggestions about what to do to fix the problem, especially if the suggestion might not always be applicable.

For example, instead of:

IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)

write:

Primary: could not create shared memory segment: %m
Detail: Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint: the addendum

Rationale: keeping the primary message short helps keep it to the point, and lets clients lay out screen space on the assumption that one line is enough for error messages. Detail and hint messages can be relegated to a verbose mode, or perhaps a pop-up error-details window. Also, details and hints would normally be suppressed from the server log to save space. Reference to implementation details is best avoided since users don’t know the details anyway.
### 46.3.2. Formatting

Don’t put any specific assumptions about formatting into the message texts. Expect clients and the server log to wrap lines to fit their own needs. In long messages, newline characters (\n) can be used to indicate suggested paragraph breaks. Don’t end a message with a newline. Don’t use tabs or other formatting characters. (In error context displays, newlines are automatically added to separate levels of context such as function calls.)

**Rationale:** Messages are not necessarily displayed on terminal-type displays. In GUI displays or browsers these formatting instructions are at best ignored.

### 46.3.3. Quotation marks

English text should use double quotes when quoting is appropriate. Text in other languages should consistently use one kind of quotes that is consistent with publishing customs and computer output of other programs.

**Rationale:** The choice of double quotes over single quotes is somewhat arbitrary, but tends to be the preferred use. Some have suggested choosing the kind of quotes depending on the type of object according to SQL conventions (namely, strings single quoted, identifiers double quoted). But this is a language-internal technical issue that many users aren’t even familiar with, it won’t scale to other kinds of quoted terms, it doesn’t translate to other languages, and it’s pretty pointless, too.

### 46.3.4. Use of quotes

Use quotes always to delimit file names, user-supplied identifiers, and other variables that might contain words. Do not use them to mark up variables that will not contain words (for example, operator names).

There are functions in the backend that will double-quote their own output at need (for example, `format_type_be()`). Do not put additional quotes around the output of such functions.

**Rationale:** Objects can have names that create ambiguity when embedded in a message. Be consistent about denoting where a plugged-in name starts and ends. But don’t clutter messages with unnecessary or duplicate quote marks.

### 46.3.5. Grammar and punctuation

The rules are different for primary error messages and for detail/hint messages:

**Primary error messages:** Do not capitalize the first letter. Do not end a message with a period. Do not even think about ending a message with an exclamation point.

**Detail and hint messages:** Use complete sentences, and end each with a period. Capitalize the first word of sentences. Put two spaces after the period if another sentence follows (for English text; might be inappropriate in other languages).

**Rationale:** Avoiding punctuation makes it easier for client applications to embed the message into a variety of grammatical contexts. Often, primary messages are not grammatically complete sentences anyway. (And if they’re long enough to be more than one sentence, they should be split into primary and detail parts.) However, detail and hint messages are longer and might need to include multiple sentences. For consistency, they should follow complete-sentence style even when there’s only one sentence.
46.3.6. Upper case vs. lower case

Use lower case for message wording, including the first letter of a primary error message. Use upper case for SQL commands and key words if they appear in the message.

Rationale: It’s easier to make everything look more consistent this way, since some messages are complete sentences and some not.

46.3.7. Avoid passive voice

Use the active voice. Use complete sentences when there is an acting subject (“A could not do B”). Use telegram style without subject if the subject would be the program itself; do not use “I” for the program.

Rationale: The program is not human. Don’t pretend otherwise.

46.3.8. Present vs past tense

Use past tense if an attempt to do something failed, but could perhaps succeed next time (perhaps after fixing some problem). Use present tense if the failure is certainly permanent.

There is a nontrivial semantic difference between sentences of the form:

could not open file "%s": %m

and:

cannot open file "%s"

The first one means that the attempt to open the file failed. The message should give a reason, such as “disk full” or “file doesn’t exist”. The past tense is appropriate because next time the disk might not be full anymore or the file in question might exist.

The second form indicates that the functionality of opening the named file does not exist at all in the program, or that it’s conceptually impossible. The present tense is appropriate because the condition will persist indefinitely.

Rationale: Granted, the average user will not be able to draw great conclusions merely from the tense of the message, but since the language provides us with a grammar we should use it correctly.

46.3.9. Type of the object

When citing the name of an object, state what kind of object it is.

Rationale: Otherwise no one will know what “foo.bar.baz” refers to.

46.3.10. Brackets

Square brackets are only to be used (1) in command synopses to denote optional arguments, or (2) to denote an array subscript.

Rationale: Anything else does not correspond to widely-known customary usage and will confuse people.
46.3.11. Assembling error messages

When a message includes text that is generated elsewhere, embed it in this style:

could not open file %s: %m

Rationale: It would be difficult to account for all possible error codes to paste this into a single smooth sentence, so some sort of punctuation is needed. Putting the embedded text in parentheses has also been suggested, but it’s unnatural if the embedded text is likely to be the most important part of the message, as is often the case.

46.3.12. Reasons for errors

Messages should always state the reason why an error occurred. For example:

BAD: could not open file %s
BETTER: could not open file %s (I/O failure)

If no reason is known you better fix the code.

46.3.13. Function names

Don’t include the name of the reporting routine in the error text. We have other mechanisms for finding that out when needed, and for most users it’s not helpful information. If the error text doesn’t make as much sense without the function name, reword it.

BAD: pg_atoi: error in "z": cannot parse "z"
BETTER: invalid input syntax for integer: "z"

Avoid mentioning called function names, either; instead say what the code was trying to do:

BAD: open() failed: %m
BETTER: could not open file %s: %m

If it really seems necessary, mention the system call in the detail message. (In some cases, providing the actual values passed to the system call might be appropriate information for the detail message.)

Rationale: Users don’t know what all those functions do.

46.3.14. Tricky words to avoid

Unable. “Unable” is nearly the passive voice. Better use “cannot” or “could not”, as appropriate.

Bad. Error messages like “bad result” are really hard to interpret intelligently. It’s better to write why the result is “bad”, e.g., “invalid format”.

Illegal. “Illegal” stands for a violation of the law, the rest is “invalid”. Better yet, say why it’s invalid.

Unknown. Try to avoid “unknown”. Consider “error: unknown response”. If you don’t know what the response is, how do you know it’s erroneous? “Unrecognized” is often a better choice. Also, be sure to include the value being complained of.
BAD: unknown node type
BETTER: unrecognized node type: 42

**Find vs. Exists.** If the program uses a nontrivial algorithm to locate a resource (e.g., a path search) and that algorithm fails, it is fair to say that the program couldn’t “find” the resource. If, on the other hand, the expected location of the resource is known but the program cannot access it there then say that the resource doesn’t “exist”. Using “find” in this case sounds weak and confuses the issue.

**May vs. Can vs. Might.** “May” suggests permission (e.g. "You may borrow my rake."). and has little use in documentation or error messages. “Can” suggests ability (e.g. "I can lift that log."). and “might” suggests possibility (e.g. "It might rain today."). Using the proper word clarifies meaning and assists translation.

**Contractions.** Avoid contractions, like “can’t”; use “cannot” instead.

### 46.3.15. Proper spelling

Spell out words in full. For instance, avoid:

- `spec`
- `stats`
- `parens`
- `auth`
- `xact`

**Rationale:** This will improve consistency.

### 46.3.16. Localization

Keep in mind that error message texts need to be translated into other languages. Follow the guidelines in Section 47.2.2 to avoid making life difficult for translators.
Chapter 47. Native Language Support

47.1. For the Translator

PostgreSQL programs (server and client) can issue their messages in your favorite language — if the messages have been translated. Creating and maintaining translated message sets needs the help of people who speak their own language well and want to contribute to the PostgreSQL effort. You do not have to be a programmer at all to do this. This section explains how to help.

47.1.1. Requirements

We won’t judge your language skills — this section is about software tools. Theoretically, you only need a text editor. But this is only in the unlikely event that you do not want to try out your translated messages. When you configure your source tree, be sure to use the `--enable-nls` option. This will also check for the libintl library and the `msgfmt` program, which all end users will need anyway. To try out your work, follow the applicable portions of the installation instructions.

If you want to start a new translation effort or want to do a message catalog merge (described later), you will need the programs `xgettext` and `msgmerge`, respectively, in a GNU-compatible implementation. Later, we will try to arrange it so that if you use a packaged source distribution, you won’t need `xgettext`. (From CVS, you will still need it.) GNU Gettext 0.10.36 or later is currently recommended.

Your local gettext implementation should come with its own documentation. Some of that is probably duplicated in what follows, but for additional details you should look there.

47.1.2. Concepts

The pairs of original (English) messages and their (possibly) translated equivalents are kept in message catalogs, one for each program (although related programs can share a message catalog) and for each target language. There are two file formats for message catalogs: The first is the “PO” file (for Portable Object), which is a plain text file with special syntax that translators edit. The second is the “MO” file (for Machine Object), which is a binary file generated from the respective PO file and is used while the internationalized program is run. Translators do not deal with MO files; in fact hardly anyone does.

The extension of the message catalog file is no surprise either .po or .mo. The base name is either the name of the program it accompanies, or the language the file is for, depending on the situation. This is a bit confusing. Examples are `psql.po` (PO file for `psql`) or `fr.mo` (MO file in French).

The file format of the PO files is illustrated here:

```plaintext
msgid "original string"
msgstr "translated string"
msgid "more original"
```
msgstr "another translated"
"string can be broken up like this"

The msgid’s are extracted from the program source. (They need not be, but this is the most common way.) The msgstr lines are initially empty and are filled in with useful strings by the translator. The strings can contain C-style escape characters and can be continued across lines as illustrated. (The next line must start at the beginning of the line.)

The # character introduces a comment. If whitespace immediately follows the # character, then this is a comment maintained by the translator. There can also be automatic comments, which have a non-whitespace character immediately following the #. These are maintained by the various tools that operate on the PO files and are intended to aid the translator.

#. automatic comment
#: filename.c:1023
#, flags, flags

The #. style comments are extracted from the source file where the message is used. Possibly the programmer has inserted information for the translator, such as about expected alignment. The #: comment indicates the exact location(s) where the message is used in the source. The translator need not look at the program source, but he can if there is doubt about the correct translation. The #, comments contain flags that describe the message in some way. There are currently two flags: fuzzy is set if the message has possibly been outdated because of changes in the program source. The translator can then verify this and possibly remove the fuzzy flag. Note that fuzzy messages are not made available to the end user. The other flag is c-format, which indicates that the message is a printf-style format template. This means that the translation should also be a format string with the same number and type of placeholders. There are tools that can verify this, which key off the c-format flag.

47.1.3. Creating and maintaining message catalogs

OK, so how does one create a “blank” message catalog? First, go into the directory that contains the program whose messages you want to translate. If there is a file nls.mk, then this program has been prepared for translation.

If there are already some .po files, then someone has already done some translation work. The files are named language.po, where language is the ISO 639-1 two-letter language code (in lower case), e.g., fr.po for French. If there is really a need for more than one translation effort per language then the files can also be named language_region.po where region is the ISO 3166-1 two-letter country code (in upper case), e.g., pt_BR.po for Portuguese in Brazil. If you find the language you wanted you can just start working on that file.

If you need to start a new translation effort, then first run the command:

gmake init-po

This will create a file programe.pot (.pot to distinguish it from PO files that are “in production”. The T stands for “template”.) Copy this file to language.po and edit it. To make it known that the new language is available, also edit the file nls.mk and add the language (or language and country) code to the line that looks like:

2. http://www.iso.org/iso/country_names_and_code_elements
Chapter 47. Native Language Support

AVAIL_LANGUAGES := de fr

(Other languages can appear, of course.)

As the underlying program or library changes, messages might be changed or added by the programmers. In this case you do not need to start from scratch. Instead, run the command:

gmake update-po

which will create a new blank message catalog file (the pot file you started with) and will merge it with the existing PO files. If the merge algorithm is not sure about a particular message it marks it “fuzzy” as explained above. For the case where something went really wrong, the old PO file is saved with a .po.old extension.

47.1.4. Editing the PO files

The PO files can be edited with a regular text editor. The translator should only change the area between the quotes after the msgstr directive, add comments, and alter the fuzzy flag. There is (unsurprisingly) a PO mode for Emacs, which I find quite useful.

The PO files need not be completely filled in. The software will automatically fall back to the original string if no translation (or an empty translation) is available. It is no problem to submit incomplete translations for inclusions in the source tree; that gives room for other people to pick up your work. However, you are encouraged to give priority to removing fuzzy entries after doing a merge. Remember that fuzzy entries will not be installed; they only serve as reference what might be the right translation.

Here are some things to keep in mind while editing the translations:

• Make sure that if the original ends with a newline, the translation does, too. Similarly for tabs, etc.
• If the original is a printf format string, the translation also needs to be. The translation also needs to have the same format specifiers in the same order. Sometimes the natural rules of the language make this impossible or at least awkward. In that case you can modify the format specifiers like this:

msgstr "Die Datei %2$s hat %1$u Zeichen."

Then the first placeholder will actually use the second argument from the list. The digits $ needs to follow the % immediately, before any other format manipulators. (This feature really exists in the printf family of functions. You might not have heard of it before because there is little use for it outside of message internationalization.)
• If the original string contains a linguistic mistake, report that (or fix it yourself in the program source) and translate normally. The corrected string can be merged in when the program sources have been updated. If the original string contains a factual mistake, report that (or fix it yourself) and do not translate it. Instead, you can mark the string with a comment in the PO file.
• Maintain the style and tone of the original string. Specifically, messages that are not sentences (cannot open file %s) should probably not start with a capital letter (if your language distinguishes letter case) or end with a period (if your language uses punctuation marks). It might help to read Section 46.3.
• If you don’t know what a message means, or if it is ambiguous, ask on the developers’ mailing list. Chances are that English speaking end users might also not understand it or find it ambiguous, so it’s best to improve the message.
47.2. For the Programmer

47.2.1. Mechanics

This section describes how to implement native language support in a program or library that is part of the PostgreSQL distribution. Currently, it only applies to C programs.

Adding NLS support to a program

1. Insert this code into the start-up sequence of the program:

```c
#ifdef ENABLE_NLS
#include <locale.h>
#endif
...

#ifdef ENABLE_NLS
setlocale(LC_ALL, "");
bindtextdomain("prognamex", LOCALEDIR);
textdomain("prognamex");
#endif
(The `prognamex` can actually be chosen freely.)
```

2. Wherever a message that is a candidate for translation is found, a call to `gettext()` needs to be inserted. E.g.:

```c
fprintf(stderr, "panic level %d\n", lvl);
```

would be changed to:

```c
fprintf(stderr, gettext("panic level %d\n"), lvl);
```

(`gettext` is defined as a no-op if no NLS is configured.)

This tends to add a lot of clutter. One common shortcut is to use:

```c
#define _(x) gettext(x)
```

Another solution is feasible if the program does much of its communication through one or a few functions, such as `ereport()` in the backend. Then you make this function call `gettext` internally on all input strings.

3. Add a file `nls.mk` in the directory with the program sources. This file will be read as a makefile. The following variable assignments need to be made here:

- **CATALOG_NAME**
  - The program name, as provided in the `textdomain()` call.

- **AVAIL_LANGUAGES**
  - List of provided translations — initially empty.

- **GETTEXT_FILES**
  - List of files that contain translatable strings, i.e., those marked with `gettext` or an alternative solution. Eventually, this will include nearly all source files of the program. If this list gets too long you can make the first “file” be a `+` and the second word be a file that contains one file name per line.
The tools that generate message catalogs for the translators to work on need to know what function calls contain translatable strings. By default, only `gettext()` calls are known. If you used `_` or other identifiers you need to list them here. If the translatable string is not the first argument, the item needs to be of the form `func:2` (for the second argument).

The build system will automatically take care of building and installing the message catalogs.

### 47.2.2. Message-writing guidelines

Here are some guidelines for writing messages that are easily translatable.

- Do not construct sentences at run-time, like:

  ```c
  printf("Files were %s.\n", flag ? "copied" : "removed");
  ```

  The word order within the sentence might be different in other languages. Also, even if you remember to call `gettext()` on each fragment, the fragments might not translate well separately. It’s better to duplicate a little code so that each message to be translated is a coherent whole. Only numbers, file names, and such-like run-time variables should be inserted at run time into a message text.

- For similar reasons, this won’t work:

  ```c
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  because it assumes how the plural is formed. If you figured you could solve it like this:

  ```c
  if (n==1)
    printf("copied 1 file");
  else
    printf("copied %d files", n);
  ```

  then be disappointed. Some languages have more than two forms, with some peculiar rules. We might have a solution for this in the future, but for now the matter is best avoided altogether. You could write:

  ```c
  printf("number of copied files: %d", n);
  ```

- If you want to communicate something to the translator, such as about how a message is intended to line up with other output, precede the occurrence of the string with a comment that starts with `translator`, e.g.:

  ```c
  /* translator: This message is not what it seems to be. */
  ```

  These comments are copied to the message catalog files so that the translators can see them.
Chapter 48. Writing A Procedural Language Handler

All calls to functions that are written in a language other than the current “version 1” interface for compiled languages (this includes functions in user-defined procedural languages, functions written in SQL, and functions using the version 0 compiled language interface), go through a call handler function for the specific language. It is the responsibility of the call handler to execute the function in a meaningful way, such as by interpreting the supplied source text. This chapter outlines how a new procedural language’s call handler can be written.

The call handler for a procedural language is a “normal” function that must be written in a compiled language such as C, using the version-1 interface, and registered with PostgreSQL as taking no arguments and returning the type `language_handler`. This special pseudotype identifies the function as a call handler and prevents it from being called directly in SQL commands.

The call handler is called in the same way as any other function: It receives a pointer to a `FunctionCallInfoData` struct containing argument values and information about the called function, and it is expected to return a `Datum` result (and possibly set the `isnull` field of the `FunctionCallInfoData` structure, if it wishes to return an SQL null result). The difference between a call handler and an ordinary callee function is that the `flinfo->fn_oid` field of the `FunctionCallInfoData` structure will contain the OID of the actual function to be called, not of the call handler itself. The call handler must use this field to determine which function to execute. Also, the passed argument list has been set up according to the declaration of the target function, not of the call handler.

It’s up to the call handler to fetch the entry of the function from the system table `pg_proc` and to analyze the argument and return types of the called function. The `AS` clause from the `CREATE FUNCTION` command for the function will be found in the `prosrc` column of the `pg_proc` row. This is commonly source text in the procedural language, but in theory it could be something else, such as a path name to a file, or anything else that tells the call handler what to do in detail.

Often, the same function is called many times per SQL statement. A call handler can avoid repeated lookups of information about the called function by using the `flinfo->fn_extra` field. This will initially be `NULL`, but can be set by the call handler to point at information about the called function. On subsequent calls, if `flinfo->fn_extra` is already non-`NULL` then it can be used and the information lookup step skipped. The call handler must make sure that `flinfo->fn_extra` is made to point at memory that will live at least until the end of the current query, since an `FmgrInfo` data structure could be kept that long. One way to do this is to allocate the extra data in the memory context specified by `flinfo->fn_mcxt`; such data will normally have the same lifespan as the `FmgrInfo` itself. But the handler could also choose to use a longer-lived memory context so that it can cache function definition information across queries.

When a procedural-language function is invoked as a trigger, no arguments are passed in the usual way, but the `FunctionCallInfoData`’s context field points at a `TriggerData` structure, rather than being `NULL` as it is in a plain function call. A language handler should provide mechanisms for procedural-language functions to get at the trigger information.

This is a template for a procedural-language handler written in C:

```c
#include "postgres.h"
#include "executor/spi.h"
```
Chapter 48. Writing A Procedural Language Handler

#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Called as a trigger procedure
         */
        TriggerData *trigdata = (TriggerData *) fcinfo->context;
        retval = ...
    }
    else
    {
        /*
         * Called as a function
         */
        retval = ...
    }

    return retval;
}

Only a few thousand lines of code have to be added instead of the dots to complete the call handler. After having compiled the handler function into a loadable module (see Section 34.9.6), the following commands then register the sample procedural language:

CREATE FUNCTION plsample_call_handler() RETURNS language_handler AS 'filename'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;

The procedural languages included in the standard distribution are good references when trying to write your own call handler. Look into the src/pl subdirectory of the source tree.
49.1. Query Handling as a Complex Optimization Problem

Among all relational operators the most difficult one to process and optimize is the join. The number of possible query plans grows exponentially with the number of joins in the query. Further optimization effort is caused by the support of a variety of join methods (e.g., nested loop, hash join, merge join in PostgreSQL) to process individual joins and a diversity of indexes (e.g., B-tree, hash, GiST and GIN in PostgreSQL) as access paths for relations.

The normal PostgreSQL query optimizer performs a near-exhaustive search over the space of alternative strategies. This algorithm, first introduced in IBM’s System R database, produces a near-optimal join order, but can take an enormous amount of time and memory space when the number of joins in the query grows large. This makes the ordinary PostgreSQL query optimizer inappropriate for queries that join a large number of tables.

The Institute of Automatic Control at the University of Mining and Technology, in Freiberg, Germany, encountered some problems when it wanted to use PostgreSQL as the backend for a decision support knowledge based system for the maintenance of an electrical power grid. The DBMS needed to handle large join queries for the inference machine of the knowledge based system. The number of joins in these queries made using the normal query optimizer infeasible.

In the following we describe the implementation of a genetic algorithm to solve the join ordering problem in a manner that is efficient for queries involving large numbers of joins.

49.2. Genetic Algorithms

The genetic algorithm (GA) is a heuristic optimization method which operates through nondeterministic, randomized search. The set of possible solutions for the optimization problem is considered as a population of individuals. The degree of adaptation of an individual to its environment is specified by its fitness.

The coordinates of an individual in the search space are represented by chromosomes, in essence a set of character strings. A gene is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be binary or integer.

Through simulation of the evolutionary operations recombination, mutation, and selection new generations of search points are found that show a higher average fitness than their ancestors.
According to the comp.ai.genetic FAQ it cannot be stressed too strongly that a GA is not a pure random search for a solution to a problem. A GA uses stochastic processes, but the result is distinctly non-random (better than random).

**Figure 49-1. Structured Diagram of a Genetic Algorithm**

\[
\begin{align*}
P(t) & \quad \text{generation of ancestors at a time } t \\
P'(t) & \quad \text{generation of descendants at a time } t \\
\end{align*}
\]

49.3. Genetic Query Optimization (GEQO) in PostgreSQL

The GEQO module approaches the query optimization problem as though it were the well-known traveling salesman problem (TSP). Possible query plans are encoded as integer strings. Each string represents the join order from one relation of the query to the next. For example, the join tree

```
/\ 2
/ \ 3
 1
```

is encoded by the integer string ’4-1-3-2’, which means, first join relation ’4’ and ’1’, then ’3’, and then ’2’, where 1, 2, 3, 4 are relation IDs within the PostgreSQL optimizer.

Specific characteristics of the GEQO implementation in PostgreSQL are:

- Usage of a steady state GA (replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time;
Chapter 49. Genetic Query Optimizer

- Usage of *edge recombination crossover* which is especially suited to keep edge losses low for the solution of the TSP by means of a GA;
- Mutation as genetic operator is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

Parts of the GEQO module are adapted from D. Whitley’s Genitor algorithm.

The GEQO module allows the PostgreSQL query optimizer to support large join queries effectively through non-exhaustive search.

### 49.3.1. Generating Possible Plans with GEQO

The GEQO planning process uses the standard planner code to generate plans for scans of individual relations. Then join plans are developed using the genetic approach. As shown above, each candidate join plan is represented by a sequence in which to join the base relations. In the initial stage, the GEQO code simply generates some possible join sequences at random. For each join sequence considered, the standard planner code is invoked to estimate the cost of performing the query using that join sequence. (For each step of the join sequence, all three possible join strategies are considered; and all the initially-determined relation scan plans are available. The estimated cost is the cheapest of these possibilities.) Join sequences with lower estimated cost are considered “more fit” than those with higher cost. The genetic algorithm discards the least fit candidates. Then new candidates are generated by combining genes of more-fit candidates — that is, by using randomly-chosen portions of known low-cost join sequences to create new sequences for consideration. This process is repeated until a preset number of join sequences have been considered; then the best one found at any time during the search is used to generate the finished plan.

This process is inherently nondeterministic, because of the randomized choices made during both the initial population selection and subsequent “mutation” of the best candidates. Hence different plans may be selected from one run to the next, resulting in varying run time and varying output row order.

### 49.3.2. Future Implementation Tasks for PostgreSQL GEQO

Work is still needed to improve the genetic algorithm parameter settings. In file `src/backend/optimizer/geqo/geqo_main.c`, routines `gimme_pool_size` and `gimme_number_generations`, we have to find a compromise for the parameter settings to satisfy two competing demands:

- Optimality of the query plan
- Computing time

In the current implementation, the fitness of each candidate join sequence is estimated by running the standard planner’s join selection and cost estimation code from scratch. To the extent that different candidates use similar sub-sequences of joins, a great deal of work will be repeated. This could be made significantly faster by retaining cost estimates for sub-joins. The problem is to avoid expending unreasonable amounts of memory on retaining that state.

At a more basic level, it is not clear that solving query optimization with a GA algorithm designed for TSP is appropriate. In the TSP case, the cost associated with any substring (partial tour) is independent of the rest of the tour, but this is certainly not true for query optimization. Thus it is questionable whether edge recombination crossover is the most effective mutation procedure.
49.4. Further Reading

The following resources contain additional information about genetic algorithms:

- Evolutionary Computation and its application to art and design², by Craig Reynolds
- Fundamentals of Database Systems
- The design and implementation of the POSTGRES query optimizer

¹. [http://www.cs.bham.ac.uk/Mirrors/ftp.de.uu.net/EC/clife/www/location.htm](http://www.cs.bham.ac.uk/Mirrors/ftp.de.uu.net/EC/clife/www/location.htm)
². [http://www.red3d.com/cwr/evolve.html](http://www.red3d.com/cwr/evolve.html)
Chapter 50. Index Access Method Interface

Definition

This chapter defines the interface between the core PostgreSQL system and index access methods, which manage individual index types. The core system knows nothing about indexes beyond what is specified here, so it is possible to develop entirely new index types by writing add-on code.

All indexes in PostgreSQL are what are known technically as secondary indexes; that is, the index is physically separate from the table file that it describes. Each index is stored as its own physical relation and so is described by an entry in the pg_class catalog. The contents of an index are entirely under the control of its index access method. In practice, all index access methods divide indexes into standard-size pages so that they can use the regular storage manager and buffer manager to access the index contents. (All the existing index access methods furthermore use the standard page layout described in Section 53.3, and they all use the same format for index tuple headers; but these decisions are not forced on an access method.)

An index is effectively a mapping from some data key values to tuple identifiers, or TIDs, of row versions (tuples) in the index’s parent table. A TID consists of a block number and an item number within that block (see Section 53.3). This is sufficient information to fetch a particular row version from the table. Indexes are not directly aware that under MVCC, there might be multiple extant versions of the same logical row; to an index, each tuple is an independent object that needs its own index entry. Thus, an update of a row always creates all-new index entries for the row, even if the key values did not change. Index entries for dead tuples are reclaimed (by vacuuming) when the dead tuples themselves are reclaimed.

50.1. Catalog Entries for Indexes

Each index access method is described by a row in the pg_am system catalog (see Section 44.3). The principal contents of a pg_am row are references to pg_proc entries that identify the index access functions supplied by the access method. The APIs for these functions are defined later in this chapter. In addition, the pg_am row specifies a few fixed properties of the access method, such as whether it can support multicolumn indexes. There is not currently any special support for creating or deleting pg_am entries; anyone able to write a new access method is expected to be competent to insert an appropriate row for themselves.

To be useful, an index access method must also have one or more operator families and operator classes defined in pg_opfamily, pg_opclass, pg_amop, and pg_amproc. These entries allow the planner to determine what kinds of query qualifications can be used with indexes of this access method. Operator families and classes are described in Section 34.14, which is prerequisite material for reading this chapter.

An individual index is defined by a pg_class entry that describes it as a physical relation, plus a pg_index entry that shows the logical content of the index — that is, the set of index columns it has and the semantics of those columns, as captured by the associated operator classes. The index columns (key values) can be either simple columns of the underlying table or expressions over the table rows. The index access method normally has no interest in where the index key values come from (it is always handed precomputed key values) but it will be very interested in the operator class.
information in pg_index. Both of these catalog entries can be accessed as part of the Relation data structure that is passed to all operations on the index.

Some of the flag columns of pg_am have nonobvious implications. The requirements of amcanunique are discussed in Section 50.5. The amcanmulticol flag asserts that the access method supports multicol indexes, while amoptionalkey asserts that it allows scans where no indexable restriction clause is given for the first index column. When amcanmulticol is false, amoptionalkey essentially says whether the access method allows full-index scans without any restriction clause. Access methods that support multiple index columns must support scans that omit restrictions on any or all of the columns after the first; however they are permitted to require some restriction to appear for the first index column, and this is signaled by setting amoptionalkey false. amindexnulls asserts that index entries are created for NULL key values. Since most indexable operators are strict and hence cannot return TRUE for NULL inputs, it is at first sight attractive to not store index entries for null values: they could never be returned by an index scan anyway. However, this argument fails when an index scan has no restriction clause for a given index column.

In practice this means that indexes that have amoptionalkey true must index nulls, since the planner might decide to use such an index with no scan keys at all. A related restriction is that an index access method that supports multiple index columns must support indexing null values in columns after the first, because the planner will assume the index can be used for queries that do not restrict these columns. For example, consider an index on (a,b) and a query with WHERE a = 4. The system will assume the index can be used to scan for rows with a = 4, which is wrong if the index omits rows where b is null. It is, however, OK to omit rows where the first indexed column is null. Thus, amindexnulls should be set true only if the index access method indexes all rows, including arbitrary combinations of null values. An index access method that sets amindexnulls may also set amsearchnulls, indicating that it supports IS NULL clauses as search conditions.

50.2. Index Access Method Functions

The index construction and maintenance functions that an index access method must provide are:

```c
IndexBuildResult *
ambuild (Relation heapRelation,
    Relation indexRelation,
    IndexInfo *indexInfo);
```

Build a new index. The index relation has been physically created, but is empty. It must be filled in with whatever fixed data the access method requires, plus entries for all tuples already existing in the table. Ordinarily the ambuild function will call IndexBuildHeapScan() to scan the table for existing tuples and compute the keys that need to be inserted into the index. The function must return a palloc’d struct containing statistics about the new index.

```c
bool
aminsert (Relation indexRelation,
    Datum *values,
    bool *isnull,
    ItemPointer heap_tid,
    Relation heapRelation,
    bool check_uniqueness);
```

Insert a new tuple into an existing index. The values and isnull arrays give the key values to be indexed, and heap_tid is the TID to be indexed. If the access method supports unique indexes (its pg_am.amcanunique flag is true) then check_uniqueness might be true, in which case the
access method must verify that there is no conflicting row; this is the only situation in which the access method normally needs the heapRelation parameter. See Section 50.5 for details. The result is TRUE if an index entry was inserted, FALSE if not. (A FALSE result does not denote an error condition, but is used for cases such as an index AM refusing to index a NULL.)

IndexBulkDeleteResult * ambulkdelete (IndexVacuumInfo *info, 
IndexBulkDeleteResult *stats, 
IndexBulkDeleteCallback callback, 
void *callback_state);

Delete tuple(s) from the index. This is a “bulk delete” operation that is intended to be implemented by scanning the whole index and checking each entry to see if it should be deleted. The passed-in callback function must be called, in the style callback(TID, callback_state) returns bool, to determine whether any particular index entry, as identified by its referenced TID, is to be deleted. Must return either NULL or a palloc’d struct containing statistics about the effects of the deletion operation. It is OK to return NULL if no information needs to be passed on to amvacuumcleanup.

Because of limited maintenance_work_mem, ambulkdelete might need to be called more than once when many tuples are to be deleted. The stats argument is the result of the previous call for this index (it is NULL for the first call within a VACUUM operation). This allows the AM to accumulate statistics across the whole operation. Typically, ambulkdelete will modify and return the same struct if the passed stats is not null.

IndexBulkDeleteResult * amvacuumcleanup (IndexVacuumInfo *info, 
IndexBulkDeleteResult *stats);

Clean up after a VACUUM operation (zero or more ambulkdelete calls). This does not have to do anything beyond returning index statistics, but it might perform bulk cleanup such as reclaiming empty index pages. stats is whatever the last ambulkdelete call returned, or NULL if ambulkdelete was not called because no tuples needed to be deleted. If the result is not NULL it must be a palloc’d struct. The statistics it contains will be used to update pg_class, and will be reported by VACUUM if VERBOSE is given. It is OK to return NULL if the index was not changed at all during the VACUUM operation, but otherwise correct stats should be returned.

void amcostestimate (PlannerInfo *root, 
IndexOptInfo *index, 
List *indexQuals, 
RelOptInfo *outer_rel, 
Cost *indexStartupCost, 
Cost *indexTotalCost, 
Selectivity *indexSelectivity, 
double *indexCorrelation);

Estimate the costs of an index scan. This function is described fully in Section 50.6, below.

bytea * amoptions (ArrayType *reloptions, 
bool validate);
Parse and validate the reloptions array for an index. This is called only when a non-null reloptions array exists for the index. reloptions is a text array containing entries of the form name=value. The function should construct a bytea value, which will be copied into the rd_options field of the index’s relcache entry. The data contents of the bytea value are open for the access method to define, but the standard access methods currently all use struct StdRdOptions. When validate is true, the function should report a suitable error message if any of the options are unrecognized or have invalid values; when validate is false, invalid entries should be silently ignored. (validate is false when loading options already stored in pg_catalog; an invalid entry could only be found if the access method has changed its rules for options, and in that case ignoring obsolete entries is appropriate.) It is OK to return NULL if default behavior is wanted.

The purpose of an index, of course, is to support scans for tuples matching an indexable WHERE condition, often called a qualifier or scan key. The semantics of index scanning are described more fully in Section 50.3, below. The scan-related functions that an index access method must provide are:

IndexScanDesc
ambegin (Relation indexRelation,
   int nkeys,
   ScanKey key);

Begin a new scan. The key array (of length nkeys) describes the scan key(s) for the index scan. The result must be a palloc’d struct. For implementation reasons the index access method must create this struct by calling RelationGetIndexScan(). In most cases ambegin itself does little beyond making that call; the interesting parts of index-scan startup are in amrescan.

boolean
amgettuple (IndexScanDesc scan,
   ScanDirection direction);

Fetch the next tuple in the given scan, moving in the given direction (forward or backward in the index). Returns TRUE if a tuple was obtained, FALSE if no matching tuples remain. In the TRUE case the tuple TID is stored into the scan structure. Note that “success” means only that the index contains an entry that matches the scan keys, not that the tuple necessarily still exists in the heap or will pass the caller’s snapshot test.

boolean
amgetmulti (IndexScanDesc scan,
   ItemPointer tids,
   int32 max_tids,
   int32 *returned_tids);

Fetch multiple tuples in the given scan. Returns TRUE if the scan should continue, FALSE if no matching tuples remain. tids points to a caller-supplied array of max_tids ItemPointerData records, which the call fills with TIDs of matching tuples. *returned_tids is set to the number of TIDs actually returned. This can be less than max_tids, or even zero, even when the return value is TRUE. (This provision allows the access method to choose the most efficient stopping points in its scan, for example index page boundaries.) amgetmulti and amgettuple cannot be used in the same index scan; there are other restrictions too when using amgetmulti, as explained in Section 50.3.

void
amrescan (IndexScanDesc scan,
   ScanKey key);
Chapter 50. Index Access Method Interface Definition

Restart the given scan, possibly with new scan keys (to continue using the old keys, NULL is passed for key). Note that it is not possible for the number of keys to be changed. In practice the restart feature is used when a new outer tuple is selected by a nested-loop join and so a new key comparison value is needed, but the scan key structure remains the same. This function is also called by RelationGetIndexScan(), so it is used for initial setup of an index scan as well as rescanning.

```c
void amendscan (IndexScanDesc scan);
```

End a scan and release resources. The scan struct itself should not be freed, but any locks or pins taken internally by the access method must be released.

```c
void ammarkpos (IndexScanDesc scan);
```

Mark current scan position. The access method need only support one remembered scan position per scan.

```c
void amrestrpos (IndexScanDesc scan);
```

Restore the scan to the most recently marked position.

By convention, the pg_proc entry for an index access method function should show the correct number of arguments, but declare them all as type internal (since most of the arguments have types that are not known to SQL, and we don’t want users calling the functions directly anyway). The return type is declared as void, internal, or boolean as appropriate. The only exception is amoptions, which should be correctly declared as taking text[] and bool and returning bytea. This provision allows client code to execute amoptions to test validity of options settings.

50.3. Index Scanning

In an index scan, the index access method is responsible for regurgitating the TIDs of all the tuples it has been told about that match the scan keys. The access method is not involved in actually fetching those tuples from the index’s parent table, nor in determining whether they pass the scan’s time qualification test or other conditions.

A scan key is the internal representation of a WHERE clause of the form index_key operator constant, where the index key is one of the columns of the index and the operator is one of the members of the operator family associated with that index column. An index scan has zero or more scan keys, which are implicitly ANDed — the returned tuples are expected to satisfy all the indicated conditions.

The operator family can indicate that the index is lossy for a particular operator; this implies that the index scan will return all the entries that pass the scan key, plus possibly additional entries that do not. The core system’s index-scan machinery will then apply that operator again to the heap tuple to verify whether or not it really should be selected. For non-lossy operators, the index scan must return exactly the set of matching entries, as there is no recheck.

Note that it is entirely up to the access method to ensure that it correctly finds all and only the entries passing all the given scan keys. Also, the core system will simply hand off all the WHERE clauses that match the index keys and operator families, without any semantic analysis to determine whether they are redundant or contradictory. As an example, given WHERE x > 4 AND x > 14 where x is
Chapter 50. Index Access Method Interface Definition

A b-tree indexed column, it is left to the b-tree amrescan function to realize that the first scan key is redundant and can be discarded. The extent of preprocessing needed during amrescan will depend on the extent to which the index access method needs to reduce the scan keys to a “normalized” form.

Some access methods return index entries in a well-defined order, others do not. If entries are returned in sorted order, the access method should set pg_am.amcanorder true to indicate that it supports ordered scans. All such access methods must use btree-compatible strategy numbers for their equality and ordering operators.

The amgettuple function has a direction argument, which can be either ForwardScanDirection (the normal case) or BackwardScanDirection. If the first call after amrescan specifies BackwardScanDirection, then the set of matching index entries is to be scanned back-to-front rather than in the normal front-to-back direction, so amgettuple must return the last matching tuple in the index, rather than the first one as it normally would. (This will only occur for access methods that advertise they support ordered scans.) After the first call, amgettuple must be prepared to advance the scan in either direction from the most recently returned entry.

The access method must support “marking” a position in a scan and later returning to the marked position. The same position might be restored multiple times. However, only one position need be remembered per scan; a new ammarkpos call overrides the previously marked position.

Both the scan position and the mark position (if any) must be maintained consistently in the face of concurrent insertions or deletions in the index. It is OK if a freshly-inserted entry is not returned by a scan that would have found the entry if it had existed when the scan started, or for the scan to return such an entry upon rescanning or backing up even though it had not been returned the first time through. Similarly, a concurrent delete might or might not be reflected in the results of a scan. What is important is that insertions or deletions not cause the scan to miss or multiply return entries that were not themselves being inserted or deleted.

Instead of using amgettuple, an index scan can be done with amgetmulti to fetch multiple tuples per call. This can be noticeably more efficient than amgettuple because it allows avoiding lock/unlock cycles within the access method. In principle amgetmulti should have the same effects as repeated amgettuple calls, but we impose several restrictions to simplify matters. In the first place, amgetmulti does not take a direction argument, and therefore it does not support backwards scan nor intrascan reversal of direction. The access method need not support marking or restoring scan positions during an amgetmulti scan, either. (These restrictions cost little since it would be difficult to use these features in an amgetmulti scan anyway: adjusting the caller’s buffered list of TIDs would be complex.) Finally, amgetmulti does not guarantee any locking of the returned tuples, with implications spelled out in Section 50.4.

50.4. Index Locking Considerations

Index access methods must handle concurrent updates of the index by multiple processes. The core PostgreSQL system obtains AccessShareLock on the index during an index scan, and RowExclusiveLock when updating the index (including plain VACUUM). Since these lock types do not conflict, the access method is responsible for handling any fine-grained locking it might need. An exclusive lock on the index as a whole will be taken only during index creation, destruction, REINDEX, or VACUUM FULL.

Building an index type that supports concurrent updates usually requires extensive and subtle analysis of the required behavior. For the b-tree and hash index types, you can read about the design decisions involved in src/backend/access/nbtree/README and src/backend/access/hash/README.
Aside from the index’s own internal consistency requirements, concurrent updates create issues about consistency between the parent table (the heap) and the index. Because PostgreSQL separates accesses and updates of the heap from those of the index, there are windows in which the index might be inconsistent with the heap. We handle this problem with the following rules:

- A new heap entry is made before making its index entries. (Therefore a concurrent index scan is likely to fail to see the heap entry. This is okay because the index reader would be uninterested in an uncommitted row anyway. But see Section 50.5.)
- When a heap entry is to be deleted (by VACUUM), all its index entries must be removed first.
- An index scan must maintain a pin on the index page holding the item last returned by amgettuple, and ambulkdelete cannot delete entries from pages that are pinned by other backends. The need for this rule is explained below.

Without the third rule, it is possible for an index reader to see an index entry just before it is removed by VACUUM, and then to arrive at the corresponding heap entry after that was removed by VACUUM. This creates no serious problems if that item number is still unused when the reader reaches it, since an empty item slot will be ignored by heap_fetch(). But what if a third backend has already reused the item slot for something else? When using an MVCC-compliant snapshot, there is no problem because the new occupant of the slot is certain to be too new to pass the snapshot test. However, with a non-MVCC-compliant snapshot (such as SnapshotNow), it would be possible to accept and return a row that does not in fact match the scan keys. We could defend against this scenario by requiring the scan keys to be rechecked against the heap row in all cases, but that is too expensive. Instead, we use a pin on an index page as a proxy to indicate that the reader might still be “in flight” from the index entry to the matching heap entry. Making ambulkdelete block on such a pin ensures that VACUUM cannot delete the heap entry before the reader is done with it. This solution costs little in run time, and adds blocking overhead only in the rare cases where there actually is a conflict.

This solution requires that index scans be “synchronous”: we have to fetch each heap tuple immediately after scanning the corresponding index entry. This is expensive for a number of reasons. An “asynchronous” scan in which we collect many TIDs from the index, and only visit the heap tuples sometime later, requires much less index locking overhead and can allow a more efficient heap access pattern. Per the above analysis, we must use the synchronous approach for non-MVCC-compliant snapshots, but an asynchronous scan is workable for a query using an MVCC snapshot.

In an amgetmulti index scan, the access method need not guarantee to keep an index pin on any of the returned tuples. (It would be impractical to pin more than the last one anyway.) Therefore it is only safe to use such scans with MVCC-compliant snapshots.

### 50.5. Index Uniqueness Checks

PostgreSQL enforces SQL uniqueness constraints using unique indexes, which are indexes that disallow multiple entries with identical keys. An access method that supports this feature sets pg_am.amcanunique true. (At present, only b-tree supports it.)

Because of MVCC, it is always necessary to allow duplicate entries to exist physically in an index: the entries might refer to successive versions of a single logical row. The behavior we actually want to enforce is that no MVCC snapshot could include two rows with equal index keys. This breaks down into the following cases that must be checked when inserting a new row into a unique index:

- If a conflicting valid row has been deleted by the current transaction, it’s okay. (In particular, since an UPDATE always deletes the old row version before inserting the new version, this will allow an
UPDATE on a row without changing the key.)

- If a conflicting row has been inserted by an as-yet-uncommitted transaction, the would-be inserter must wait to see if that transaction commits. If it rolls back then there is no conflict. If it commits without deleting the conflicting row again, there is a uniqueness violation. (In practice we just wait for the other transaction to end and then redo the visibility check in toto.)

- Similarly, if a conflicting valid row has been deleted by an as-yet-uncommitted transaction, the would-be inserter must wait for that transaction to commit or abort, and then repeat the test.

Furthermore, immediately before raising a uniqueness violation according to the above rules, the access method must recheck the liveness of the row being inserted. If it is committed dead then no error should be raised. (This case cannot occur during the ordinary scenario of inserting a row that’s just been created by the current transaction. It can happen during `CREATE UNIQUE INDEX CONCURRENTLY`, however.)

We require the index access method to apply these tests itself, which means that it must reach into the heap to check the commit status of any row that is shown to have a duplicate key according to the index contents. This is without a doubt ugly and non-modular, but it saves redundant work: if we did a separate probe then the index lookup for a conflicting row would be essentially repeated while finding the place to insert the new row’s index entry. What’s more, there is no obvious way to avoid race conditions unless the conflict check is an integral part of insertion of the new index entry.

The main limitation of this scheme is that it has no convenient way to support deferred uniqueness checks.

### 50.6. Index Cost Estimation Functions

The `amcostestimate` function is given a list of `WHERE` clauses that have been determined to be usable with the index. It must return estimates of the cost of accessing the index and the selectivity of the `WHERE` clauses (that is, the fraction of parent-table rows that will be retrieved during the index scan). For simple cases, nearly all the work of the cost estimator can be done by calling standard routines in the optimizer; the point of having an `amcostestimate` function is to allow index access methods to provide index-type-specific knowledge, in case it is possible to improve on the standard estimates.

Each `amcostestimate` function must have the signature:

```c
void
amcostestimate (PlannerInfo *root,
                 IndexOptInfo *index,
                 List *indexQuals,
                 RelOptInfo *outer_rel,
                 Cost *indexStartupCost,
                 Cost *indexTotalCost,
                 Selectivity *indexSelectivity,
                 double *indexCorrelation);
```

The first four parameters are inputs:

- `root`  
  The planner’s information about the query being processed.
Chapter 50. Index Access Method Interface Definition

index
The index being considered.

indexQuals
List of index qual clauses (implicitly ANDed); a NIL list indicates no qualifiers are available.
Note that the list contains expression trees, not ScanKeys.

outer_rel
If the index is being considered for use in a join inner indexscan, the planner’s information about
the outer side of the join. Otherwise NULL. When non-NULL, some of the qual clauses will
be join clauses with this rel rather than being simple restriction clauses. Also, the cost estimator
should expect that the index scan will be repeated for each row of the outer rel.

The last four parameters are pass-by-reference outputs:

*indexStartupCost
Set to cost of index start-up processing

*indexTotalCost
Set to total cost of index processing

*indexSelectivity
Set to index selectivity

*indexCorrelation
Set to correlation coefficient between index scan order and underlying table’s order

Note that cost estimate functions must be written in C, not in SQL or any available procedural lan-
guage, because they must access internal data structures of the planner/optimizer.

The index access costs should be computed using the parameters used by
src/backend/optimizer/path/costsize.c: a sequential disk block fetch has cost
seq_page_cost, a nonsequential fetch has cost random_page_cost, and the cost of processing
one index row should usually be taken as cpu_index_tuple_cost. In addition, an appropriate
multiple of cpu_operator_cost should be charged for any comparison operators invoked during
index processing (especially evaluation of the indexQuals themselves).

The access costs should include all disk and CPU costs associated with scanning the index itself, but
not the costs of retrieving or processing the parent-table rows that are identified by the index.

The “start-up cost” is the part of the total scan cost that must be expended before we can begin to
fetch the first row. For most indexes this can be taken as zero, but an index type with a high start-up
cost might want to set it nonzero.

The indexSelectivity should be set to the estimated fraction of the parent table rows that will be
retrieved during the index scan. In the case of a lossy index, this will typically be higher than the
fraction of rows that actually pass the given qual conditions.

The indexCorrelation should be set to the correlation (ranging between -1.0 and 1.0) between the
index order and the table order. This is used to adjust the estimate for the cost of fetching rows from
the parent table.

In the join case, the returned numbers should be averages expected for any one scan of the index.
Cost Estimation

A typical cost estimator will proceed as follows:

1. Estimate and return the fraction of parent-table rows that will be visited based on the given qual conditions. In the absence of any index-type-specific knowledge, use the standard optimizer function `clauselist_selectivity()`:

   ```c
   indexSelectivity = clauselist_selectivity(root, indexQuals, index->rel->relid, JOIN_INNER);
   ```

2. Estimate the number of index rows that will be visited during the scan. For many index types this is the same as indexSelectivity times the number of rows in the index, but it might be more.

   (Note that the index’s size in pages and rows is available from the IndexOptInfo struct.)

3. Estimate the number of index pages that will be retrieved during the scan. This might be just indexSelectivity times the index’s size in pages.

4. Compute the index access cost. A generic estimator might do this:

   ```c
   /*
   * Our generic assumption is that the index pages will be read
   * sequentially, so they cost seq_page_cost each, not random_page_cost.
   * Also, we charge for evaluation of the indexquals at each index row.
   * All the costs are assumed to be paid incrementally during the scan.
   */
   cost_qual_eval(&index_qual_cost, indexQuals, root);
   *indexStartupCost = index_qual_cost.startup;
   *indexTotalCost = seq_page_cost * numIndexPages +
                   (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
   ```

   However, the above does not account for amortization of index reads across repeated index scans in the join case.

5. Estimate the index correlation. For a simple ordered index on a single field, this can be retrieved from pg_statistic. If the correlation is not known, the conservative estimate is zero (no correlation).

Examples of cost estimator functions can be found in `src/backend/utils/adt/selfuncs.c`. 
Chapter 51. GiST Indexes

51.1. Introduction

GiST stands for Generalized Search Tree. It is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes. B-trees, R-trees and many other indexing schemes can be implemented in GiST.

One advantage of GiST is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert.

Some of the information here is derived from the University of California at Berkeley’s GiST Indexing Project web site¹ and Marcel Kornacker’s thesis, Access Methods for Next-Generation Database Systems². The GiST implementation in PostgreSQL is primarily maintained by Teodor Sigaev and Oleg Bartunov, and there is more information on their website³.

51.2. Extensibility

Traditionally, implementing a new index access method meant a lot of difficult work. It was necessary to understand the inner workings of the database, such as the lock manager and Write-Ahead Log. The GiST interface has a high level of abstraction, requiring the access method implementer to only implement the semantics of the data type being accessed. The GiST layer itself takes care of concurrency, logging and searching the tree structure.

This extensibility should not be confused with the extensibility of the other standard search trees in terms of the data they can handle. For example, PostgreSQL supports extensible B-trees and hash indexes. That means that you can use PostgreSQL to build a B-tree or hash over any data type you want. But B-trees only support range predicates (<, =, >), and hash indexes only support equality queries.

So if you index, say, an image collection with a PostgreSQL B-tree, you can only issue queries such as “is imagex equal to imagey”, “is imagex less than imagey” and “is imagex greater than imagey”? Depending on how you define “equals”, “less than” and “greater than” in this context, this could be useful. However, by using a GiST based index, you could create ways to ask domain-specific questions, perhaps “find all images of horses” or “find all over-exposed images”.

All it takes to get a GiST access method up and running is to implement seven user-defined methods, which define the behavior of keys in the tree. Of course these methods have to be pretty fancy to support fancy queries, but for all the standard queries (B-trees, R-trees, etc.) they’re relatively straightforward. In short, GiST combines extensibility along with generality, code reuse, and a clean interface.

¹. http://gist.cs.berkeley.edu/
51.3. Implementation

There are seven methods that an index operator class for GiST must provide:

consistent

Given a predicate \( p \) on a tree page, and a user query, \( q \), this method will return false if it is certain that both \( p \) and \( q \) cannot be true for a given data item.

union

This method consolidates information in the tree. Given a set of entries, this function generates a new predicate that is true for all the entries.

compress

Converts the data item into a format suitable for physical storage in an index page.

decompress

The reverse of the \texttt{compress} method. Converts the index representation of the data item into a format that can be manipulated by the database.

penalty

Returns a value indicating the “cost” of inserting the new entry into a particular branch of the tree. Items will be inserted down the path of least \texttt{penalty} in the tree. Values returned by \texttt{penalty} should be non-negative. If a negative value is returned, it will be treated as zero.

picksplit

When a page split is necessary, this function decides which entries on the page are to stay on the old page, and which are to move to the new page.

same

Returns true if two entries are identical, false otherwise.

51.4. Examples

The PostgreSQL source distribution includes several examples of index methods implemented using GiST. The core system currently provides text search support (indexing for \texttt{tsvector} and \texttt{tsquery}) as well as R-Tree equivalent functionality for some of the built-in geometric data types (see \texttt{src/backend/access/gist/gistproc.c}). The following contrib modules also contain GiST operator classes:

\texttt{btree_gist}

B-Tree equivalent functionality for several data types

cube

Indexing for multidimensional cubes

\texttt{hstore}

Module for storing (key, value) pairs

\texttt{intarray}

RD-Tree for one-dimensional array of int4 values
51.5. Crash Recovery

Usually, replay of the WAL log is sufficient to restore the integrity of a GiST index following a database crash. However, there are some corner cases in which the index state is not fully rebuilt. The index will still be functionally correct, but there might be some performance degradation. When this occurs, the index can be repaired by vacuuming its table, or by rebuilding the index using reindex. In some cases a plain vacuum is not sufficient, and either vacuum full or reindex is needed. The need for one of these procedures is indicated by occurrence of this log message during crash recovery:

LOG: index NNN/NNN/NNN needs vacuum or reindex to finish crash recovery

or this log message during routine index insertions:

LOG: index "FOO" needs vacuum or reindex to finish crash recovery

If a plain vacuum finds itself unable to complete recovery fully, it will return a notice:

NOTICE: index "FOO" needs vacuum full or reindex to finish crash recovery
Chapter 52. GIN Indexes

52.1. Introduction

GIN stands for Generalized Inverted Index. It is an index structure storing a set of (key, posting list) pairs, where a “posting list” is a set of rows in which the key occurs. Each indexed value can contain many keys, so the same row ID can appear in multiple posting lists.

It is generalized in the sense that a GIN index does not need to be aware of the operation that it accelerates. Instead, it uses custom strategies defined for particular data types.

One advantage of GIN is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert. This is much the same advantage as using GiST.

The GIN implementation in PostgreSQL is primarily maintained by Teodor Sigaev and Oleg Bartunov. There is more information about GIN on their website.

52.2. Extensibility

The GIN interface has a high level of abstraction, requiring the access method implementer only to implement the semantics of the data type being accessed. The GIN layer itself takes care of concurrency, logging and searching the tree structure.

All it takes to get a GIN access method working is to implement four user-defined methods, which define the behavior of keys in the tree and the relationships between keys, indexed values, and indexable queries. In short, GIN combines extensibility with generality, code reuse, and a clean interface.

The four methods that an index operator class for GIN must provide are:

```c
int compare(Datum a, Datum b)

Compares keys (not indexed values!) and returns an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second.

Datum* extractValue(Datum inputValue, int32 *nkeys)

Returns an array of keys given a value to be indexed. The number of returned keys must be stored into *nkeys.

Datum* extractQuery(Datum query, int32 *nkeys, StrategyNumber n)

Returns an array of keys given a value to be queried; that is, query is the value on the right-hand side of an indexable operator whose left-hand side is the indexed column. n is the strategy number of the operator within the operator class (see Section 34.14.2). Often, extractQuery will need to consult n to determine the data type of query and the key values that need to be extracted. The number of returned keys must be stored into *nkeys. If number of keys is equal to zero then extractQuery should store 0 or -1 into *nkeys. 0 means that any row matches the query and sequence scan should be produced. -1 means nothing can satisfy query. Choice of value should be based on semantics meaning of operation with given strategy number.
```

1. http://www.sai.msu.su/~megera/wiki/Gin
bool consistent(bool check[], StrategyNumber n, Datum query)

Returns TRUE if the indexed value satisfies the query operator with strategy number n (or would satisfy, if the operator is marked RECHECK in the operator class). The check array has the same length as the number of keys previously returned by extractQuery for this query. Each element of the check array is TRUE if the indexed value contains the corresponding query key, ie, if (check[i] == TRUE) the i-th key of the extractQuery result array is present in the indexed value. The original query datum (not the extracted key array!) is passed in case the consistent method needs to consult it.

52.3. Implementation

Internally, a GIN index contains a B-tree index constructed over keys, where each key is an element of the indexed value (a member of an array, for example) and where each tuple in a leaf page is either a pointer to a B-tree over heap pointers (PT, posting tree), or a list of heap pointers (PL, posting list) if the list is small enough.

52.4. GIN tips and tricks

Create vs insert

In most cases, insertion into a GIN index is slow due to the likelihood of many keys being inserted for each value. So, for bulk insertions into a table it is advisable to drop the GIN index and recreate it after finishing bulk insertion.

maintenance_work_mem

Build time for a GIN index is very sensitive to the maintenance_work_mem setting; it doesn’t pay to skimp on work memory during index creation.

gin_fuzzy_search_limit

The primary goal of developing GIN indexes was to create support for highly scalable, full-text search in PostgreSQL, and there are often situations when a full-text search returns a very large set of results. Moreover, this often happens when the query contains very frequent words, so that the large result set is not even useful. Since reading many tuples from the disk and sorting them could take a lot of time, this is unacceptable for production. (Note that the index search itself is very fast.)

To facilitate controlled execution of such queries GIN has a configurable soft upper limit on the number of rows returned, the gin_fuzzy_search_limit configuration parameter. It is set to 0 (meaning no limit) by default. If a non-zero limit is set, then the returned set is a subset of the whole result set, chosen at random.

“Soft” means that the actual number of returned results could differ slightly from the specified limit, depending on the query and the quality of the system’s random number generator.

52.5. Limitations

GIN doesn’t support full index scans: because there are often many keys per value, each heap pointer would be returned many times, and there is no easy way to prevent this.
When `extractQuery` returns zero keys, GIN will emit an error. Depending on the operator, a void query might match all, some, or none of the indexed values (for example, every array contains the empty array, but does not overlap the empty array), and GIN cannot determine the correct answer, nor produce a full-index-scan result if it could determine that that was correct.

It is not an error for `extractValue` to return zero keys, but in this case the indexed value will be unrepresented in the index. This is another reason why full index scan is not useful — it would miss such rows.

GIN searches keys only by equality matching. This might be improved in future.

### 52.6. Examples

The PostgreSQL source distribution includes GIN operator classes for `tsvector` and for one-dimensional arrays of all internal types. The following `contrib` modules also contain GIN operator classes:

- **hstore**
  - Module for storing (key, value) pairs
- **intarray**
  - Enhanced support for `int4[]`
- **pg_trgm**
  - Text similarity using trigram matching
Chapter 53. Database Physical Storage

This chapter provides an overview of the physical storage format used by PostgreSQL databases.

53.1. Database File Layout

This section describes the storage format at the level of files and directories.

All the data needed for a database cluster is stored within the cluster’s data directory, commonly referred to as PGDATA (after the name of the environment variable that can be used to define it). A common location for PGDATA is /var/lib/pgsql/data. Multiple clusters, managed by different server instances, can exist on the same machine.

The PGDATA directory contains several subdirectories and control files, as shown in Table 53-1. In addition to these required items, the cluster configuration files postgresql.conf, pg_hba.conf, and pg_ident.conf are traditionally stored in PGDATA (although in PostgreSQL 8.0 and later, it is possible to keep them elsewhere).

Table 53-1. Contents of PGDATA

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG_VERSION</td>
<td>A file containing the major version number of PostgreSQL</td>
</tr>
<tr>
<td>base</td>
<td>Subdirectory containing per-database subdirectories</td>
</tr>
<tr>
<td>global</td>
<td>Subdirectory containing cluster-wide tables, such as pg_database</td>
</tr>
<tr>
<td>pg_clog</td>
<td>Subdirectory containing transaction commit status data</td>
</tr>
<tr>
<td>pg_multixact</td>
<td>Subdirectory containing multitransaction status data (used for shared row locks)</td>
</tr>
<tr>
<td>pg_subtrans</td>
<td>Subdirectory containing subtransaction status data</td>
</tr>
<tr>
<td>pg_tblspc</td>
<td>Subdirectory containing symbolic links to tablespaces</td>
</tr>
<tr>
<td>pg_twophase</td>
<td>Subdirectory containing state files for prepared transactions</td>
</tr>
<tr>
<td>pg_xlog</td>
<td>Subdirectory containing WAL (Write Ahead Log) files</td>
</tr>
<tr>
<td>postmaster.opts</td>
<td>A file recording the command-line options the server was last started with</td>
</tr>
<tr>
<td>postmaster.pid</td>
<td>A lock file recording the current server PID and shared memory segment ID (not present after server shutdown)</td>
</tr>
</tbody>
</table>

For each database in the cluster there is a subdirectory within PGDATA/base, named after the
Chapter 53. Database Physical Storage

database’s OID in pg_database. This subdirectory is the default location for the database’s files; in particular, its system catalogs are stored there.

Each table and index is stored in a separate file, named after the table or index’s filenode number, which can be found in pg_class.relfilenode.

Caution

Note that while a table’s filenode often matches its OID, this is not necessarily the case; some operations, like TRUNCATE, REINDEX, CLUSTER and some forms of ALTER TABLE, can change the filenode while preserving the OID. Avoid assuming that filenode and table OID are the same.

When a table or index exceeds 1 GB, it is divided into gigabyte-sized segments. The first segment’s file name is the same as the filenode; subsequent segments are named filenode.1, filenode.2, etc. This arrangement avoids problems on platforms that have file size limitations. The contents of tables and indexes are discussed further in Section 53.3.

A table that has columns with potentially large entries will have an associated TOAST table, which is used for out-of-line storage of field values that are too large to keep in the table rows proper. pg_class.reltoastrelid links from a table to its TOAST table, if any. See Section 53.2 for more information.

Tablespaces make the scenario more complicated. Each user-defined tablespace has a symbolic link inside the PGDATA/pg_tblspc directory, which points to the physical tablespace directory (as specified in its CREATE TABLESPACE command). The symbolic link is named after the tablespace’s OID. Inside the physical tablespace directory there is a subdirectory for each database that has elements in the tablespace, named after the database’s OID. Tables within that directory follow the filenode naming scheme. The pg_default tablespace is not accessed through pg_tblspc, but corresponds to PGDATA/base. Similarly, the pg_global tablespace is not accessed through pg_tblspc, but corresponds to PGDATA/global.

Temporary files (for operations such as sorting more data than can fit in memory) are created within PGDATA/base/pgsql_tmp, or within a pgsql_tmp subdirectory of a tablespace directory if a tablespace other than pg_default is specified for them. The name of a temporary file has the form pgsql_tmpPPP.NNN, where PPP is the PID of the owning backend and NNN distinguishes different files of that backend.

53.2. TOAST

This section provides an overview of TOAST (The Oversized-Attribute Storage Technique).

PostgreSQL uses a fixed page size (commonly 8 kB), and does not allow tuples to span multiple pages. Therefore, it is not possible to store very large field values directly. To overcome this limitation, large field values are compressed and/or broken up into multiple physical rows. This happens transparently to the user, with only small impact on most of the backend code. The technique is affectionately known as TOAST (or “the best thing since sliced bread”).

Only certain data types support TOAST — there is no need to impose the overhead on data types that cannot produce large field values. To support TOAST, a data type must have a variable-length (varlena) representation, in which the first 32-bit word of any stored value contains the total length of the value in bytes (including itself). TOAST does not constrain the rest of the representation. All the C-level functions supporting a TOAST-able data type must be careful to handle TOASTed input
values. (This is normally done by invoking `pg_detoast_datum` before doing anything with an input value, but in some cases more efficient approaches are possible.)

TOAST usurps two bits of the varlena length word (the high-order bits on big-endian machines, the low-order bits on little-endian machines), thereby limiting the logical size of any value of a TOAST-able data type to 1 GB ($2^{30} - 1$ bytes). When both bits are zero, the value is an ordinary un-TOASTed value of the data type, and the remaining bits of the length word give the total datum size (including length word) in bytes. When the highest-order or lowest-order bit is set, the value has only a single-byte header instead of the normal four-byte header, and the remaining bits give the total datum size (including length byte) in bytes. As a special case, if the remaining bits are all zero (which would be impossible for a self-inclusive length), the value is a pointer to out-of-line data stored in a separate TOAST table. (The size of a TOAST pointer is given in the second byte of the datum.) Values with single-byte headers aren’t aligned on any particular boundary, either. Lastly, when the highest-order or lowest-order bit is clear but the adjacent bit is set, the content of the datum has been compressed and must be decompressed before use. In this case the remaining bits of the length word give the total size of the compressed datum, not the original data. Note that compression is also possible for out-of-line data but the varlena header does not tell whether it has occurred — the content of the TOAST pointer tells that, instead.

If any of the columns of a table are TOAST-able, the table will have an associated TOAST table, whose OID is stored in the table’s `pg_class.reltoastrelid` entry. Out-of-line TOASTed values are kept in the TOAST table, as described in more detail below.

The compression technique used is a fairly simple and very fast member of the LZ family of compression techniques. See `src/backend/utils/adt/pg_lzcompress.c` for the details.

Out-of-line values are divided (after compression if used) into chunks of at most `TOAST_MAX_CHUNK_SIZE` bytes (by default this value is chosen so that four chunk rows will fit on a page, making it about 2000 bytes). Each chunk is stored as a separate row in the TOAST table for the owning table. Every TOAST table has the columns `chunk_id` (an OID identifying the particular TOASTed value), `chunk_seq` (a sequence number for the chunk within its value), and `chunk_data` (the actual data of the chunk). A unique index on `chunk_id` and `chunk_seq` provides fast retrieval of the values. A pointer datum representing an out-of-line TOASTed value therefore needs to store the OID of the TOAST table in which to look and the OID of the specific value (its `chunk_id`). For convenience, pointer datums also store the logical datum size (original uncompressed data length) and actual stored size (different if compression was applied). Allowing for the varlena header bytes, the total size of a TOAST pointer datum is therefore 18 bytes regardless of the actual size of the represented value.

The TOAST code is triggered only when a row value to be stored in a table is wider than `TOAST_TUPLE_THRESHOLD` bytes (normally 2 kB). The TOAST code will compress and/or move field values out-of-line until the row value is shorter than `TOAST_TUPLE_TARGET` bytes (also normally 2 kB) or no more gains can be had. During an UPDATE operation, values of unchanged fields are normally preserved as-is; so an UPDATE of a row with out-of-line values incurs no TOAST costs if none of the out-of-line values change.

The TOAST code recognizes four different strategies for storing TOAST-able columns:

- **PLAIN** prevents either compression or out-of-line storage; furthermore it disables use of single-byte headers for varlena types. This is the only possible strategy for columns of non-TOAST-able data types.

- **EXTENDED** allows both compression and out-of-line storage. This is the default for most TOAST-able data types. Compression will be attempted first, then out-of-line storage if the row is still too big.
Chapter 53. Database Physical Storage

• **EXTERNAL** allows out-of-line storage but not compression. Use of **EXTERNAL** will make substring operations on wide text and bytea columns faster (at the penalty of increased storage space) because these operations are optimized to fetch only the required parts of the out-of-line value when it is not compressed.

• **MAIN** allows compression but not out-of-line storage. (Actually, out-of-line storage will still be performed for such columns, but only as a last resort when there is no other way to make the row small enough.)

Each TOAST-able data type specifies a default strategy for columns of that data type, but the strategy for a given table column can be altered with `ALTER TABLE SET STORAGE`.

This scheme has a number of advantages compared to a more straightforward approach such as allowing row values to span pages. Assuming that queries are usually qualified by comparisons against relatively small key values, most of the work of the executor will be done using the main row entry. The big values of TOASTed attributes will only be pulled out (if selected at all) at the time the result set is sent to the client. Thus, the main table is much smaller and more of its rows fit in the shared buffer cache than would be the case without any out-of-line storage. Sort sets shrink also, and sorts will more often be done entirely in memory. A little test showed that a table containing typical HTML pages and their URLs was stored in about half of the raw data size including the TOAST table, and that the main table contained only about 10% of the entire data (the URLs and some small HTML pages). There was no run time difference compared to an un-TOASTed comparison table, in which all the HTML pages were cut down to 7 kB to fit.

53.3. Database Page Layout

This section provides an overview of the page format used within PostgreSQL tables and indexes. Sequences and TOAST tables are formatted just like a regular table.

In the following explanation, a byte is assumed to contain 8 bits. In addition, the term item refers to an individual data value that is stored on a page. In a table, an item is a row; in an index, an item is an index entry.

Every table and index is stored as an array of pages of a fixed size (usually 8 kB, although a different page size can be selected when compiling the server). In a table, all the pages are logically equivalent, so a particular item (row) can be stored in any page. In indexes, the first page is generally reserved as a metapage holding control information, and there can be different types of pages within the index, depending on the index access method.

Table 53-2 shows the overall layout of a page. There are five parts to each page.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageHeaderData</td>
<td>24 bytes long. Contains general information about the page, including free space pointers.</td>
</tr>
<tr>
<td>ItemIdData</td>
<td>Array of (offset,length) pairs pointing to the actual items. 4 bytes per item.</td>
</tr>
<tr>
<td>Free space</td>
<td>The unallocated space. New item pointers are allocated from the start of this area, new items from the end.</td>
</tr>
</tbody>
</table>

1. Actually, index access methods need not use this page format. All the existing index methods do use this basic format, but the data kept on index metapages usually doesn’t follow the item layout rules.
The first 24 bytes of each page consists of a page header (PageHeaderData). Its format is detailed in Table 53-3. The first two fields track the most recent WAL entry related to this page. Next is a 2-byte field containing flag bits. This is followed by three 2-byte integer fields (pd_lower, pd_upper, and pd_special). These contain byte offsets from the page start to the start of unallocated space, to the end of unallocated space, and to the start of the special space. The next 2 bytes of the page header, pd_pagesize_version, store both the page size and a version indicator. Beginning with PostgreSQL 8.3 the version number is 4; PostgreSQL 8.1 and 8.2 used version number 3; PostgreSQL 8.0 used version number 2; PostgreSQL 7.3 and 7.4 used version number 1; prior releases used version number 0. (The basic page layout and header format has not changed in most of these versions, but the layout of heap row headers has.) The page size is basically only present as a cross-check; there is no support for having more than one page size in an installation. The last field is a hint that shows whether pruning the page is likely to be profitable: it tracks the oldest un-pruned XMAX on the page.

Table 53-3. PageHeaderData Layout

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pd_lsn</td>
<td>XLogRecPtr</td>
<td>8 bytes</td>
<td>LSN: next byte after last byte of xlog record for last change to this page</td>
</tr>
<tr>
<td>pd_tli</td>
<td>uint16</td>
<td>2 bytes</td>
<td>TimeLineID of last change (only its lowest 16 bits)</td>
</tr>
<tr>
<td>pd_flags</td>
<td>uint16</td>
<td>2 bytes</td>
<td>Flag bits</td>
</tr>
<tr>
<td>pd_lower</td>
<td>LocationIndex</td>
<td>2 bytes</td>
<td>Offset to start of free space</td>
</tr>
<tr>
<td>pd_upper</td>
<td>LocationIndex</td>
<td>2 bytes</td>
<td>Offset to end of free space</td>
</tr>
<tr>
<td>pd_special</td>
<td>LocationIndex</td>
<td>2 bytes</td>
<td>Offset to start of special space</td>
</tr>
<tr>
<td>pd_pagesize_version</td>
<td>uint16</td>
<td>2 bytes</td>
<td>Page size and layout version number information</td>
</tr>
<tr>
<td>pd_prune_xid</td>
<td>TransactionId</td>
<td>4 bytes</td>
<td>Oldest unpruned XMAX on page, or zero if none</td>
</tr>
</tbody>
</table>

All the details can be found in src/include/storage(bufpage.h).

Following the page header are item identifiers (ItemIdData), each requiring four bytes. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a few attribute bits which affect its interpretation. New item identifiers are allocated as needed from the beginning of the unallocated space. The number of item identifiers present can be determined by looking at pd_lower, which is increased to allocate a new identifier. Because an item identifier is never moved until it is
freed, its index can be used on a long-term basis to reference an item, even when the item itself is moved around on the page to compact free space. In fact, every pointer to an item (ItemPointer, also known as CTID) created by PostgreSQL consists of a page number and the index of an item identifier.

The items themselves are stored in space allocated backwards from the end of unallocated space. The exact structure varies depending on what the table is to contain. Tables and sequences both use a structure named HeapTupleHeaderData, described below.

The final section is the “special section” which can contain anything the access method wishes to store. For example, b-tree indexes store links to the page’s left and right siblings, as well as some other data relevant to the index structure. Ordinary tables do not use a special section at all (indicated by setting pd_special to equal the page size).

All table rows are structured in the same way. There is a fixed-size header (occupying 23 bytes on most machines), followed by an optional null bitmap, an optional object ID field, and the user data. The header is detailed in Table 53-4. The actual user data (columns of the row) begins at the offset indicated by t_hoff, which must always be a multiple of the MAXALIGN distance for the platform. The null bitmap is only present if the HEAP_HASNULL bit is set in t_infomask. If it is present it begins just after the fixed header and occupies enough bytes to have one bit per data column (that is, t_natts bits altogether). In this list of bits, a 1 bit indicates not-null, a 0 bit is a null. When the bitmap is not present, all columns are assumed not-null. The object ID is only present if the HEAP_HASOID bit is set in t_infomask. If present, it appears just before the t_hoff boundary. Any padding needed to make t_hoff a MAXALIGN multiple will appear between the null bitmap and the object ID. (This in turn ensures that the object ID is suitably aligned.)

Table 53-4. HeapTupleHeaderData Layout

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_xmin</td>
<td>TransactionId</td>
<td>4 bytes</td>
<td>insert XID stamp</td>
</tr>
<tr>
<td>t_xmax</td>
<td>TransactionId</td>
<td>4 bytes</td>
<td>delete XID stamp</td>
</tr>
<tr>
<td>t_cid</td>
<td>CommandId</td>
<td>4 bytes</td>
<td>insert and/or delete CID stamp (overlays with t_xvac)</td>
</tr>
<tr>
<td>t_xvac</td>
<td>TransactionId</td>
<td>4 bytes</td>
<td>XID for VACUUM operation moving a row version</td>
</tr>
<tr>
<td>t_ctid</td>
<td>ItemPointerData</td>
<td>6 bytes</td>
<td>current TID of this or newer row version</td>
</tr>
<tr>
<td>t_infomask2</td>
<td>int16</td>
<td>2 bytes</td>
<td>number of attributes, plus various flag bits</td>
</tr>
<tr>
<td>t_infomask</td>
<td>uint16</td>
<td>2 bytes</td>
<td>various flag bits</td>
</tr>
<tr>
<td>t_hoff</td>
<td>uint8</td>
<td>1 byte</td>
<td>offset to user data</td>
</tr>
</tbody>
</table>

All the details can be found in src/include/access/htup.h.

Interpreting the actual data can only be done with information obtained from other tables, mostly pg_attribute. The key values needed to identify field locations are attlen and attalign. There is no way to directly get a particular attribute, except when there are only fixed width fields and no null values. All this trickery is wrapped up in the functions heap_getattr, fastgetattr and heap_getsysattr. To read the data you need to examine each attribute in turn. First check whether the field is NULL according to the null bitmap. If it is, go to the next. Then make sure you have the right alignment.
If the field is a fixed width field, then all the bytes are simply placed. If it’s a variable length field (attlen = -1) then it’s a bit more complicated. All variable-length datatypes share the common header *struct varlena*, which includes the total length of the stored value and some flag bits. Depending on the flags, the data can be either inline or in a TOAST table; it might be compressed, too (see Section 53.2).
Chapter 54. BKI Backend Interface

Backend Interface (BKI) files are scripts in a special language that is understood by the PostgreSQL backend when running in the “bootstrap” mode. The bootstrap mode allows system catalogs to be created and filled from scratch, whereas ordinary SQL commands require the catalogs to exist already. BKI files can therefore be used to create the database system in the first place. (And they are probably not useful for anything else.)

initdb uses a BKI file to do part of its job when creating a new database cluster. The input file used by initdb is created as part of building and installing PostgreSQL by a program named genbki.sh, which reads some specially formatted C header files in the src/include/catalog/ directory of the source tree. The created BKI file is called postgres.bki and is normally installed in the share subdirectory of the installation tree.

Related information can be found in the documentation for initdb.

54.1. BKI File Format

This section describes how the PostgreSQL backend interprets BKI files. This description will be easier to understand if the postgres.bki file is at hand as an example.

BKI input consists of a sequence of commands. Commands are made up of a number of tokens, depending on the syntax of the command. Tokens are usually separated by whitespace, but need not be if there is no ambiguity. There is no special command separator; the next token that syntactically cannot belong to the preceding command starts a new one. (Usually you would put a new command on a new line, for clarity.) Tokens can be certain key words, special characters (parentheses, commas, etc.), numbers, or double-quoted strings. Everything is case sensitive.

Lines starting with # are ignored.

54.2. BKI Commands

create [bootstrap] [shared_relation] [without_oids] tablename tableoid (name1 = type1 [, name2 = type2, ...])

Create a table named tablename, and having the OID tableoid, with the columns given in parentheses.

The following column types are supported directly by bootstrap.c: bool, bytea, char (1 byte), name, int2, int4, regproc, regclass, regtype, text, oid, tid, xid, cid, int2vector, oidvector, _int4 (array), _text (array), _oid (array), _char (array), _aclitem (array). Although it is possible to create tables containing columns of other types, this cannot be done until after pg_type has been created and filled with appropriate entries. (That effectively means that only these column types can be used in bootstrapped tables, but non-bootstrap catalogs can contain any built-in type.)

When bootstrap is specified, the table will only be created on disk; nothing is entered into pg_class, pg_attribute, etc. for it. Thus the table will not be accessible by ordinary SQL
operations until such entries are made the hard way (with insert commands). This option is used for creating pg_class etc themselves.

The table is created as shared if shared_relation is specified. It will have OIDs unless without_oids is specified.

open tablename

Open the table named tablename for insertion of data. Any currently open table is closed.

close [tablename]

Close the open table. The name of the table can be given as a cross-check, but this is not required.

insert [OID = oid_value] (value1 value2 ...)

Insert a new row into the open table using value1, value2, etc., for its column values and oid_value for its OID. If oid_value is zero (0) or the clause is omitted, and the table has OIDs, then the next available OID is assigned.

NULL values can be specified using the special key word _null_. Values containing spaces must be double quoted.

declare [unique] index indexname indexoid on tablename using amname (opclass1 name1 [, ...])

Create an index named indexname, having OID indexoid, on the table named tablename, using the amname access method. The fields to index are called name1, name2 etc., and the operator classes to use are opclass1, opclass2 etc., respectively. The index file is created and appropriate catalog entries are made for it, but the index contents are not initialized by this command.

declare toast toasttableoid toastindexoid on tablename

Create a TOAST table for the table named tablename. The TOAST table is assigned OID toasttableoid and its index is assigned OID toastindexoid. As with declare index, filling of the index is postponed.

build indices

Fill in the indices that have previously been declared.

54.3. Structure of the Bootstrap BKI File

The open command cannot be used until the tables it uses exist and have entries for the table that is to be opened. (These minimum tables are pg_class, pg_attribute, pg_proc, and pg_type.) To allow those tables themselves to be filled, create with the bootstrap option implicitly opens the created table for data insertion.

Also, the declare index and declare toast commands cannot be used until the system catalogs they need have been created and filled in.

Thus, the structure of the postgres.bki file has to be:

1. create bootstrap one of the critical tables
2. insert data describing at least the critical tables
3. close
4. Repeat for the other critical tables.
5. create (without bootstrap) a noncritical table

6. open

7. insert desired data

8. close

9. Repeat for the other noncritical tables.

10. Define indexes and toast tables.

11. build indices

There are doubtless other, undocumented ordering dependencies.

### 54.4. Example

The following sequence of commands will create the table test_table with OID 420, having two columns cola and colb of type int4 and text, respectively, and insert two rows into the table:

```sql
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```
Chapter 55. How the Planner Uses Statistics

This chapter builds on the material covered in Section 14.1 and Section 14.2 to show some additional details about how the planner uses the system statistics to estimate the number of rows each part of a query might return. This is a significant part of the planning process, providing much of the raw material for cost calculation.

The intent of this chapter is not to document the code in detail, but to present an overview of how it works. This will perhaps ease the learning curve for someone who subsequently wishes to read the code.

55.1. Row Estimation Examples

The examples shown below use tables in the PostgreSQL regression test database. The outputs shown are taken from version 8.3. The behavior of earlier (or later) versions might vary. Note also that since ANALYZE uses random sampling while producing statistics, the results will change slightly after any new ANALYZE.

Let's start with a very simple query:

```
EXPLAIN SELECT * FROM tenk1;
```

```
QUERY PLAN
-------------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)

How the planner determines the cardinality of tenk1 is covered in Section 14.2, but is repeated here for completeness. The number of pages and rows is looked up in pg_class:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

```
relpages | reltuples
----------+-----------
358 | 10000
```

These numbers are current as of the last VACUUM or ANALYZE on the table. The planner then fetches the actual current number of pages in the table (this is a cheap operation, not requiring a table scan). If that is different from relpages then reltuples is scaled accordingly to arrive at a current number-of-rows estimate. In this case the values are correct so the rows estimate is the same as reltuples.

Let's move on to an example with a range condition in its WHERE clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
```

```
QUERY PLAN
--------------------------------------------------------------------------------
Bitmap Heap Scan on tenk1 (cost=24.06..394.64 rows=1007 width=244)
  Recheck Cond: (unique1 < 1000)
  -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..23.80 rows=1007 width=0)
    Index Cond: (unique1 < 1000)
```
Chapter 55. How the Planner Uses Statistics

The planner examines the WHERE clause condition and looks up the selectivity function for the operator < in pg_operator. This is held in the column oprrest, and the entry in this case is scalarltsel. The scalarltsel function retrieves the histogram for uniquel from pg_statistics. For manual queries it is more convenient to look in the simpler pg_stats view:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='uniquel';
```

```
histogram_bounds
-----------------------------------------
(0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995)
```

Next the fraction of the histogram occupied by “< 1000” is worked out. This is the selectivity. The histogram divides the range into equal frequency buckets, so all we have to do is locate the bucket that our value is in and count part of it and all of the ones before. The value 1000 is clearly in the second bucket (993-1997). Assuming a linear distribution of values inside each bucket, we can calculate the selectivity as:

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
= (1 + (1000 - 993)/(1997 - 993))/10
= 0.100697
```

that is, one whole bucket plus a linear fraction of the second, divided by the number of buckets. The estimated number of rows can now be calculated as the product of the selectivity and the cardinality of tenk1:

```
rows = rel_cardinality * selectivity
= 10000 * 0.100697
= 1007 (rounding off)
```

Next let’s consider an example with an equality condition in its WHERE clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';
```

```
QUERY PLAN
----------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..483.00 rows=30 width=244)
  Filter: (stringu1 = 'CRAAAA '::name)
```

Again the planner examines the WHERE clause condition and looks up the selectivity function for =, which is eqsel. For equality estimation the histogram is not useful; instead the list of most common values (MCVs) is used to determine the selectivity. Let’s have a look at the MCVs, with some additional columns that will be useful later:

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';
```

```
null_frac | 0
n_distinct | 676
most_common_vals | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSSAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

Since CRAAAA appears in the list of MCVs, the selectivity is merely the corresponding entry in the list of most common frequencies (MCFs):
Chapter 55. How the Planner Uses Statistics

selectivity = mcf[3]
  = 0.003

As before, the estimated number of rows is just the product of this with the cardinality of tenk1:

rows = 10000 * 0.003
  = 30

Now consider the same query, but with a constant that is not in the MCV list:

EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

QUERY PLAN
----------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..483.00 rows=15 width=244)
  Filter: (stringu1 = 'xxx'::name)

This is quite a different problem: how to estimate the selectivity when the value is not in the MCV list. The approach is to use the fact that the value is not in the list, combined with the knowledge of the frequencies for all of the MCVs:

\[ \text{selectivity} = \frac{1 - \sum \text{mvf}}{\text{num\_distinct} - \text{num\_mcv}} \]

\[ = \frac{1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003)}{676 - 10} \]

\[ = 0.0014559 \]

That is, add up all the frequencies for the MCVs and subtract them from one, then divide by the number of other distinct values. This amounts to assuming that the fraction of the column that is not any of the MCVs is evenly distributed among all the other distinct values. Notice that there are no null values so we don’t have to worry about those (otherwise we’d subtract the null fraction from the numerator as well). The estimated number of rows is then calculated as usual:

\[ \text{rows} = 10000 \times 0.0014559 \]

\[ = 15 \] (rounding off)

The previous example with \texttt{unique1 < 1000} was an oversimplification of what \texttt{scalarltsel} really does; now that we have seen an example of the use of MCVs, we can fill in some more detail. The example was correct as far as it went, because since \texttt{unique1} is a unique column it has no MCVs (obviously, no value is any more common than any other value). For a non-unique column, there will normally be both a histogram and an MCV list, and the histogram does not include the portion of the column population represented by the MCVs. We do things this way because it allows more precise estimation. In this situation \texttt{scalarltsel} directly applies the condition (e.g., “< 1000”) to each value of the MCV list, and adds up the frequencies of the MCVs for which the condition is true. This gives an exact estimate of the selectivity within the portion of the table that is MCVs. The histogram is then used in the same way as above to estimate the selectivity in the portion of the table that is not MCVs, and then the two numbers are combined to estimate the overall selectivity. For example, consider

EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

QUERY PLAN
------------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..483.00 rows=3077 width=244)
Filter: (stringul < 'IAAAAA'::name)

We already saw the MCV information for stringul, and here is its histogram:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringul';
```

```
histogram_bounds
--------------------------------------------------------------------------------
(AAAAAA, CQAAAA, FRAAAA, IBAAAA, KRAAAA, NFAAAA, PSAAAA, SGAAAA, VAAAAA, XLAAAA, ZZAAAA)
```

Checking the MCV list, we find that the condition stringul < 'IAAAAA' is satisfied by the first six entries and not the last four, so the selectivity within the MCV part of the population is

\[
selectivity = \sum \text{relevant mvfs}
= 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
= 0.01833333
\]

Summing all the MCFs also tells us that the total fraction of the population represented by MCVs is 0.03033333, and therefore the fraction represented by the histogram is 0.96966667 (again, there are no nulls, else we’d have to exclude them here). We can see that the value IAAAAA falls nearly at the end of the third histogram bucket. Using some rather cheesy assumptions about the frequency of different characters, the planner arrives at the estimate 0.298387 for the portion of the histogram population that is less than IAAAAA. We then combine the estimates for the MCV and non-MCV populations:

\[
selectivity = \text{mcv selectivity} + \text{histogram selectivity} \times \text{histogram fraction}
= 0.01833333 + 0.298387 \times 0.96966667
= 0.307669
\]

\[
\text{rows} = 10000 \times 0.307669
= 3077 \text{ (rounding off)}
\]

In this particular example, the correction from the MCV list is fairly small, because the column distribution is actually quite flat (the statistics showing these particular values as being more common than others are mostly due to sampling error). In a more typical case where some values are significantly more common than others, this complicated process gives a useful improvement in accuracy because the selectivity for the most common values is found exactly.

Now let’s consider a case with more than one condition in the `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringul = 'xxx';
```

```
QUERY PLAN
--------------------------------------------------------------------------------
Bitmap Heap Scan on tenk1 (cost=23.80..396.91 rows=1 width=244)
  Recheck Cond: (unique1 < 1000)
  Filter: (stringul = 'xxx'::name)
    -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..23.80 rows=1007 width=0)
      Index Cond: (unique1 < 1000)
```

The planner assumes that the two conditions are independent, so that the individual selectivities of the clauses can be multiplied together:

\[
selectivity = \text{selectivity(unique1 < 1000)} \times \text{selectivity(stringul = 'xxx')}
= 0.100697 \times 0.0014559
= 0.0001466
\]
Chapter 55. How the Planner Uses Statistics

\[
\text{rows} = 10000 \times 0.0001466 = 1 \text{ (rounding off)}
\]

Notice that the number of rows estimated to be returned from the bitmap index scan reflects only the condition used with the index; this is important since it affects the cost estimate for the subsequent heap fetches.

Finally we will examine a query that involves a join:

**EXPLAIN**

\[
\text{SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;}
\]

**QUERY PLAN**

```
+-------------------------------------------+---------------------------+-----------------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
| Nested Loop (cost=4.64..456.23 rows=50 width=488) |
| -> Bitmap Heap Scan on tenk1 t1 (cost=4.64..142.17 rows=50 width=244) |
| | Recheck Cond: (unique1 < 50) |
| | -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.63 rows=50 width=0) |
| | | Index Cond: (unique1 < 50) |
| | -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.00..6.27 rows=1 width=244) |
| | | Index Cond: (t2.unique2 = t1.unique2) |
```

The restriction on `tenk1, unique1 < 50`, is evaluated before the nested-loop join. This is handled analogously to the previous range example. This time the value 50 falls into the first bucket of the `unique1` histogram:

\[
\text{selectivity} = \frac{(0 + (50 - \text{bucket}[1].\text{min})/(\text{bucket}[1].\text{max} - \text{bucket}[1].\text{min}))}{\text{num_buckets}}
\]

\[
= \frac{(0 + (50 - 0)/(993 - 0))}{10} = 0.005035
\]

\[
\text{rows} = 10000 \times 0.005035 = 50 \text{ (rounding off)}
\]

The restriction for the join is `t2.unique2 = t1.unique2`. The operator is just our familiar `=`, however the selectivity function is obtained from the `oprjoin` column of `pg_operator`, and is `eqjoinsel`. `eqjoinsel` looks up the statistical information for both `tenk2` and `tenk1`:

**SELECT**

\[
\text{SELECT tablename, null_frac, n_distinct, most_common_vals FROM pg_stats WHERE tablename IN (‘tenk1’, ‘tenk2’) AND attname=’unique2’;}
\]

<table>
<thead>
<tr>
<th>tablename</th>
<th>null_frac</th>
<th>n_distinct</th>
<th>most_common_vals</th>
</tr>
</thead>
<tbody>
<tr>
<td>tenk1</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>tenk2</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

In this case there is no MCV information for `unique2` because all the values appear to be unique, so we use an algorithm that relies only on the number of distinct values for both relations together with their null fractions:

\[
\text{selectivity} = (1 - \text{null_fracl}) \times (1 - \text{null_fracl2}) \times \min(1/\text{n_distinct1}, 1/\text{n_distinct2})
\]

\[
= (1 - 0) \times (1 - 0) / \max(10000, 10000) = 0.0001
\]

This is, subtract the null fraction from one for each of the relations, and divide by the maximum of the numbers of distinct values. The number of rows that the join is likely to emit is calculated as the cardinality of the Cartesian product of the two inputs, multiplied by the selectivity:
Chapter 55. How the Planner Uses Statistics

\[
\text{rows} = (\text{outer_cardinality} \times \text{inner_cardinality}) \times \text{selectivity} \\
= (50 \times 10000) \times 0.0001 \\
= 50
\]

Had there been MCV lists for the two columns, eqjoinsel would have used direct comparison of the MCV lists to determine the join selectivity within the part of the column populations represented by the MCVs. The estimate for the remainder of the populations follows the same approach shown here.

Notice that we showed \text{inner_cardinality} as 10000, that is, the unmodified size of \text{tenk2}. It might appear from inspection of the \text{EXPLAIN} output that the estimate of join rows comes from 50 \times 1, that is, the number of outer rows times the estimated number of rows obtained by each inner indexscan on \text{tenk2}. But this is not the case: the join relation size is estimated before any particular join plan has been considered. If everything is working well then the two ways of estimating the join size will produce about the same answer, but due to roundoff error and other factors they sometimes diverge significantly.

For those interested in further details, estimation of the size of a table (before any \text{WHERE} clauses) is done in \text{src/backend/optimizer/util/plancat.c}. The generic logic for clause selectivities is in \text{src/backend/optimizer/path/clausesel.c}. The operator-specific selectivity functions are mostly found in \text{src/backend/utils/adt/selfuncs.c}. 
VIII. Appendixes
Appendix A. PostgreSQL Error Codes

All messages emitted by the PostgreSQL server are assigned five-character error codes that follow the SQL standard’s conventions for “SQLSTATE” codes. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message. The error codes are less likely to change across PostgreSQL releases, and also are not subject to change due to localization of error messages. Note that some, but not all, of the error codes produced by PostgreSQL are defined by the SQL standard; some additional error codes for conditions not defined by the standard have been invented or borrowed from other databases.

According to the standard, the first two characters of an error code denote a class of errors, while the last three characters indicate a specific condition within that class. Thus, an application that does not recognize the specific error code can still be able to infer what to do from the error class.

Table A-1 lists all the error codes defined in PostgreSQL 8.3.23. (Some are not actually used at present, but are defined by the SQL standard.) The error classes are also shown. For each error class there is a “standard” error code having the last three characters 000. This code is used only for error conditions that fall within the class but do not have any more-specific code assigned.

The PL/pgSQL condition name for each error code is the same as the phrase shown in the table, with underscores substituted for spaces. For example, code 22012, DIVISION BY ZERO, has condition name DIVISION_BY_ZERO. Condition names can be written in either upper or lower case. (Note that PL/pgSQL does not recognize warning, as opposed to error, condition names; those are classes 00, 01, and 02.)

Table A-1. PostgreSQL Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>SUCCESSFUL COMPLETION</td>
<td>successful_completion</td>
</tr>
<tr>
<td>01000</td>
<td>WARNING</td>
<td>warning</td>
</tr>
<tr>
<td>0100C</td>
<td>DYNAMIC RESULT SETS RETURNED</td>
<td>dynamic_result_sets_returned</td>
</tr>
<tr>
<td>01008</td>
<td>IMPLICIT ZERO BIT PADDING</td>
<td>implicit_zero_bit_padding</td>
</tr>
<tr>
<td>01003</td>
<td>NULL VALUE ELIMINATED IN SET FUNCTION</td>
<td>null_value_eliminated_in_set_function</td>
</tr>
<tr>
<td>01007</td>
<td>PRIVILEGE NOT GRANTED</td>
<td>privilege_not_granted</td>
</tr>
<tr>
<td>01006</td>
<td>PRIVILEGE NOT REVOKED</td>
<td>privilege_not_revoked</td>
</tr>
<tr>
<td>01004</td>
<td>STRING DATA RIGHT TRUNCATION</td>
<td>string_data_right_truncation</td>
</tr>
<tr>
<td>01001</td>
<td>DEPRECATED FEATURE</td>
<td>deprecated_feature</td>
</tr>
<tr>
<td>02000</td>
<td>NO DATA</td>
<td>no_data</td>
</tr>
<tr>
<td>02001</td>
<td>NO ADDITIONAL DYNAMIC RESULT SETS RETURNED</td>
<td>no_additional_dynamic_result_sets_returned</td>
</tr>
</tbody>
</table>

Class 03 — SQL Statement Not Yet Complete
### Appendix A. PostgreSQL Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>03000</td>
<td>SQL STATEMENT NOT YET COMPLETE</td>
<td>sql_statement_not_yet_complete</td>
</tr>
<tr>
<td><strong>Class 08 — Connection Exception</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08000</td>
<td>CONNECTION EXCEPTION</td>
<td>connection_exception</td>
</tr>
<tr>
<td>08003</td>
<td>CONNECTION DOES NOT EXIST</td>
<td>connection_does_not_exist</td>
</tr>
<tr>
<td>08006</td>
<td>CONNECTION FAILURE</td>
<td>connection_failure</td>
</tr>
<tr>
<td>08001</td>
<td>SQLCLIENT UNABLE TO ESTABLISH SQLCONNECTION</td>
<td>sqlclient_unable_to_establish_sqlconnection</td>
</tr>
<tr>
<td>08004</td>
<td>SQLSERVER REJECTED ESTABLISHMENT OF SQLCONNECTION</td>
<td>sqlserver_rejected_establishment_of_sqlconnection</td>
</tr>
<tr>
<td>08007</td>
<td>TRANSACTION RESOLUTION UNKNOWN</td>
<td>transaction_resolution_unknown</td>
</tr>
<tr>
<td>08P01</td>
<td>PROTOCOL VIOLATION</td>
<td>protocol_violation</td>
</tr>
<tr>
<td><strong>Class 09 — Triggered Action Exception</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>09000</td>
<td>TRIGGERED ACTION EXCEPTION</td>
<td>triggered_action_exception</td>
</tr>
<tr>
<td><strong>Class 0A — Feature Not Supported</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0A000</td>
<td>FEATURE NOT SUPPORTED</td>
<td>feature_not_supported</td>
</tr>
<tr>
<td><strong>Class 0B — Invalid Transaction Initiation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0B000</td>
<td>INVALID TRANSACTION INITIATION</td>
<td>invalid_transaction_initiation</td>
</tr>
<tr>
<td><strong>Class 0F — Locator Exception</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F000</td>
<td>LOCATOR EXCEPTION</td>
<td>locator_exception</td>
</tr>
<tr>
<td>0F001</td>
<td>INVALID LOCATOR SPECIFICATION</td>
<td>invalid_locator_specification</td>
</tr>
<tr>
<td><strong>Class 0L — Invalid Grantor</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0L000</td>
<td>INVALID GRANTOR</td>
<td>invalid_grantor</td>
</tr>
<tr>
<td>0LP01</td>
<td>INVALID GRANT OPERATION</td>
<td>invalid_grant_operation</td>
</tr>
<tr>
<td><strong>Class 0P — Invalid Role Specification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0P000</td>
<td>INVALID ROLE SPECIFICATION</td>
<td>invalid_role_specification</td>
</tr>
<tr>
<td><strong>Class 21 — Cardinality Violation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21000</td>
<td>CARDINALITY VIOLATION</td>
<td>cardinality_violation</td>
</tr>
<tr>
<td><strong>Class 22 — Data Exception</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22000</td>
<td>DATA EXCEPTION</td>
<td>data_exception</td>
</tr>
<tr>
<td>2202E</td>
<td>ARRAY SUBSCRIPT ERROR</td>
<td>array_subscript_error</td>
</tr>
<tr>
<td>22021</td>
<td>CHARACTER NOT IN REPERTOIRE</td>
<td>character_not_in_repertoire</td>
</tr>
<tr>
<td>22008</td>
<td>DATETIME FIELD OVERFLOW</td>
<td>datetime_field_overflow</td>
</tr>
</tbody>
</table>
## Appendix A. PostgreSQL Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>22012</td>
<td>DIVISION BY ZERO</td>
<td>division_by_zero</td>
</tr>
<tr>
<td>22005</td>
<td>ERROR IN ASSIGNMENT</td>
<td>error_in_assignment</td>
</tr>
<tr>
<td>2200B</td>
<td>ESCAPE CHARACTER CONFLICT</td>
<td>escape_character_conflict</td>
</tr>
<tr>
<td>22022</td>
<td>INDICATOR OVERFLOW</td>
<td>indicator_overflow</td>
</tr>
<tr>
<td>22015</td>
<td>INTERVAL FIELD OVERFLOW</td>
<td>interval_field_overflow</td>
</tr>
<tr>
<td>2201E</td>
<td>INVALID ARGUMENT FOR LOGARITHM</td>
<td>invalid_argument_for_logarithm</td>
</tr>
<tr>
<td>2201F</td>
<td>INVALID ARGUMENT FOR POWER FUNCTION</td>
<td>invalid_argument_for_power_function</td>
</tr>
<tr>
<td>2201G</td>
<td>INVALID ARGUMENT FOR WIDTH BUCKET FUNCTION</td>
<td>invalid_argument_for_width_bucket_function</td>
</tr>
<tr>
<td>22018</td>
<td>INVALID CHARACTER VALUE FOR CAST</td>
<td>invalid_character_value_for_cast</td>
</tr>
<tr>
<td>22007</td>
<td>INVALID DATETIME FORMAT</td>
<td>invalid_datetime_format</td>
</tr>
<tr>
<td>22019</td>
<td>INVALID ESCAPE CHARACTER</td>
<td>invalid_escape_character</td>
</tr>
<tr>
<td>2200D</td>
<td>INVALID ESCAPE OCTET</td>
<td>invalid_escape_octet</td>
</tr>
<tr>
<td>22025</td>
<td>INVALID ESCAPE SEQUENCE</td>
<td>invalid_escape_sequence</td>
</tr>
<tr>
<td>22P06</td>
<td>NONSTANDARD USE OF ESCAPE CHARACTER</td>
<td>nonstandard_use_of_escape_character</td>
</tr>
<tr>
<td>22010</td>
<td>INVALID INDICATOR PARAMETER VALUE</td>
<td>invalid_indicator_parameter_value</td>
</tr>
<tr>
<td>22020</td>
<td>INVALID LIMIT VALUE</td>
<td>invalid_limit_value</td>
</tr>
<tr>
<td>22023</td>
<td>INVALID PARAMETER VALUE</td>
<td>invalid_parameter_value</td>
</tr>
<tr>
<td>2201B</td>
<td>INVALID REGULAR EXPRESSION</td>
<td>invalid_regular_expression</td>
</tr>
<tr>
<td>22009</td>
<td>INVALID TIME ZONE DISPLACEMENT VALUE</td>
<td>invalid_time_zone_displacement_value</td>
</tr>
<tr>
<td>2200C</td>
<td>INVALID USE OF ESCAPE CHARACTER</td>
<td>invalid_use_of_escape_character</td>
</tr>
<tr>
<td>2200G</td>
<td>MOST SPECIFIC TYPE MISMATCH</td>
<td>most_specific_type_mismatch</td>
</tr>
<tr>
<td>22004</td>
<td>NULL VALUE NOT ALLOWED</td>
<td>null_value_not_allowed</td>
</tr>
<tr>
<td>22002</td>
<td>NULL VALUE NO INDICATOR PARAMETER</td>
<td>null_value_no_indicator_parameter</td>
</tr>
<tr>
<td>22003</td>
<td>NUMERIC VALUE OUT OF RANGE</td>
<td>numeric_value_out_of_range</td>
</tr>
</tbody>
</table>
## Appendix A. PostgreSQL Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>22026</td>
<td>STRING DATA LENGTH MISMATCH</td>
<td>string_data_length_mismatch</td>
</tr>
<tr>
<td>22001</td>
<td>STRING DATA RIGHT TRUNCATION</td>
<td>string_data_right_truncation</td>
</tr>
<tr>
<td>22011</td>
<td>SUBSTRING ERROR</td>
<td>substring_error</td>
</tr>
<tr>
<td>22027</td>
<td>TRIM ERROR</td>
<td>trim_error</td>
</tr>
<tr>
<td>22024</td>
<td>UNTERMINATED C STRING</td>
<td>unterminated_c_string</td>
</tr>
<tr>
<td>2200F</td>
<td>ZERO LENGTH CHARACTER STRING</td>
<td>zero_length_character_string</td>
</tr>
<tr>
<td>22P01</td>
<td>FLOATING POINT EXCEPTION</td>
<td>floating_point_exception</td>
</tr>
<tr>
<td>22P02</td>
<td>INVALID TEXT REPRESENTATION</td>
<td>invalid_text_representation</td>
</tr>
<tr>
<td>22P03</td>
<td>INVALID BINARY REPRESENTATION</td>
<td>invalid_binary_representation</td>
</tr>
<tr>
<td>22P04</td>
<td>BAD COPY FILE FORMAT</td>
<td>bad_copy_file_format</td>
</tr>
<tr>
<td>22P05</td>
<td>UNTRANSLATABLE CHARACTER</td>
<td>untranslatable_character</td>
</tr>
<tr>
<td>2200L</td>
<td>NOT AN XML DOCUMENT</td>
<td>not_an_xml_document</td>
</tr>
<tr>
<td>2200M</td>
<td>INVALID XML DOCUMENT</td>
<td>invalid_xml_document</td>
</tr>
<tr>
<td>2200N</td>
<td>INVALID XML CONTENT</td>
<td>invalid_xml_content</td>
</tr>
<tr>
<td>2200S</td>
<td>INVALID XML COMMENT</td>
<td>invalid_xml_comment</td>
</tr>
<tr>
<td>2200T</td>
<td>INVALID XML PROCESSING INSTRUCTION</td>
<td>invalid_xml_processing_instruction</td>
</tr>
</tbody>
</table>

### Class 23 — Integrity Constraint Violation

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>23000</td>
<td>INTEGRITY CONSTRAINT VIOLATION</td>
<td>integrity_constraint_violation</td>
</tr>
<tr>
<td>23001</td>
<td>RESTRICT VIOLATION</td>
<td>restrict_violation</td>
</tr>
<tr>
<td>23502</td>
<td>NOT NULL VIOLATION</td>
<td>not_null_violation</td>
</tr>
<tr>
<td>23503</td>
<td>FOREIGN KEY VIOLATION</td>
<td>foreign_key_violation</td>
</tr>
<tr>
<td>23505</td>
<td>UNIQUE VIOLATION</td>
<td>unique_violation</td>
</tr>
<tr>
<td>23514</td>
<td>CHECK VIOLATION</td>
<td>check_violation</td>
</tr>
</tbody>
</table>

### Class 24 — Invalid Cursor State

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>24000</td>
<td>INVALID CURSOR STATE</td>
<td>invalid_cursor_state</td>
</tr>
</tbody>
</table>

### Class 25 — Invalid Transaction State

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>25000</td>
<td>INVALID TRANSACTION STATE</td>
<td>invalid_transaction_state</td>
</tr>
<tr>
<td>25001</td>
<td>ACTIVE SQL TRANSACTION</td>
<td>active_sql_transaction</td>
</tr>
<tr>
<td>25002</td>
<td>BRANCH TRANSACTION ALREADY ACTIVE</td>
<td>branch_transaction_already_active</td>
</tr>
<tr>
<td>25008</td>
<td>HELD CURSOR REQUIRES SAME ISOLATION LEVEL</td>
<td>held_cursor_requires_same_isolation_level</td>
</tr>
</tbody>
</table>
### Appendix A. PostgreSQL Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>25003</td>
<td>INAPPROPRIATE ACCESS MODE FOR BRANCH TRANSACTION</td>
<td>inappropriate_access_mode_for_branch_transaction</td>
</tr>
<tr>
<td>25004</td>
<td>INAPPROPRIATE ISOLATION LEVEL FOR BRANCH TRANSACTION</td>
<td>inappropriate_isolation_level_for_branch_transaction</td>
</tr>
<tr>
<td>25005</td>
<td>NO ACTIVE SQL TRANSACTION FOR BRANCH TRANSACTION</td>
<td>no_active_sql_transaction_for_branch_transaction</td>
</tr>
<tr>
<td>25006</td>
<td>READ ONLY SQL TRANSACTION</td>
<td>read_only_sql_transaction</td>
</tr>
<tr>
<td>25007</td>
<td>SCHEMA AND DATA STATEMENT MIXING NOT SUPPORTED</td>
<td>schema_and_data_statement_mixing_not_supported</td>
</tr>
<tr>
<td>25P01</td>
<td>NO ACTIVE SQL TRANSACTION</td>
<td>no_active_sql_transaction</td>
</tr>
<tr>
<td>25P02</td>
<td>IN FAILED SQL TRANSACTION</td>
<td>in_failed_sql_transaction</td>
</tr>
</tbody>
</table>

**Class 26 — Invalid SQL Statement Name**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>26000</td>
<td>INVALID SQL STATEMENT NAME</td>
<td>invalid_sql_statement_name</td>
</tr>
</tbody>
</table>

**Class 27 — Triggered Data Change Violation**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>27000</td>
<td>TRIGGERED DATA CHANGE VIOLATION</td>
<td>triggered_data_change_violation</td>
</tr>
</tbody>
</table>

**Class 28 — Invalid Authorization Specification**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>28000</td>
<td>INVALID AUTHORIZATION SPECIFICATION</td>
<td>invalid_authorization_specification</td>
</tr>
</tbody>
</table>

**Class 2B — Dependent Privilege Descriptors Still Exist**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>2B000</td>
<td>DEPENDENT PRIVILEGE DESCRIPTORS STILL EXIST</td>
<td>dependent_privilege_descriptors_still_exist</td>
</tr>
<tr>
<td>2BP01</td>
<td>DEPENDENT OBJECTS STILL EXIST</td>
<td>dependent_objects_still_exist</td>
</tr>
</tbody>
</table>

**Class 2D — Invalid Transaction Termination**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D000</td>
<td>INVALID TRANSACTION TERMINATION</td>
<td>invalid_transaction_termination</td>
</tr>
</tbody>
</table>

**Class 2F — SQL Routine Exception**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>2F000</td>
<td>SQL ROUTINE EXCEPTION</td>
<td>sql_routine_exception</td>
</tr>
<tr>
<td>2F005</td>
<td>FUNCTION EXECUTED NO RETURN STATEMENT</td>
<td>function_executed_no_return_statement</td>
</tr>
<tr>
<td>2F002</td>
<td>MODIFYING SQL DATA NOT PERMITTED</td>
<td>modifying_sql_data_not_permitted</td>
</tr>
<tr>
<td>2F003</td>
<td>PROHIBITED SQL STATEMENT ATTEMPTED</td>
<td>prohibited_sql_statement_attempted</td>
</tr>
</tbody>
</table>
## Appendix A. PostgreSQL Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>2F004</td>
<td>READING SQL DATA NOT PERMITTED</td>
<td>reading_sql_data_not_permitted</td>
</tr>
</tbody>
</table>

### Class 34 — Invalid Cursor Name

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>34000</td>
<td>INVALID CURSOR NAME</td>
<td>invalid_cursor_name</td>
</tr>
</tbody>
</table>

### Class 38 — External Routine Exception

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>38000</td>
<td>EXTERNAL ROUTINE EXCEPTION</td>
<td>external_routine_exception</td>
</tr>
<tr>
<td>38001</td>
<td>CONTAINING SQL DATA NOT PERMITTED</td>
<td>containing_sql_not_permitted</td>
</tr>
<tr>
<td>38002</td>
<td>MODIFYING SQL DATA NOT PERMITTED</td>
<td>modifying_sql_data_not_permitted</td>
</tr>
<tr>
<td>38003</td>
<td>PROHIBITED SQL STATEMENT ATTEMPTED</td>
<td>prohibited_sql_statement_attempted</td>
</tr>
<tr>
<td>38004</td>
<td>READING SQL DATA NOT PERMITTED</td>
<td>reading_sql_data_not_permitted</td>
</tr>
</tbody>
</table>

### Class 39 — External Routine Invocation Exception

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>39000</td>
<td>EXTERNAL ROUTINE INVOCATION EXCEPTION</td>
<td>external_routine_invocation_exception</td>
</tr>
<tr>
<td>39001</td>
<td>INVALID SQLSTATE RETURNED</td>
<td>invalid_sqlstate_returned</td>
</tr>
<tr>
<td>39002</td>
<td>NULL VALUE NOT ALLOWED</td>
<td>null_value_not_allowed</td>
</tr>
<tr>
<td>39P01</td>
<td>TRIGGER PROTOCOL VIOLATED</td>
<td>trigger_protocol_violated</td>
</tr>
<tr>
<td>39P02</td>
<td>SRF PROTOCOL VIOLATED</td>
<td>srf_protocol_violated</td>
</tr>
</tbody>
</table>

### Class 3B — Savepoint Exception

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>3B000</td>
<td>SAVEPOINT EXCEPTION</td>
<td>savepoint_exception</td>
</tr>
<tr>
<td>3B001</td>
<td>INVALID SAVEPOINT SPECIFICATION</td>
<td>invalid_savepoint_specification</td>
</tr>
</tbody>
</table>

### Class 3D — Invalid Catalog Name

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D000</td>
<td>INVALID CATALOG NAME</td>
<td>invalid_catalog_name</td>
</tr>
</tbody>
</table>

### Class 3F — Invalid Schema Name

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F000</td>
<td>INVALID SCHEMA NAME</td>
<td>invalid_schema_name</td>
</tr>
</tbody>
</table>

### Class 40 — Transaction Rollback

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>40000</td>
<td>TRANSACTION ROLLBACK</td>
<td>transaction_rollback</td>
</tr>
<tr>
<td>40002</td>
<td>TRANSACTION INTEGRITY CONSTRAINT VIOLATION</td>
<td>transaction_integrity_constraint_violation</td>
</tr>
<tr>
<td>40001</td>
<td>SERIALIZATION FAILURE</td>
<td>serialization_failure</td>
</tr>
<tr>
<td>40003</td>
<td>STATEMENT COMPLETION UNKNOWN</td>
<td>statement_completion_unknown</td>
</tr>
<tr>
<td>40P01</td>
<td>DEADLOCK DETECTED</td>
<td>deadlock_detected</td>
</tr>
</tbody>
</table>

### Class 42 — Syntax Error or Access Rule Violation

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>42000</td>
<td>SYNTAX ERROR OR ACCESS RULE VIOLATION</td>
<td>syntax_error_or_access_rule_violation</td>
</tr>
<tr>
<td>Error Code</td>
<td>Meaning</td>
<td>Constant</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>42601</td>
<td>SYNTAX ERROR</td>
<td>syntax_error</td>
</tr>
<tr>
<td>42501</td>
<td>INSUFFICIENT PRIVILEGE</td>
<td>insufficient_privilege</td>
</tr>
<tr>
<td>42846</td>
<td>CANNOT COERC</td>
<td>cannot_coerce</td>
</tr>
<tr>
<td>42803</td>
<td>GROUPING ERROR</td>
<td>grouping_error</td>
</tr>
<tr>
<td>42830</td>
<td>INVALID FOREIGN KEY</td>
<td>invalid_foreign_key</td>
</tr>
<tr>
<td>42602</td>
<td>INVALID NAME</td>
<td>invalid_name</td>
</tr>
<tr>
<td>42622</td>
<td>NAME TOO LONG</td>
<td>name_too_long</td>
</tr>
<tr>
<td>42939</td>
<td>RESERVED NAME</td>
<td>reserved_name</td>
</tr>
<tr>
<td>42804</td>
<td>DATATYPE MISMATCH</td>
<td>datatype_mismatch</td>
</tr>
<tr>
<td>42P18</td>
<td>INDETERMINATE DATATYPE</td>
<td>indeterminate_datatype</td>
</tr>
<tr>
<td>42809</td>
<td>WRONG OBJECT TYPE</td>
<td>wrong_object_type</td>
</tr>
<tr>
<td>42703</td>
<td>UNDEFINED COLUMN</td>
<td>undefined_column</td>
</tr>
<tr>
<td>42883</td>
<td>UNDEFINED FUNCTION</td>
<td>undefined_function</td>
</tr>
<tr>
<td>42P01</td>
<td>UNDEFINED TABLE</td>
<td>undefined_table</td>
</tr>
<tr>
<td>42P02</td>
<td>UNDEFINED PARAMETER</td>
<td>undefined_parameter</td>
</tr>
<tr>
<td>42704</td>
<td>UNDEFINED OBJECT</td>
<td>undefined_object</td>
</tr>
<tr>
<td>42701</td>
<td>DUPLICATE COLUMN</td>
<td>duplicate_column</td>
</tr>
<tr>
<td>42P03</td>
<td>DUPLICATE CURSOR</td>
<td>duplicate_cursor</td>
</tr>
<tr>
<td>42P04</td>
<td>DUPLICATE DATABASE</td>
<td>duplicate_database</td>
</tr>
<tr>
<td>42723</td>
<td>DUPLICATE FUNCTION</td>
<td>duplicate_function</td>
</tr>
<tr>
<td>42P05</td>
<td>DUPLICATE PREPARED STATEMENT</td>
<td>duplicate_prepared_statement</td>
</tr>
<tr>
<td>42P06</td>
<td>DUPLICATE SCHEMA</td>
<td>duplicate_schema</td>
</tr>
<tr>
<td>42P07</td>
<td>DUPLICATE TABLE</td>
<td>duplicate_table</td>
</tr>
<tr>
<td>42712</td>
<td>DUPLICATE ALIAS</td>
<td>duplicate_alias</td>
</tr>
<tr>
<td>42710</td>
<td>DUPLICATE OBJECT</td>
<td>duplicate_object</td>
</tr>
<tr>
<td>42702</td>
<td>AMBIGUOUS COLUMN</td>
<td>ambiguous_column</td>
</tr>
<tr>
<td>42725</td>
<td>AMBIGUOUS FUNCTION</td>
<td>ambiguous_function</td>
</tr>
<tr>
<td>42P08</td>
<td>AMBIGUOUS PARAMETER</td>
<td>ambiguous_parameter</td>
</tr>
<tr>
<td>42P09</td>
<td>AMBIGUOUS ALIAS</td>
<td>ambiguous_alias</td>
</tr>
<tr>
<td>42P10</td>
<td>INVALID COLUMN REFERENCE</td>
<td>invalid_column_reference</td>
</tr>
<tr>
<td>42611</td>
<td>INVALID COLUMN DEFINITION</td>
<td>invalid_column_definition</td>
</tr>
<tr>
<td>42P11</td>
<td>INVALID CURSOR DEFINITION</td>
<td>invalid_cursor_definition</td>
</tr>
<tr>
<td>42P12</td>
<td>INVALID DATABASE DEFINITION</td>
<td>invalid_database_definition</td>
</tr>
<tr>
<td>42P13</td>
<td>INVALID FUNCTION DEFINITION</td>
<td>invalid_function_definition</td>
</tr>
<tr>
<td>42P14</td>
<td>INVALID PREPARED STATEMENT DEFINITION</td>
<td>invalid_prepared_statement_definition</td>
</tr>
</tbody>
</table>
## Appendix A. PostgreSQL Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>42P15</td>
<td>INVALID SCHEMA DEFINITION</td>
<td>invalid_schema_definition</td>
</tr>
<tr>
<td>42P16</td>
<td>INVALID TABLE DEFINITION</td>
<td>invalid_table_definition</td>
</tr>
<tr>
<td>42P17</td>
<td>INVALID OBJECT DEFINITION</td>
<td>invalid_object_definition</td>
</tr>
</tbody>
</table>

**Class 44 — WITH CHECK OPTION Violation**

| 44000      | WITH CHECK OPTION VIOLATION           | with_check_option_violation |

**Class 53 — Insufficient Resources**

| 53000      | INSUFFICIENT RESOURCES                | insufficient_resources     |
| 53100      | DISK FULL                              | disk_full                  |
| 53200      | OUT OF MEMORY                          | out_of_memory              |
| 53300      | TOO MANY CONNECTIONS                   | too_many_connections       |

**Class 54 — Program Limit Exceeded**

| 54000      | PROGRAM LIMIT EXCEEDED                 | program_limit_exceeded     |
| 54001      | STATEMENT TOO COMPLEX                   | statement_too_complex      |
| 54011      | TOO MANY COLUMNS                        | too_many_columns           |
| 54023      | TOO MANY ARGUMENTS                      | too_many_arguments         |

**Class 55 — Object Not In Prerequisite State**

| 55000      | OBJECT NOT IN PREREQUISITE STATE        | object_not_in_prerequisite_state |
| 55006      | OBJECT IN USE                           | object_in_use                |
| 55P02      | CANT CHANGE RUNTIME PARAM               | cant_change_runtime_param    |
| 55P03      | LOCK NOT AVAILABLE                      | lock_not_available           |

**Class 57 — Operator Intervention**

| 57000      | OPERATOR INTERVENTION                   | operator_intervention       |
| 57014      | QUERY CANCELED                          | query_canceled              |
| 57P01      | ADMIN SHUTDOWN                          | admin_shutdown              |
| 57P02      | CRASH SHUTDOWN                          | crash_shutdown              |
| 57P03      | CANNOT CONNECT NOW                      | cannot_connect_now          |

**Class 58 — System Error (errors external to PostgreSQL itself)**

| 58030      | IO ERROR                                | io_error                    |
| 58P01      | UNDEFINED FILE                          | undefined_file              |
| 58P02      | DUPLICATE FILE                          | duplicate_file              |

**Class F0 — Configuration File Error**

<p>| F0000      | CONFIG FILE ERROR                       | config_file_error           |
| F0001      | LOCK FILE EXISTS                        | lock_file_exists            |</p>
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0000</td>
<td>PLPGSQL ERROR</td>
<td>plpgsql_error</td>
</tr>
<tr>
<td>P0001</td>
<td>RAISE EXCEPTION</td>
<td>raise_exception</td>
</tr>
<tr>
<td>P0002</td>
<td>NO DATA FOUND</td>
<td>no_data_found</td>
</tr>
<tr>
<td>P0003</td>
<td>TOO MANY ROWS</td>
<td>too_many_rows</td>
</tr>
<tr>
<td>XX000</td>
<td>INTERNAL ERROR</td>
<td>internal_error</td>
</tr>
<tr>
<td>XX001</td>
<td>DATA CORRUPTED</td>
<td>data_corrupted</td>
</tr>
<tr>
<td>XX002</td>
<td>INDEX CORRUPTED</td>
<td>index_corrupted</td>
</tr>
</tbody>
</table>
Appendix B. Date/Time Support

PostgreSQL uses an internal heuristic parser for all date/time input support. Dates and times are input as strings, and are broken up into distinct fields with a preliminary determination of what kind of information can be in the field. Each field is interpreted and either assigned a numeric value, ignored, or rejected. The parser contains internal lookup tables for all textual fields, including months, days of the week, and time zones.

This appendix includes information on the content of these lookup tables and describes the steps used by the parser to decode dates and times.

B.1. Date/Time Input Interpretation

The date/time type inputs are all decoded using the following procedure.

1. Break the input string into tokens and categorize each token as a string, time, time zone, or number.
   a. If the numeric token contains a colon (:), this is a time string. Include all subsequent digits and colons.
   b. If the numeric token contains a dash (-), slash (/), or two or more dots (.), this is a date string which might have a text month. If a date token has already been seen, it is instead interpreted as a time zone name (e.g., America/New_York).
   c. If the token is numeric only, then it is either a single field or an ISO 8601 concatenated date (e.g., 19990113 for January 13, 1999) or time (e.g., 141516 for 14:15:16).
   d. If the token starts with a plus (+) or minus (-), then it is either a numeric time zone or a special field.

2. If the token is a text string, match up with possible strings:
   a. Do a binary-search table lookup for the token as a time zone abbreviation.
   b. If not found, do a similar binary-search table lookup to match the token as either a special string (e.g., today), day (e.g., Thursday), month (e.g., January), or noise word (e.g., at, on).
   c. If still not found, throw an error.

3. When the token is a number or number field:
   a. If there are eight or six digits, and if no other date fields have been previously read, then interpret as a “concatenated date” (e.g., 19990118 or 990118). The interpretation is YYYYMMDD or YYMMD.D.
   b. If the token is three digits and a year has already been read, then interpret as day of year.
   c. If four or six digits and a year has already been read, then interpret as a time (HHMM or HHMMSS).
Appendix B. Date/Time Support

d. If three or more digits and no date fields have yet been found, interpret as a year (this forces yy-mm-dd ordering of the remaining date fields).

e. Otherwise the date field ordering is assumed to follow the DateStyle setting: mm-dd-yy, dd-mm-yy, or yy-mm-dd. Throw an error if a month or day field is found to be out of range.

4. If BC has been specified, negate the year and add one for internal storage. (There is no year zero in the Gregorian calendar, so numerically 1 BC becomes year zero.)

5. If BC was not specified, and if the year field was two digits in length, then adjust the year to four digits. If the field is less than 70, then add 2000, otherwise add 1900.

Tip: Gregorian years AD 1-99 can be entered by using 4 digits with leading zeros (e.g., 0099 is AD 99).

B.2. Date/Time Key Words

Table B-1 shows the tokens that are recognized as names of months.

<table>
<thead>
<tr>
<th>Month</th>
<th>Abbreviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>January</td>
<td>Jan</td>
</tr>
<tr>
<td>February</td>
<td>Feb</td>
</tr>
<tr>
<td>March</td>
<td>Mar</td>
</tr>
<tr>
<td>April</td>
<td>Apr</td>
</tr>
<tr>
<td>May</td>
<td></td>
</tr>
<tr>
<td>June</td>
<td>Jun</td>
</tr>
<tr>
<td>July</td>
<td>Jul</td>
</tr>
<tr>
<td>August</td>
<td>Aug</td>
</tr>
<tr>
<td>September</td>
<td>Sep, Sept</td>
</tr>
<tr>
<td>October</td>
<td>Oct</td>
</tr>
<tr>
<td>November</td>
<td>Nov</td>
</tr>
<tr>
<td>December</td>
<td>Dec</td>
</tr>
</tbody>
</table>

Table B-2 shows the tokens that are recognized as names of days of the week.

<table>
<thead>
<tr>
<th>Day</th>
<th>Abbreviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunday</td>
<td>Sun</td>
</tr>
<tr>
<td>Monday</td>
<td>Mon</td>
</tr>
<tr>
<td>Tuesday</td>
<td>Tue, Tues</td>
</tr>
<tr>
<td>Wednesday</td>
<td>Wed, Weds</td>
</tr>
</tbody>
</table>
Table B-3 shows the tokens that serve various modifier purposes.

Table B-3. Date/Time Field Modifiers

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTIME</td>
<td>Ignored</td>
</tr>
<tr>
<td>AM</td>
<td>Time is before 12:00</td>
</tr>
<tr>
<td>AT</td>
<td>Ignored</td>
</tr>
<tr>
<td>JULIAN, JD, J</td>
<td>Next field is Julian Day</td>
</tr>
<tr>
<td>ON</td>
<td>Ignored</td>
</tr>
<tr>
<td>PM</td>
<td>Time is on or after 12:00</td>
</tr>
<tr>
<td>T</td>
<td>Next field is time</td>
</tr>
</tbody>
</table>

The key word ABSTIME is ignored for historical reasons: In very old releases of PostgreSQL, invalid values of type abstime were emitted as Invalid Abstime. This is no longer the case however and this key word will likely be dropped in a future release.

### B.3. Date/Time Configuration Files

Since timezone abbreviations are not well standardized, PostgreSQL provides a means to customize the set of abbreviations accepted by the server. The timezone_abbreviations run-time parameter determines the active set of abbreviations. While this parameter can be altered by any database user, the possible values for it are under the control of the database administrator — they are in fact names of configuration files stored in .../share/timezonesets/ of the installation directory. By adding or altering files in that directory, the administrator can set local policy for timezone abbreviations.

timezone_abbreviations can be set to any file name found in .../share/timezonesets/, if the file’s name is entirely alphabetic. (The prohibition against non-alphabetic characters in timezone_abbreviations prevents reading files outside the intended directory, as well as reading editor backup files and other extraneous files.)

A timezone abbreviation file can contain blank lines and comments beginning with #. Non-comment lines must have one of these formats:

```
time_zone_name offset
@INCLUDE file_name
@OVERRIDE
```

A time_zone_name is just the abbreviation being defined. The offset is the zone’s offset in seconds from UTC, positive being east from Greenwich and negative being west. For example, -18000 would be five hours west of Greenwich, or North American east coast standard time. D indicates that the zone name represents local daylight-savings time rather than standard time. Since all known time zone offsets are on 15 minute boundaries, the number of seconds has to be a multiple of 900.
The @INCLUDE syntax allows inclusion of another file in the .../share/timezonesets/ directory. Inclusion can be nested, to a limited depth.

The @OVERRIDE syntax indicates that subsequent entries in the file can override previous entries (i.e., entries obtained from included files). Without this, conflicting definitions of the same timezone abbreviation are considered an error.

In an unmodified installation, the file Default contains all the non-conflicting time zone abbreviations for most of the world. Additional files Australia and India are provided for those regions: these files first include the Default file and then add or modify timezones as needed.

For reference purposes, a standard installation also contains files Africa.txt, America.txt, etc, containing information about every time zone abbreviation known to be in use according to the zic timezone database. The zone name definitions found in these files can be copied and pasted into a custom configuration file as needed. Note that these files cannot be directly referenced as timezone_abbreviations settings, because of the dot embedded in their names.

Note: If an error occurs while reading the time zone data sets, no new value is applied but the old set is kept. If the error occurs while starting the database, startup fails.

Caution

Time zone abbreviations defined in the configuration file override non-timezone meanings built into PostgreSQL. For example, the Australia configuration file defines SAT (for South Australian Standard Time). When this file is active, SAT will not be recognized as an abbreviation for Saturday.

Caution

If you modify files in .../share/timezonesets/, it is up to you to make backups — a normal database dump will not include this directory.

B.4. History of Units

The Julian calendar was introduced by Julius Caesar in 45 BC. It was in common use in the Western world until the year 1582, when countries started changing to the Gregorian calendar. In the Julian calendar, the tropical year is approximated as 365 1/4 days = 365.25 days. This gives an error of about 1 day in 128 years.

The accumulating calendar error prompted Pope Gregory XIII to reform the calendar in accordance with instructions from the Council of Trent. In the Gregorian calendar, the tropical year is approximated as 365 + 97 / 400 days = 365.2425 days. Thus it takes approximately 3300 years for the tropical year to shift one day with respect to the Gregorian calendar.

The approximation 365+97/400 is achieved by having 97 leap years every 400 years, using the following rules:

Every year divisible by 4 is a leap year.
However, every year divisible by 100 is not a leap year.
However, every year divisible by 400 is a leap year after all.
Appendix B. Date/Time Support

So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years. By contrast, in the older Julian calendar all years divisible by 4 are leap years.

The papal bull of February 1582 decreed that 10 days should be dropped from October 1582 so that 15 October should follow immediately after 4 October. This was observed in Italy, Poland, Portugal, and Spain. Other Catholic countries followed shortly after, but Protestant countries were reluctant to change, and the Greek Orthodox countries didn’t change until the start of the 20th century. The reform was observed by Great Britain and Dominions (including what is now the USA) in 1752. Thus 2 September 1752 was followed by 14 September 1752. This is why Unix systems have the `cal` program produce the following:

```
$ cal 9 1752
  September 1752
   S M Tu W Th F S
  1  2 14 15 16
 17 18 19 20 21 22 23
 24 25 26 27 28 29 30
```

The SQL standard states that “Within the definition of a ‘datetime literal’, the ‘datetime value’s are constrained by the natural rules for dates and times according to the Gregorian calendar”. Dates between 1582-10-05 and 1582-10-14, although eliminated in some countries by Papal fiat, conform to “natural rules” and are hence valid dates. PostgreSQL follows the SQL standard’s lead by counting dates exclusively in the Gregorian calendar, even for years before that calendar was in use.

Different calendars have been developed in various parts of the world, many predating the Gregorian system. For example, the beginnings of the Chinese calendar can be traced back to the 14th century BC. Legend has it that the Emperor Huangdi invented that calendar in 2637 BC. The People’s Republic of China uses the Gregorian calendar for civil purposes. The Chinese calendar is used for determining festivals.

The “Julian Date” is unrelated to the “Julian calendar”. The Julian Date system was invented by the French scholar Joseph Justus Scaliger (1540-1609) and probably takes its name from Scaliger’s father, the Italian scholar Julius Caesar Scaliger (1484-1558). In the Julian Date system, each day has a sequential number, starting from JD 0 (which is sometimes called the Julian Date). JD 0 corresponds to 1 January 4713 BC in the Julian calendar, or 24 November 4714 BC in the Gregorian calendar. Julian Date counting is most often used by astronomers for labeling their nightly observations, and therefore a date runs from noon UTC to the next noon UTC, rather than from midnight to midnight: JD 0 designates the 24 hours from noon UTC on 1 January 4713 BC to noon UTC on 2 January 4713 BC.

Although PostgreSQL supports Julian Date notation for input and output of dates (and also uses them for some internal datetime calculations), it does not observe the nicety of having dates run from noon to noon. PostgreSQL treats a Julian Date as running from midnight to midnight.
Appendix C. SQL Key Words

Table C-1 lists all tokens that are key words in the SQL standard and in PostgreSQL 8.3.23. Background information can be found in Section 4.1.1.

SQL distinguishes between reserved and non-reserved key words. According to the standard, reserved key words are the only real key words; they are never allowed as identifiers. Non-reserved key words only have a special meaning in particular contexts and can be used as identifiers in other contexts. Most non-reserved key words are actually the names of built-in tables and functions specified by SQL. The concept of non-reserved key words essentially only exists to declare that some predefined meaning is attached to a word in some contexts.

In the PostgreSQL parser life is a bit more complicated. There are several different classes of tokens ranging from those that can never be used as an identifier to those that have absolutely no special status in the parser as compared to an ordinary identifier. (The latter is usually the case for functions specified by SQL.) Even reserved key words are not completely reserved in PostgreSQL, but can be used as column labels (for example, SELECT 55 AS CHECK, even though CHECK is a reserved key word).

In Table C-1 in the column for PostgreSQL we classify as “non-reserved” those key words that are explicitly known to the parser but are allowed as column or table names. Some key words that are otherwise non-reserved cannot be used as function or data type names and are marked accordingly. (Most of these words represent built-in functions or data types with special syntax. The function or type is still available but it cannot be redefined by the user.) Labeled “reserved” are those tokens that are not allowed as column or table names. Some reserved key words are allowable as names for functions or data types; this is also shown in the table. If not so marked, a reserved key word is only allowed as an “AS” column label name.

As a general rule, if you get spurious parser errors for commands that contain any of the listed key words as an identifier you should try to quote the identifier to see if the problem goes away.

It is important to understand before studying Table C-1 that the fact that a key word is not reserved in PostgreSQL does not mean that the feature related to the word is not implemented. Conversely, the presence of a key word does not indicate the existence of a feature.

Table C-1. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABORT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABS</td>
<td></td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>ABSOLUTE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ACCESS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACTION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ADA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>ADD</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ADMIN</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>AFTER</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>AGGREGATE</td>
<td>non-reserved</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIAS</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ALL</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ALSO</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALTER</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ALWAYS</td>
<td>non-reserved</td>
<td></td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>ANALYZE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANALYZE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ANY</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ARE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ARRAY</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>AS</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ASC</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ASENSITIVE</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSERTION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>ASYMMETRIC</td>
<td>reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>AT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ATOMIC</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATTRIBUTE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTHORIZATION</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>AVG</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>BACKWARD</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASE64</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>BEGIN</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>BERNOULLI</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BETWEEN</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>BIGINT</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BINARY</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>BITVAR</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIT_LENGTH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>BLOB</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>BOTH</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>BREADTH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>BY</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>C</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>CACHE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALLED</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>CARDINALITY</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CASCADE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CASCADED</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CASE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CAST</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CATALOG</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CATALOG_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>CEIL</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEILING</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAIN</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CHARACTERISTICS</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHARACTERS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHARACTER_LENGTH</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CHARACTER_SET_CATALOG</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>CHARACTER_SET_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>CHARACTER_SET_SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>CHAR_LENGTH</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CHECK</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CHECKED</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHECKPOINT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLASS</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLASS_ORIGIN</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
</tbody>
</table>
### Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>CLOSE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>CLUSTER</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>COALESCE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>COLLATE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>COLLATION</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>COLLATION_CATALOG</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>COLLATION_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>COLLATION_SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>COLLECT</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLUMN</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>COMMAND_FUNCTION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>COMMAND_FUNCTION_CODE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>COMMENT</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>COMMITTED</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>COMPLETION</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONCURRENTLY</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONDITION</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONDITION_NUMBER</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>CONFIGURATION</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONNECT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>CONNECTION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>CONNECTION_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>CONSTRAINT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>CONSTRAINT_CATALOG</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>CONSTRAINT_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>CONSTRAINT_SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>CONSTRUCTOR</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>CONTAINS</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>CONTENT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CONVERSION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>CONVERT</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>COPY</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CORR</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CORRESPONDING</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>COST</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>COUNT</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>COVAR_POP</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>COVAR_SAMP</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CREATE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CREATEDB</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CREATEROLE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CREATEUSER</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CROSS</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CSV</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CUBE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CUME_DIST</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT_DATE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT_DEFAULT_TRANSFORM_GROUP</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT_PATH</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT_ROLE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT_TIME</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT_TRANSFORM_GROUP_FOR_TYPE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURRENT_USER</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURSOR</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CURSOR_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>CYCLE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DATA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DATABASE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DATE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DATETIME_INTERVAL_CODE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DATETIME_INTERVAL_PRECISION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>DAY</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEALLOCATE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEC</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DECLARE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEFAULTS</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEFERRABLE</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEFERRED</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEFINED</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEFINER</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEGREE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DELETE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DENSE_RANK</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPTH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DEREF</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DERIVED</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DESC</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DESCRIPTOR</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DESTROY</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DESTRUCTOR</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETERMINISTIC</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DIAGNOSTICS</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DICTIONARY</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DISABLE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DISCARD</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DISPATCH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DO</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOCUMENT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DROP</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNAMIC_FUNCTION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>DYNAMIC_FUNCTION_CODE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EACH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>ELEMENT</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ENABLE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCODING</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCRYPTED</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>END-EXEC</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ENUM</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUALS</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>ESCAPE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>EVERY</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXCEPT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>EXCLUDE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXCLUDING</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>EXISTING</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXISTS</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>(cannot be function or type)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>EXTRACT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>(cannot be function or type)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FAMILY</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FETCH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FILTER</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FINAL</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRST</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FLOAT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>(cannot be function or type)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOOR</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOLLOWING</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FORCE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOREIGN</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>FORWARD</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOUND</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FREE</td>
<td>reserved</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>FREEZE</td>
<td>reserved (can be function or type)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FROM</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FULL</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>FUSION</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GENERAL</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GENERATED</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>GO</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>GOTO</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>GRANT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>GRANTED</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>GREATEST</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>GROUP</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>GROUPING</td>
<td>reserved</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>HANDLER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAVING</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>HEADER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HEX</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HIERARCHY</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOLD</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>HOST</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOUR</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>IDENTITY</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>IF</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IGNORE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILIKE</td>
<td>reserved (can be function or type)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>IMMEDIATE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>IMMUTABLE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPLEMENTATION</td>
<td></td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>IMPLICIT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INCLUDING</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INCREMENT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEX</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEXES</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDICATOR</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INFIX</td>
<td></td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INHERIT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INHERITS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INITIALIZE</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INITIALLY</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INNER</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INOUT</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INPUT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INSENSITIVE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INSERT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INSTANCE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTANTIABLE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSTEAD</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INTEGER</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INTERSECT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INTERSECTION</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERVAL</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INTO</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>INVOKER</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>IS</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ISNULL</td>
<td>reserved (can be function or type)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISOLATION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ITERATE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOIN</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>K</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>KEY</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>KEY_MEMBER</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>KEY_TYPE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>LANCMPILER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LARGE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LAST</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LATERAL</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEADING</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LEAST</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LENGTH</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>LESS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEVEL</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LIKE</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LIMIT</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LISTEN</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LN</td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCAL</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LOCALTIME</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LOCALTIMESTAMP</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>LOCATION</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCATOR</td>
<td>non-reserved</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>LOCK</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGIN</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOWER</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>M</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAP</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>MAPPING</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MATCH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>MATCHED</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>MAXVALUE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMBER</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MERGE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MESSAGE_LENGTH</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>MESSAGE_OCTET_LENGTH</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>MESSAGE_TEXT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>METHOD</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>MINUTE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>MINVALUE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOD</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODIFIES</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODIFY</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODULE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>MONTH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>MORE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>MOVE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTISET</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUMPS</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>NAMES</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NATIONAL</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NATURAL</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NCHAR</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NCLOB</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NESTING</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEXT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NO</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NOCREATEDB</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOCREATEROLE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOCREATEUSER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOINHERIT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOLOGIN</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------</td>
<td>----------------</td>
<td>----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>NONE</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>NORMALIZE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NORMALIZED</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOSUPERUSER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NOTHING</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTIFY</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTNULL</td>
<td>reserved (can be function or type)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOWAIT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>NULLIF</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>NULLS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMBER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OBJECT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>OCTETS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCTET_LENGTH</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OF</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OFF</td>
<td>reserved</td>
<td></td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>OFFSET</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OIDS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OLD</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>ON</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ONLY</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OPEN</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OPERATION</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPERATOR</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPTION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ORDER</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ORDERING</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDINALITY</td>
<td>non-reserved</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>OTHERS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------</td>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>OUT</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>OUTER</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OVER</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVERLAPS</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>OVERRIDING</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>OWNED</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OWNER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAD</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARAMETERS</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARAMETER_MODE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>PARAMETER_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>PARAMETER_ORDINAL_POSITION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>PARAMETER_SPECIFIC_CATALOG</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>PARAMETER_SPECIFIC_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>PARAMETER_SPECIFIC_SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>PARSER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARTIAL</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>PARTITION</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PASCAL</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PATH</td>
<td></td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>PERCENTILE_CONT</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PERCENTILE_DISC</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PERCENT_RANK</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLACING</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLANS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLI</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------</td>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>POSITION</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td></td>
<td>(cannot be function or type)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POSTFIX</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POWER</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRECEDING</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRECISION</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td></td>
<td>(cannot be function or type)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREFIX</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREORDER</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREPARE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>PREPARED</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRESERVE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>PRIMARY</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>PRIOR</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>PRIVILEGES</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>PROCEDURAL</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>QUOTE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANGE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANK</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>READS</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td></td>
<td>(cannot be function or type)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REASSIGN</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECHECK</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECURSIVE</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REF</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFERENCES</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>REFERENCING</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGR_AVGX</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGR_AVGY</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGR_COUNT</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGR_INTERCEPT</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGR_R2</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGR_SLOPE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGR_SXX</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REGR_SXY</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------</td>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>REGR_SYY</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REINDEX</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>RELATIVE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>RELEASE</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RENAME</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPEATABLE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>REPLACE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPLICA</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESET</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESTART</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>RESTRICT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>RESULT</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>RETURNED_CARDINALITY</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETURNED_LENGTH</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>RETURNED_OCTET_LENGTH</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>RETURNED_SQLSTATE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>RETURNING</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RIGHTS</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>REVOKE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>RIGHT</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ROLE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ROLLUP</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUTINE</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUTINE_CATALOG</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUTINE_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUTINE_SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROW</td>
<td>non-reserved (cannot be function or type)</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>ROWS</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>ROW_COUNT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>ROW_NUMBER</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RULE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAVEPOINT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SCHEMA_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SCOPE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SCOPE_CATALOG</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SCOPE_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SCOPE_SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SCROLL</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SEARCH</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SECOND</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SECTION</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SECURITY</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SELECT</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SELF</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SENSITIVE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SERVER_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SESSION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SESSION_USER</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SET</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SETOF</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SETS</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SHARE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SHOW</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SIMILAR</td>
<td>reserved (can be function or type)</td>
<td>reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SIMPLE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SIZE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SOME</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SOURCE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SPACE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SPECIFIC</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SPECIFICITY</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SPECIFIC_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SQL</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>
### Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLERROR</td>
<td></td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SQLEXCEPTION</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SQLWARNING</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SQRT</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STABLE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STANDALONE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>START</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>STATE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>STATIC</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>STATISTICS</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STDDEV_POP</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STDDEV_SAMP</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STDIN</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STDOUT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORAGE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRICT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRIP</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>STYLE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>SUBCLASS_ORIGIN</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SUBLIST</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBMULTISET</td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>SUBSTRING</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>SUM</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SUPERUSER</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYMMETRIC</td>
<td>reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>SYSSID</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSTEM</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>SYSTEM_USER</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TABLE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TABLESAMPLE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TABLESPACE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>TEMP</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEMPLATE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEMPORARY</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Word</td>
<td>PostgreSQL</td>
<td>SQL:2003</td>
<td>SQL:1999</td>
<td>SQL-92</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------</td>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>TEXT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>THAN</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>THEN</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TIES</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TIMEZONE_HOUR</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TIMEZONE_MINUTE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TO</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TOP_LEVEL_COUNT</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAILING</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TRANSACTION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TRANSACTIONS_COMMITTED</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
</tr>
<tr>
<td>TRANSACTIONS_ROLLED_BACK</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSACTION_ACTIVE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSFORMS</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSLATE</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>TRANSLATION</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>TREAT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>TRIGGER</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>TRIGGER_CATALOG</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIGGER_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIGGER_SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIM</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TRUE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUSTED</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
</tbody>
</table>
## Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>UESCAPE</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNBOUNDED</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>UNCOMMITTED</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>UNDER</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td>reserved</td>
</tr>
<tr>
<td>UNENCRIPTED</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNION</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>UNLISTEN</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNNAMED</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>non-reserved</td>
</tr>
<tr>
<td>UNNEST</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>UNTIL</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>UPPER</td>
<td>reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>USAGE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>USER</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>USER_DEFINED_TYPE_CATALOG</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER_DEFINED_TYPE_CODE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER_DEFINED_TYPE_NAME</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER_DEFINED_TYPE_SCHEMA</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USING</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>VACUUM</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VALID</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VALIDATOR</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VALUE</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>VALUES</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>VARIABLE</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARYING</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>VAR_POP</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VAR_SAMP</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERBOSE</td>
<td>reserved (can be function or type)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERSION</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIEW</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>
## Appendix C. SQL Key Words

<table>
<thead>
<tr>
<th>Key Word</th>
<th>PostgreSQL</th>
<th>SQL:2003</th>
<th>SQL:1999</th>
<th>SQL-92</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOLATILE</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHEN</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>WHENEVER</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>WHERE</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>WHITESPACE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WIDTH_BUCKET</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WINDOW</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WITH</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>WITHIN</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WITHOUT</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>WORK</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>WRITE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>XML</td>
<td>non-reserved</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLAGG</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLATTRIBUTES</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>XMLBINARY</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLCOMMENT</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLCONCAT</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>XMLELEMENT</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>XMLFOREST</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>XMLNAMESPACES</td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLPARSE</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>XMLPI</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>XMLROOT</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>XMLSERIALIZE</td>
<td>non-reserved</td>
<td>(cannot be function or type)</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>YEAR</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>YES</td>
<td>non-reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZONE</td>
<td>non-reserved</td>
<td>non-reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>
Appendix D. SQL Conformance

This section attempts to outline to what extent PostgreSQL conforms to the current SQL standard. The following information is not a full statement of conformance, but it presents the main topics in as much detail as is both reasonable and useful for users.

The formal name of the SQL standard is ISO/IEC 9075 “Database Language SQL”. A revised version of the standard is released from time to time; the most recent full update appearing in 2003, with a partial update appearing in 2006. The 2003 version is referred to as ISO/IEC 9075:2003, or simply as SQL:2003. The versions prior to that were SQL:1999 and SQL-92. Each version replaces the previous one, so claims of conformance to earlier versions have no official merit. PostgreSQL development aims for conformance with the latest official version of the standard where such conformance does not contradict traditional features or common sense. The PostgreSQL project is not represented in the ISO/IEC 9075 Working Group during the preparation of the SQL standard releases, but even so, many of the features required by the SQL standard are supported, though sometimes with slightly differing syntax or function. Further moves towards conformance can be expected over time.

SQL-92 defined three feature sets for conformance: Entry, Intermediate, and Full. Most database management systems claiming SQL standard conformance were conforming at only the Entry level, since the entire set of features in the Intermediate and Full levels was either too voluminous or in conflict with legacy behaviors.

Starting with SQL:1999, the SQL standard defines a large set of individual features rather than the ineffectively broad three levels found in SQL-92. A large subset of these features represents the “Core” features, which every conforming SQL implementation must supply. The rest of the features are purely optional. Some optional features are grouped together to form “packages”, which SQL implementations can claim conformance to, thus claiming conformance to particular groups of features.

The SQL:2003 standard is also split into a number of parts. Each is known by a shorthand name. Note that these parts are not consecutively numbered.

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

PostgreSQL covers parts 1, 2, 11, and 14. Part 3 is similar to the ODBC interface, and part 4 is similar to the PL/pgSQL programming language, but exact conformance is not specifically intended or verified in either case.

PostgreSQL supports most of the major features of SQL:2003. Out of 164 mandatory features required for full Core conformance, PostgreSQL conforms to at least 150. In addition, there is a long list of
supported optional features. It might be worth noting that at the time of writing, no current version of any database management system claims full conformance to Core SQL:2003.

In the following two sections, we provide a list of those features that PostgreSQL supports, followed by a list of the features defined in SQL:2003 which are not yet supported in PostgreSQL. Both of these lists are approximate: There might be minor details that are nonconforming for a feature that is listed as supported, and large parts of an unsupported feature might in fact be implemented. The main body of the documentation always contains the most accurate information about what does and does not work.

**Note:** Feature codes containing a hyphen are subfeatures. Therefore, if a particular subfeature is not supported, the main feature is listed as unsupported even if some other subfeatures are supported.

### D.1. Supported Features

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Package</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>B012</td>
<td></td>
<td>Embedded C</td>
<td></td>
</tr>
<tr>
<td>B021</td>
<td></td>
<td>Direct SQL</td>
<td></td>
</tr>
<tr>
<td>E011</td>
<td>Core</td>
<td>Numeric data types</td>
<td></td>
</tr>
<tr>
<td>E011-01</td>
<td>Core</td>
<td>INTEGER and SMALLINT data types</td>
<td></td>
</tr>
<tr>
<td>E011-02</td>
<td>Core</td>
<td>REAL, DOUBLE PRECISION, and FLOAT data types</td>
<td></td>
</tr>
<tr>
<td>E011-03</td>
<td>Core</td>
<td>DECIMAL and NUMERIC data types</td>
<td></td>
</tr>
<tr>
<td>E011-04</td>
<td>Core</td>
<td>Arithmetic operators</td>
<td></td>
</tr>
<tr>
<td>E011-05</td>
<td>Core</td>
<td>Numeric comparison</td>
<td></td>
</tr>
<tr>
<td>E011-06</td>
<td>Core</td>
<td>Implicit casting among the numeric data types</td>
<td></td>
</tr>
<tr>
<td>E021</td>
<td>Core</td>
<td>Character data types</td>
<td></td>
</tr>
<tr>
<td>E021-01</td>
<td>Core</td>
<td>CHARACTER data type</td>
<td></td>
</tr>
<tr>
<td>E021-02</td>
<td>Core</td>
<td>CHARACTER VARYING data type</td>
<td></td>
</tr>
<tr>
<td>E021-03</td>
<td>Core</td>
<td>Character literals</td>
<td></td>
</tr>
<tr>
<td>E021-04</td>
<td>Core</td>
<td>CHARACTER_LENGTH function</td>
<td>Trims trailing spaces from CHARACTER values before counting</td>
</tr>
<tr>
<td>E021-05</td>
<td>Core</td>
<td>OCTET_LENGTH function</td>
<td></td>
</tr>
<tr>
<td>E021-06</td>
<td>Core</td>
<td>SUBSTRING function</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-----------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>E021-07</td>
<td>Core</td>
<td>Character concatenation</td>
<td></td>
</tr>
<tr>
<td>E021-08</td>
<td>Core</td>
<td>UPPER and LOWER functions</td>
<td></td>
</tr>
<tr>
<td>E021-09</td>
<td>Core</td>
<td>TRIM function</td>
<td></td>
</tr>
<tr>
<td>E021-10</td>
<td>Core</td>
<td>Implicit casting among the character string types</td>
<td></td>
</tr>
<tr>
<td>E021-11</td>
<td>Core</td>
<td>POSITION function</td>
<td></td>
</tr>
<tr>
<td>E021-12</td>
<td>Core</td>
<td>Character comparison</td>
<td></td>
</tr>
<tr>
<td>E031</td>
<td>Core</td>
<td>Identifiers</td>
<td></td>
</tr>
<tr>
<td>E031-01</td>
<td>Core</td>
<td>Delimited identifiers</td>
<td></td>
</tr>
<tr>
<td>E031-02</td>
<td>Core</td>
<td>Lower case identifiers</td>
<td></td>
</tr>
<tr>
<td>E031-03</td>
<td>Core</td>
<td>Trailing underscore</td>
<td></td>
</tr>
<tr>
<td>E051</td>
<td>Core</td>
<td>Basic query specification</td>
<td></td>
</tr>
<tr>
<td>E051-01</td>
<td>Core</td>
<td>SELECT DISTINCT</td>
<td></td>
</tr>
<tr>
<td>E051-02</td>
<td>Core</td>
<td>GROUP BY clause</td>
<td></td>
</tr>
<tr>
<td>E051-04</td>
<td>Core</td>
<td>GROUP BY can contain columns not in &lt;select list&gt;</td>
<td></td>
</tr>
<tr>
<td>E051-05</td>
<td>Core</td>
<td>Select list items can be renamed</td>
<td>AS is required</td>
</tr>
<tr>
<td>E051-06</td>
<td>Core</td>
<td>HAVING clause</td>
<td></td>
</tr>
<tr>
<td>E051-07</td>
<td>Core</td>
<td>Qualified * in select list</td>
<td></td>
</tr>
<tr>
<td>E051-08</td>
<td>Core</td>
<td>Correlation names in the FROM clause</td>
<td></td>
</tr>
<tr>
<td>E051-09</td>
<td>Core</td>
<td>Rename columns in the FROM clause</td>
<td></td>
</tr>
<tr>
<td>E061</td>
<td>Core</td>
<td>Basic predicates and search conditions</td>
<td></td>
</tr>
<tr>
<td>E061-01</td>
<td>Core</td>
<td>Comparison predicate</td>
<td></td>
</tr>
<tr>
<td>E061-02</td>
<td>Core</td>
<td>BETWEEN predicate</td>
<td></td>
</tr>
<tr>
<td>E061-03</td>
<td>Core</td>
<td>IN predicate with list of values</td>
<td></td>
</tr>
<tr>
<td>E061-04</td>
<td>Core</td>
<td>LIKE predicate</td>
<td></td>
</tr>
<tr>
<td>E061-05</td>
<td>Core</td>
<td>LIKE predicate ESCAPE clause</td>
<td></td>
</tr>
<tr>
<td>E061-06</td>
<td>Core</td>
<td>NULL predicate</td>
<td></td>
</tr>
<tr>
<td>E061-07</td>
<td>Core</td>
<td>Quantified comparison predicate</td>
<td></td>
</tr>
<tr>
<td>E061-08</td>
<td>Core</td>
<td>EXISTS predicate</td>
<td></td>
</tr>
<tr>
<td>E061-09</td>
<td>Core</td>
<td>Subqueries in comparison predicate</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>E061-11</td>
<td>Core</td>
<td>Subqueries in IN predicate</td>
<td></td>
</tr>
<tr>
<td>E061-12</td>
<td>Core</td>
<td>Subqueries in quantified comparison predicate</td>
<td></td>
</tr>
<tr>
<td>E061-13</td>
<td>Core</td>
<td>Correlated subqueries</td>
<td></td>
</tr>
<tr>
<td>E061-14</td>
<td>Core</td>
<td>Search condition</td>
<td></td>
</tr>
<tr>
<td>E071</td>
<td>Core</td>
<td>Basic query expressions</td>
<td></td>
</tr>
<tr>
<td>E071-01</td>
<td>Core</td>
<td>UNION DISTINCT table operator</td>
<td></td>
</tr>
<tr>
<td>E071-02</td>
<td>Core</td>
<td>UNION ALL table operator</td>
<td></td>
</tr>
<tr>
<td>E071-03</td>
<td>Core</td>
<td>EXCEPT DISTINCT table operator</td>
<td></td>
</tr>
<tr>
<td>E071-05</td>
<td>Core</td>
<td>Columns combined via table operators need not have exactly the same data type</td>
<td></td>
</tr>
<tr>
<td>E071-06</td>
<td>Core</td>
<td>Table operators in subqueries</td>
<td></td>
</tr>
<tr>
<td>E081-01</td>
<td>Core</td>
<td>SELECT privilege</td>
<td></td>
</tr>
<tr>
<td>E081-02</td>
<td>Core</td>
<td>DELETE privilege</td>
<td></td>
</tr>
<tr>
<td>E081-03</td>
<td>Core</td>
<td>INSERT privilege at the table level</td>
<td></td>
</tr>
<tr>
<td>E081-04</td>
<td>Core</td>
<td>UPDATE privilege at the table level</td>
<td></td>
</tr>
<tr>
<td>E081-06</td>
<td>Core</td>
<td>REFERENCES privilege at the table level</td>
<td></td>
</tr>
<tr>
<td>E081-08</td>
<td>Core</td>
<td>WITH GRANT OPTION</td>
<td></td>
</tr>
<tr>
<td>E081-10</td>
<td>Core</td>
<td>EXECUTE privilege</td>
<td></td>
</tr>
<tr>
<td>E091</td>
<td>Core</td>
<td>Set functions</td>
<td></td>
</tr>
<tr>
<td>E091-01</td>
<td>Core</td>
<td>AVG</td>
<td></td>
</tr>
<tr>
<td>E091-02</td>
<td>Core</td>
<td>COUNT</td>
<td></td>
</tr>
<tr>
<td>E091-03</td>
<td>Core</td>
<td>MAX</td>
<td></td>
</tr>
<tr>
<td>E091-04</td>
<td>Core</td>
<td>MIN</td>
<td></td>
</tr>
<tr>
<td>E091-05</td>
<td>Core</td>
<td>SUM</td>
<td></td>
</tr>
<tr>
<td>E091-06</td>
<td>Core</td>
<td>ALL quantifier</td>
<td></td>
</tr>
<tr>
<td>E091-07</td>
<td>Core</td>
<td>DISTINCT quantifier</td>
<td></td>
</tr>
<tr>
<td>E101</td>
<td>Core</td>
<td>Basic data manipulation</td>
<td></td>
</tr>
<tr>
<td>E101-01</td>
<td>Core</td>
<td>INSERT statement</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>E101-03</td>
<td>Core</td>
<td>Searched UPDATE statement</td>
<td></td>
</tr>
<tr>
<td>E101-04</td>
<td>Core</td>
<td>Searched DELETE statement</td>
<td></td>
</tr>
<tr>
<td>E111</td>
<td>Core</td>
<td>Single row SELECT statement</td>
<td></td>
</tr>
<tr>
<td>E121-01</td>
<td>Core</td>
<td>DECLARE CURSOR</td>
<td></td>
</tr>
<tr>
<td>E121-02</td>
<td>Core</td>
<td>ORDER BY columns need not be in select list</td>
<td></td>
</tr>
<tr>
<td>E121-03</td>
<td>Core</td>
<td>Value expressions in ORDER BY clause</td>
<td></td>
</tr>
<tr>
<td>E121-04</td>
<td>Core</td>
<td>OPEN statement</td>
<td></td>
</tr>
<tr>
<td>E121-08</td>
<td>Core</td>
<td>CLOSE statement</td>
<td></td>
</tr>
<tr>
<td>E121-10</td>
<td>Core</td>
<td>FETCH statement implicit NEXT</td>
<td></td>
</tr>
<tr>
<td>E121-17</td>
<td>Core</td>
<td>WITH HOLD cursors</td>
<td></td>
</tr>
<tr>
<td>E131</td>
<td>Core</td>
<td>Null value support (nulls in lieu of values)</td>
<td></td>
</tr>
<tr>
<td>E141</td>
<td>Core</td>
<td>Basic integrity constraints</td>
<td></td>
</tr>
<tr>
<td>E141-01</td>
<td>Core</td>
<td>NOT NULL constraints</td>
<td></td>
</tr>
<tr>
<td>E141-02</td>
<td>Core</td>
<td>UNIQUE constraints of NOT NULL columns</td>
<td></td>
</tr>
<tr>
<td>E141-03</td>
<td>Core</td>
<td>PRIMARY KEY constraints</td>
<td></td>
</tr>
<tr>
<td>E141-04</td>
<td>Core</td>
<td>Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action</td>
<td></td>
</tr>
<tr>
<td>E141-06</td>
<td>Core</td>
<td>CHECK constraints</td>
<td></td>
</tr>
<tr>
<td>E141-07</td>
<td>Core</td>
<td>Column defaults</td>
<td></td>
</tr>
<tr>
<td>E141-08</td>
<td>Core</td>
<td>NOT NULL inferred on PRIMARY KEY</td>
<td></td>
</tr>
<tr>
<td>E141-10</td>
<td>Core</td>
<td>Names in a foreign key can be specified in any order</td>
<td></td>
</tr>
<tr>
<td>E151</td>
<td>Core</td>
<td>Transaction support</td>
<td></td>
</tr>
<tr>
<td>E151-01</td>
<td>Core</td>
<td>COMMIT statement</td>
<td></td>
</tr>
<tr>
<td>E151-02</td>
<td>Core</td>
<td>ROLLBACK statement</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>E152</td>
<td>Core</td>
<td>Basic SET TRANSACTION statement</td>
<td></td>
</tr>
<tr>
<td>E152-01</td>
<td>Core</td>
<td>SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause</td>
<td></td>
</tr>
<tr>
<td>E152-02</td>
<td>Core</td>
<td>SET TRANSACTION statement: READ ONLY and READ WRITE clauses</td>
<td></td>
</tr>
<tr>
<td>E161</td>
<td>Core</td>
<td>SQL comments using leading double minus</td>
<td></td>
</tr>
<tr>
<td>E171</td>
<td>Core</td>
<td>SQLSTATE support</td>
<td></td>
</tr>
<tr>
<td>F021</td>
<td>Core</td>
<td>Basic information schema</td>
<td></td>
</tr>
<tr>
<td>F021-01</td>
<td>Core</td>
<td>COLUMNS view</td>
<td></td>
</tr>
<tr>
<td>F021-02</td>
<td>Core</td>
<td>TABLES view</td>
<td></td>
</tr>
<tr>
<td>F021-03</td>
<td>Core</td>
<td>VIEWS view</td>
<td></td>
</tr>
<tr>
<td>F021-04</td>
<td>Core</td>
<td>TABLE_CONSTRAINTS view</td>
<td></td>
</tr>
<tr>
<td>F021-05</td>
<td>Core</td>
<td>REFERENTIAL_CONSTRAINTS view</td>
<td></td>
</tr>
<tr>
<td>F021-06</td>
<td>Core</td>
<td>CHECK_CONSTRAINTS view</td>
<td></td>
</tr>
<tr>
<td>F031</td>
<td>Core</td>
<td>Basic schema manipulation</td>
<td></td>
</tr>
<tr>
<td>F031-01</td>
<td>Core</td>
<td>CREATE TABLE statement to create persistent base tables</td>
<td></td>
</tr>
<tr>
<td>F031-02</td>
<td>Core</td>
<td>CREATE VIEW statement</td>
<td></td>
</tr>
<tr>
<td>F031-03</td>
<td>Core</td>
<td>GRANT statement</td>
<td></td>
</tr>
<tr>
<td>F031-04</td>
<td>Core</td>
<td>ALTER TABLE statement: ADD COLUMN clause</td>
<td></td>
</tr>
<tr>
<td>F031-13</td>
<td>Core</td>
<td>DROP TABLE statement: RESTRICT clause</td>
<td></td>
</tr>
<tr>
<td>F031-16</td>
<td>Core</td>
<td>DROP VIEW statement: RESTRICT clause</td>
<td></td>
</tr>
<tr>
<td>F031-19</td>
<td>Core</td>
<td>REVOKE statement: RESTRICT clause</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>F032</td>
<td>Package</td>
<td>CASCADE drop behavior</td>
<td></td>
</tr>
<tr>
<td>F033</td>
<td>Package</td>
<td>ALTER TABLE statement: DROP COLUMN clause</td>
<td></td>
</tr>
<tr>
<td>F034</td>
<td>Package</td>
<td>Extended REVOKE statement</td>
<td></td>
</tr>
<tr>
<td>F034-01</td>
<td>Package</td>
<td>REVOKE statement performed by other than the owner of a schema object</td>
<td></td>
</tr>
<tr>
<td>F034-02</td>
<td>Package</td>
<td>REVOKE statement: GRANT OPTION FOR clause</td>
<td></td>
</tr>
<tr>
<td>F034-03</td>
<td>Package</td>
<td>REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION</td>
<td></td>
</tr>
<tr>
<td>F041</td>
<td>Core</td>
<td>Basic joined table</td>
<td></td>
</tr>
<tr>
<td>F041-01</td>
<td>Core</td>
<td>Inner join (but not necessarily the INNER keyword)</td>
<td></td>
</tr>
<tr>
<td>F041-02</td>
<td>Core</td>
<td>INNER keyword</td>
<td></td>
</tr>
<tr>
<td>F041-03</td>
<td>Core</td>
<td>LEFT OUTER JOIN</td>
<td></td>
</tr>
<tr>
<td>F041-04</td>
<td>Core</td>
<td>RIGHT OUTER JOIN</td>
<td></td>
</tr>
<tr>
<td>F041-05</td>
<td>Core</td>
<td>Outer joins can be nested</td>
<td></td>
</tr>
<tr>
<td>F041-07</td>
<td>Core</td>
<td>The inner table in a left or right outer join can also be used in an inner join</td>
<td></td>
</tr>
<tr>
<td>F041-08</td>
<td>Core</td>
<td>All comparison operators are supported (rather than just =)</td>
<td></td>
</tr>
<tr>
<td>F051</td>
<td>Core</td>
<td>Basic date and time</td>
<td></td>
</tr>
<tr>
<td>F051-01</td>
<td>Core</td>
<td>DATE data type (including support of DATE literal)</td>
<td></td>
</tr>
<tr>
<td>F051-02</td>
<td>Core</td>
<td>TIME data type (including support of TIME literal) with fractional seconds precision of at least 0</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>F051-03</td>
<td>Core</td>
<td>TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6</td>
<td></td>
</tr>
<tr>
<td>F051-04</td>
<td>Core</td>
<td>Comparison predicate on DATE, TIME, and TIMESTAMP data types</td>
<td></td>
</tr>
<tr>
<td>F051-05</td>
<td>Core</td>
<td>Explicit CAST between datetime types and character string types</td>
<td></td>
</tr>
<tr>
<td>F051-06</td>
<td>Core</td>
<td>CURRENT_DATE</td>
<td></td>
</tr>
<tr>
<td>F051-07</td>
<td>Core</td>
<td>LOCALTIME</td>
<td></td>
</tr>
<tr>
<td>F051-08</td>
<td>Core</td>
<td>LOCALTIMESTAMP</td>
<td></td>
</tr>
<tr>
<td>F052</td>
<td>Enhanced datetime facilities</td>
<td>Intervals and datetime arithmetic</td>
<td></td>
</tr>
<tr>
<td>F053</td>
<td>Core</td>
<td>OVERLAPS predicate</td>
<td></td>
</tr>
<tr>
<td>F081</td>
<td>Core</td>
<td>UNION and EXCEPT in views</td>
<td></td>
</tr>
<tr>
<td>F111</td>
<td>Core</td>
<td>Isolation levels other than SERIALIZABLE</td>
<td></td>
</tr>
<tr>
<td>F111-01</td>
<td>Core</td>
<td>READ UNCOMMITTED isolation level</td>
<td></td>
</tr>
<tr>
<td>F111-02</td>
<td>Core</td>
<td>READ COMMITTED isolation level</td>
<td></td>
</tr>
<tr>
<td>F111-03</td>
<td>Core</td>
<td>REPEATABLE READ isolation level</td>
<td></td>
</tr>
<tr>
<td>F131</td>
<td>Core</td>
<td>Grouped operations</td>
<td></td>
</tr>
<tr>
<td>F131-01</td>
<td>Core</td>
<td>WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views</td>
<td></td>
</tr>
<tr>
<td>F131-02</td>
<td>Core</td>
<td>Multiple tables supported in queries with grouped views</td>
<td></td>
</tr>
<tr>
<td>F131-03</td>
<td>Core</td>
<td>Set functions supported in queries with grouped views</td>
<td></td>
</tr>
<tr>
<td>F131-04</td>
<td>Core</td>
<td>Subqueries with GROUP BY and HAVING clauses and grouped views</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>F131-05</td>
<td>Core</td>
<td>Single row SELECT with GROUP BY and HAVING clauses and grouped views</td>
<td></td>
</tr>
<tr>
<td>F171</td>
<td></td>
<td>Multiple schemas per user</td>
<td></td>
</tr>
<tr>
<td>F191</td>
<td>Enhanced integrity management</td>
<td>Referential delete actions</td>
<td></td>
</tr>
<tr>
<td>F201</td>
<td>Core</td>
<td>CAST function</td>
<td></td>
</tr>
<tr>
<td>F221</td>
<td>Core</td>
<td>Explicit defaults</td>
<td></td>
</tr>
<tr>
<td>F222</td>
<td></td>
<td>INSERT statement: DEFAULT VALUES clause</td>
<td></td>
</tr>
<tr>
<td>F231</td>
<td></td>
<td>Privilege tables</td>
<td></td>
</tr>
<tr>
<td>F231-01</td>
<td></td>
<td>TABLE_PRIVILEGES view</td>
<td></td>
</tr>
<tr>
<td>F231-02</td>
<td></td>
<td>COLUMN_PRIVILEGES view</td>
<td></td>
</tr>
<tr>
<td>F231-03</td>
<td></td>
<td>USAGE_PRIVILEGES view</td>
<td></td>
</tr>
<tr>
<td>F251</td>
<td></td>
<td>Domain support</td>
<td></td>
</tr>
<tr>
<td>F261</td>
<td>Core</td>
<td>CASE expression</td>
<td></td>
</tr>
<tr>
<td>F261-01</td>
<td>Core</td>
<td>Simple CASE</td>
<td></td>
</tr>
<tr>
<td>F261-02</td>
<td>Core</td>
<td>Searched CASE</td>
<td></td>
</tr>
<tr>
<td>F261-03</td>
<td>Core</td>
<td>NULLIF</td>
<td></td>
</tr>
<tr>
<td>F261-04</td>
<td>Core</td>
<td>COALESCE</td>
<td></td>
</tr>
<tr>
<td>F271</td>
<td></td>
<td>Compound character literals</td>
<td></td>
</tr>
<tr>
<td>F281</td>
<td></td>
<td>LIKE enhancements</td>
<td></td>
</tr>
<tr>
<td>F302</td>
<td></td>
<td>INTERSECT table operator</td>
<td></td>
</tr>
<tr>
<td>F302-01</td>
<td></td>
<td>INTERSECT DISTINCT table operator</td>
<td></td>
</tr>
<tr>
<td>F302-02</td>
<td></td>
<td>INTERSECT ALL table operator</td>
<td></td>
</tr>
<tr>
<td>F304</td>
<td></td>
<td>EXCEPT ALL table operator</td>
<td></td>
</tr>
<tr>
<td>F311-01</td>
<td>Core</td>
<td>CREATE SCHEMA</td>
<td></td>
</tr>
<tr>
<td>F311-02</td>
<td>Core</td>
<td>CREATE TABLE for persistent base tables</td>
<td></td>
</tr>
<tr>
<td>F311-03</td>
<td>Core</td>
<td>CREATE VIEW</td>
<td></td>
</tr>
<tr>
<td>F311-05</td>
<td>Core</td>
<td>GRANT statement</td>
<td></td>
</tr>
<tr>
<td>F321</td>
<td></td>
<td>User authorization</td>
<td></td>
</tr>
<tr>
<td>F361</td>
<td></td>
<td>Subprogram support</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>F381</td>
<td></td>
<td>Extended schema manipulation</td>
<td></td>
</tr>
<tr>
<td>F381-01</td>
<td></td>
<td>ALTER TABLE statement: ALTER COLUMN clause</td>
<td></td>
</tr>
<tr>
<td>F381-02</td>
<td></td>
<td>ALTER TABLE statement: ADD CONSTRAINT clause</td>
<td></td>
</tr>
<tr>
<td>F381-03</td>
<td></td>
<td>ALTER TABLE statement: DROP CONSTRAINT clause</td>
<td></td>
</tr>
<tr>
<td>F391</td>
<td></td>
<td>Long identifiers</td>
<td></td>
</tr>
<tr>
<td>F401</td>
<td></td>
<td>Extended joined table</td>
<td></td>
</tr>
<tr>
<td>F401-01</td>
<td></td>
<td>NATURAL JOIN</td>
<td></td>
</tr>
<tr>
<td>F401-02</td>
<td></td>
<td>FULL OUTER JOIN</td>
<td></td>
</tr>
<tr>
<td>F401-04</td>
<td></td>
<td>CROSS JOIN</td>
<td></td>
</tr>
<tr>
<td>F411</td>
<td></td>
<td>Enhanced datetime facilities</td>
<td>Time zone specification differences regarding literal interpretation</td>
</tr>
<tr>
<td>F421</td>
<td></td>
<td>National character</td>
<td></td>
</tr>
<tr>
<td>F431</td>
<td></td>
<td>Read-only scrollable cursors</td>
<td></td>
</tr>
<tr>
<td>F431-01</td>
<td></td>
<td>FETCH with explicit NEXT</td>
<td></td>
</tr>
<tr>
<td>F431-02</td>
<td></td>
<td>FETCH FIRST</td>
<td></td>
</tr>
<tr>
<td>F431-03</td>
<td></td>
<td>FETCH LAST</td>
<td></td>
</tr>
<tr>
<td>F431-04</td>
<td></td>
<td>FETCH PRIOR</td>
<td></td>
</tr>
<tr>
<td>F431-05</td>
<td></td>
<td>FETCH ABSOLUTE</td>
<td></td>
</tr>
<tr>
<td>F431-06</td>
<td></td>
<td>FETCH RELATIVE</td>
<td></td>
</tr>
<tr>
<td>F441</td>
<td></td>
<td>Extended set function support</td>
<td></td>
</tr>
<tr>
<td>F471</td>
<td></td>
<td>Core</td>
<td>Scalar subquery values</td>
</tr>
<tr>
<td>F481</td>
<td></td>
<td>Core</td>
<td>Expanded NULL predicate</td>
</tr>
<tr>
<td>F491</td>
<td></td>
<td>Enhanced integrity management</td>
<td>Constraint management</td>
</tr>
<tr>
<td>F501</td>
<td></td>
<td>Core</td>
<td>Features and conformance views</td>
</tr>
<tr>
<td>F501-01</td>
<td></td>
<td>Core</td>
<td>SQL_FEATURES view</td>
</tr>
<tr>
<td>F501-02</td>
<td></td>
<td>Core</td>
<td>SQL_SIZING view</td>
</tr>
<tr>
<td>F501-03</td>
<td></td>
<td>Core</td>
<td>SQL_LANGUAGES view</td>
</tr>
<tr>
<td>F502</td>
<td></td>
<td>Enhanced documentation tables</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>F502-01</td>
<td>SQL_SIZING_PROFILES</td>
<td>view</td>
<td></td>
</tr>
<tr>
<td>F502-02</td>
<td>SQL_IMPLEMENTATION_INFO</td>
<td>view</td>
<td></td>
</tr>
<tr>
<td>F502-03</td>
<td>SQL_PACKAGES</td>
<td>view</td>
<td></td>
</tr>
<tr>
<td>F531</td>
<td>Temporary tables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F555</td>
<td>Enhanced datetime facilities</td>
<td>Enhanced seconds precision</td>
<td></td>
</tr>
<tr>
<td>F561</td>
<td>Full value expressions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F571</td>
<td>Truth value tests</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F591</td>
<td>Derived tables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F611</td>
<td>Indicator data types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F651</td>
<td>Catalog name qualifiers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F672</td>
<td>Retrospective check constraints</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F701</td>
<td>Enhanced integrity management</td>
<td>Referential update actions</td>
<td></td>
</tr>
<tr>
<td>F711</td>
<td>ALTER domain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F761</td>
<td>Session management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F771</td>
<td>Connection management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F781</td>
<td>Self-referencing operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F791</td>
<td>Insensitive cursors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F801</td>
<td>Full set function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S071</td>
<td>Enhanced object support</td>
<td>SQL paths in function and type name resolution</td>
<td></td>
</tr>
<tr>
<td>S111</td>
<td>Enhanced object support</td>
<td>ONLY in query expressions</td>
<td></td>
</tr>
<tr>
<td>S211</td>
<td>Enhanced object support</td>
<td>User-defined cast functions</td>
<td></td>
</tr>
<tr>
<td>T031</td>
<td>BOOLEAN data type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T071</td>
<td>BIGINT data type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T141</td>
<td>SIMILAR predicate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T151</td>
<td>DISTINCT predicate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T171</td>
<td>LIKE clause in table definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T191</td>
<td>Enhanced integrity management</td>
<td>Referential action RESTRICT</td>
<td></td>
</tr>
<tr>
<td>T201</td>
<td>Enhanced integrity management</td>
<td>Comparable data types for referential constraints</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>T211-01</td>
<td>Active database, Enhanced integrity management</td>
<td>Triggers activated on UPDATE, INSERT, or DELETE of one base table</td>
<td></td>
</tr>
<tr>
<td>T211-02</td>
<td>Active database, Enhanced integrity management</td>
<td>BEFORE triggers</td>
<td></td>
</tr>
<tr>
<td>T211-03</td>
<td>Active database, Enhanced integrity management</td>
<td>AFTER triggers</td>
<td></td>
</tr>
<tr>
<td>T211-04</td>
<td>Active database, Enhanced integrity management</td>
<td>FOR EACH ROW triggers</td>
<td></td>
</tr>
<tr>
<td>T211-07</td>
<td>Active database, Enhanced integrity management</td>
<td>TRIGGER privilege</td>
<td></td>
</tr>
<tr>
<td>T212</td>
<td>Enhanced integrity management</td>
<td>Enhanced trigger capability</td>
<td></td>
</tr>
<tr>
<td>T231</td>
<td>Enhanced integrity management</td>
<td>Sensitive cursors</td>
<td></td>
</tr>
<tr>
<td>T241</td>
<td>Enhanced integrity management</td>
<td>START TRANSACTION statement</td>
<td></td>
</tr>
<tr>
<td>T271</td>
<td>Enhanced integrity management</td>
<td>Savepoints</td>
<td></td>
</tr>
<tr>
<td>T312</td>
<td>Enhanced integrity management</td>
<td>OVERLAY function</td>
<td></td>
</tr>
<tr>
<td>T321-01</td>
<td>Core</td>
<td>User-defined functions with no overloading</td>
<td></td>
</tr>
<tr>
<td>T321-03</td>
<td>Core</td>
<td>Function invocation</td>
<td></td>
</tr>
<tr>
<td>T321-06</td>
<td>Core</td>
<td>ROUTINES view</td>
<td></td>
</tr>
<tr>
<td>T321-07</td>
<td>Core</td>
<td>PARAMETERS view</td>
<td></td>
</tr>
<tr>
<td>T322</td>
<td>PSM</td>
<td>Overloading of SQL-invoked functions and procedures</td>
<td></td>
</tr>
<tr>
<td>T323</td>
<td>PSM</td>
<td>Explicit security for external routines</td>
<td></td>
</tr>
<tr>
<td>T351</td>
<td>PSM</td>
<td>Bracketed SQL comments (/<em>...</em>/ comments)</td>
<td></td>
</tr>
<tr>
<td>T441</td>
<td>PSM</td>
<td>ABS and MOD functions</td>
<td></td>
</tr>
<tr>
<td>T461</td>
<td>PSM</td>
<td>Symmetric BETWEEN predicate</td>
<td></td>
</tr>
<tr>
<td>T501</td>
<td>PSM</td>
<td>Enhanced EXISTS predicate</td>
<td></td>
</tr>
<tr>
<td>T551</td>
<td>PSM</td>
<td>Optional key words for default syntax</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>T581</td>
<td></td>
<td>Regular expression substring function</td>
<td></td>
</tr>
<tr>
<td>T591</td>
<td></td>
<td>UNIQUE constraints of possibly null columns</td>
<td></td>
</tr>
<tr>
<td>X010</td>
<td></td>
<td>XML type</td>
<td></td>
</tr>
<tr>
<td>X011</td>
<td></td>
<td>Arrays of XML type</td>
<td></td>
</tr>
<tr>
<td>X016</td>
<td></td>
<td>Persistent XML values</td>
<td></td>
</tr>
<tr>
<td>X020</td>
<td></td>
<td>XML concatenation</td>
<td></td>
</tr>
<tr>
<td>X031</td>
<td></td>
<td>XMLElement</td>
<td></td>
</tr>
<tr>
<td>X032</td>
<td></td>
<td>XMLForest</td>
<td></td>
</tr>
<tr>
<td>X033</td>
<td></td>
<td>XMLRoot</td>
<td></td>
</tr>
<tr>
<td>X034</td>
<td></td>
<td>XMLAgg</td>
<td></td>
</tr>
<tr>
<td>X036</td>
<td></td>
<td>XMLComment</td>
<td></td>
</tr>
<tr>
<td>X037</td>
<td></td>
<td>XMLPI</td>
<td></td>
</tr>
<tr>
<td>X041</td>
<td></td>
<td>Basic table mapping: nulls absent</td>
<td></td>
</tr>
<tr>
<td>X042</td>
<td></td>
<td>Basic table mapping: null as nil</td>
<td></td>
</tr>
<tr>
<td>X043</td>
<td></td>
<td>Basic table mapping: table as forest</td>
<td></td>
</tr>
<tr>
<td>X044</td>
<td></td>
<td>Basic table mapping: table as element</td>
<td></td>
</tr>
<tr>
<td>X045</td>
<td></td>
<td>Basic table mapping: table as element with target namespace</td>
<td></td>
</tr>
<tr>
<td>X046</td>
<td></td>
<td>Basic table mapping: data mapping</td>
<td></td>
</tr>
<tr>
<td>X047</td>
<td></td>
<td>Basic table mapping: metadata mapping</td>
<td></td>
</tr>
<tr>
<td>X048</td>
<td></td>
<td>Basic table mapping: base64 encoding of binary strings</td>
<td></td>
</tr>
<tr>
<td>X049</td>
<td></td>
<td>Basic table mapping: hex encoding of binary strings</td>
<td></td>
</tr>
<tr>
<td>X051</td>
<td></td>
<td>Advanced table mapping: nulls absent</td>
<td></td>
</tr>
<tr>
<td>X052</td>
<td></td>
<td>Advanced table mapping: null as nil</td>
<td></td>
</tr>
<tr>
<td>X053</td>
<td></td>
<td>Advanced table mapping: table as forest</td>
<td></td>
</tr>
<tr>
<td>X054</td>
<td></td>
<td>Advanced table mapping: table as element</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix D. SQL Conformance

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Package</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>X055</td>
<td>Advanced table mapping: target namespace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X056</td>
<td>Advanced table mapping: data mapping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X057</td>
<td>Advanced table mapping: metadata mapping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X058</td>
<td>Advanced table mapping: base64 encoding of binary strings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X059</td>
<td>Advanced table mapping: hex encoding of binary strings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X060</td>
<td>XMLParse: CONTENT option</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X061</td>
<td>XMLParse: DOCUMENT option</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X062</td>
<td>XMLParse: explicit WHITESPACE option</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X070</td>
<td>XMLSerialize: CONTENT option</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X071</td>
<td>XMLSerialize: DOCUMENT option</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X090</td>
<td>XML document predicate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X120</td>
<td>XML parameters in SQL routines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X121</td>
<td>XML parameters in external routines</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### D.2. Unsupported Features

The following features defined in SQL:2003 are not implemented in this release of PostgreSQL. In a few cases, equivalent functionality is available.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Package</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>B011</td>
<td>Embedded Ada</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B013</td>
<td>Embedded COBOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B014</td>
<td>Embedded Fortran</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B015</td>
<td>Embedded MUMPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>B016</td>
<td>Embedded Pascal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B017</td>
<td>Embedded PL/I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B031</td>
<td>Basic dynamic SQL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B032</td>
<td>Extended dynamic SQL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B032-01</td>
<td>&lt;describe input statement&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B033</td>
<td>Untyped SQL-invoked function arguments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B034</td>
<td>Dynamic specification of cursor attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B041</td>
<td>Extensions to embedded SQL exception declarations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B051</td>
<td>Enhanced execution rights</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B111</td>
<td>Module language Ada</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B112</td>
<td>Module language C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B113</td>
<td>Module language COBOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B114</td>
<td>Module language Fortran</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B115</td>
<td>Module language MUMPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B116</td>
<td>Module language Pascal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B117</td>
<td>Module language PL/I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B121</td>
<td>Routine language Ada</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B122</td>
<td>Routine language C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B123</td>
<td>Routine language COBOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B124</td>
<td>Routine language Fortran</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B125</td>
<td>Routine language MUMPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B126</td>
<td>Routine language Pascal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B127</td>
<td>Routine language PL/I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B128</td>
<td>Routine language SQL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C011</td>
<td>Core</td>
<td>Call-Level Interface</td>
<td></td>
</tr>
<tr>
<td>E081</td>
<td>Core</td>
<td>Basic Privileges</td>
<td></td>
</tr>
<tr>
<td>E081-05</td>
<td>Core</td>
<td>UPDATE privilege at the column level</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>E081-07</td>
<td>Core</td>
<td>REFERENCES privilege at the column level</td>
<td></td>
</tr>
<tr>
<td>E081-09</td>
<td>Core</td>
<td>USAGE privilege</td>
<td></td>
</tr>
<tr>
<td>E121</td>
<td>Core</td>
<td>Basic cursor support</td>
<td></td>
</tr>
<tr>
<td>E121-06</td>
<td>Core</td>
<td>Positioned UPDATE statement</td>
<td></td>
</tr>
<tr>
<td>E121-07</td>
<td>Core</td>
<td>Positioned DELETE statement</td>
<td></td>
</tr>
<tr>
<td>E153</td>
<td>Core</td>
<td>Updatable queries with subqueries</td>
<td></td>
</tr>
<tr>
<td>E182</td>
<td>Core</td>
<td>Module language</td>
<td></td>
</tr>
<tr>
<td>F121</td>
<td></td>
<td>Basic diagnostics management</td>
<td></td>
</tr>
<tr>
<td>F121-01</td>
<td></td>
<td>GET DIAGNOSTICS statement</td>
<td></td>
</tr>
<tr>
<td>F121-02</td>
<td></td>
<td>SET TRANSACTION statement: DIAGNOSTICS SIZE clause</td>
<td></td>
</tr>
<tr>
<td>F181</td>
<td>Core</td>
<td>Multiple module support</td>
<td></td>
</tr>
<tr>
<td>F262</td>
<td></td>
<td>Extended CASE expression</td>
<td></td>
</tr>
<tr>
<td>F263</td>
<td></td>
<td>Comma-separated predicates in simple CASE expression</td>
<td></td>
</tr>
<tr>
<td>F291</td>
<td></td>
<td>UNIQUE predicate</td>
<td></td>
</tr>
<tr>
<td>F301</td>
<td></td>
<td>CORRESPONDING in query expressions</td>
<td></td>
</tr>
<tr>
<td>F311</td>
<td>Core</td>
<td>Schema definition statement</td>
<td></td>
</tr>
<tr>
<td>F311-04</td>
<td>Core</td>
<td>CREATE VIEW: WITH CHECK OPTION</td>
<td></td>
</tr>
<tr>
<td>F312</td>
<td></td>
<td>MERGE statement</td>
<td></td>
</tr>
<tr>
<td>F341</td>
<td></td>
<td>Usage tables</td>
<td></td>
</tr>
<tr>
<td>F392</td>
<td></td>
<td>Unicode escapes in identifiers</td>
<td></td>
</tr>
<tr>
<td>F393</td>
<td></td>
<td>Unicode escapes in literals</td>
<td></td>
</tr>
<tr>
<td>F402</td>
<td></td>
<td>Named column joins for LOBs, arrays, and multisets</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>F442</td>
<td></td>
<td>Mixed column references in set functions</td>
<td></td>
</tr>
<tr>
<td>F451</td>
<td></td>
<td>Character set definition</td>
<td></td>
</tr>
<tr>
<td>F461</td>
<td></td>
<td>Named character sets</td>
<td></td>
</tr>
<tr>
<td>F521</td>
<td>Enhanced integrity management</td>
<td>Assertions</td>
<td></td>
</tr>
<tr>
<td>F641</td>
<td></td>
<td>Row and table constructors</td>
<td></td>
</tr>
<tr>
<td>F661</td>
<td></td>
<td>Simple tables</td>
<td></td>
</tr>
<tr>
<td>F671</td>
<td>Enhanced integrity management</td>
<td>Subqueries in CHECK intentionally omitted</td>
<td></td>
</tr>
<tr>
<td>F691</td>
<td></td>
<td>Collation and translation</td>
<td></td>
</tr>
<tr>
<td>F692</td>
<td></td>
<td>Enhanced collation support</td>
<td></td>
</tr>
<tr>
<td>F693</td>
<td></td>
<td>SQL-session and client module collations</td>
<td></td>
</tr>
<tr>
<td>F695</td>
<td></td>
<td>Translation support</td>
<td></td>
</tr>
<tr>
<td>F696</td>
<td></td>
<td>Additional translation documentation</td>
<td></td>
</tr>
<tr>
<td>F721</td>
<td></td>
<td>Deferrable constraints foreign keys only</td>
<td></td>
</tr>
<tr>
<td>F731</td>
<td></td>
<td>INSERT column privileges</td>
<td></td>
</tr>
<tr>
<td>F741</td>
<td></td>
<td>Referential MATCH types no partial match yet</td>
<td></td>
</tr>
<tr>
<td>F751</td>
<td></td>
<td>View CHECK enhancements</td>
<td></td>
</tr>
<tr>
<td>F811</td>
<td></td>
<td>Extended flagging</td>
<td></td>
</tr>
<tr>
<td>F812</td>
<td>Core</td>
<td>Basic flagging</td>
<td></td>
</tr>
<tr>
<td>F813</td>
<td></td>
<td>Extended flagging</td>
<td></td>
</tr>
<tr>
<td>F821</td>
<td></td>
<td>Local table references</td>
<td></td>
</tr>
<tr>
<td>F831</td>
<td></td>
<td>Full cursor update</td>
<td></td>
</tr>
<tr>
<td>F831-01</td>
<td></td>
<td>Updatable scrollable cursors</td>
<td></td>
</tr>
<tr>
<td>F831-02</td>
<td></td>
<td>Updatable ordered cursors</td>
<td></td>
</tr>
<tr>
<td>S011</td>
<td>Core</td>
<td>Distinct data types</td>
<td></td>
</tr>
<tr>
<td>S011-01</td>
<td>Core</td>
<td>USER_DEFINED_TYPES view</td>
<td></td>
</tr>
<tr>
<td>S023</td>
<td>Basic object support</td>
<td>Basic structured types</td>
<td></td>
</tr>
<tr>
<td>S024</td>
<td>Enhanced object support</td>
<td>Enhanced structured types</td>
<td></td>
</tr>
<tr>
<td>S025</td>
<td></td>
<td>Final structured types</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>--------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>S026</td>
<td></td>
<td>Self-referencing structured types</td>
<td></td>
</tr>
<tr>
<td>S027</td>
<td></td>
<td>Create method by specific method name</td>
<td></td>
</tr>
<tr>
<td>S028</td>
<td></td>
<td>Permutable UDT options list</td>
<td></td>
</tr>
<tr>
<td>S041</td>
<td>Basic object support</td>
<td>Basic reference types</td>
<td></td>
</tr>
<tr>
<td>S043</td>
<td>Enhanced object support</td>
<td>Enhanced reference types</td>
<td></td>
</tr>
<tr>
<td>S051</td>
<td>Basic object support</td>
<td>Create table of type</td>
<td></td>
</tr>
<tr>
<td>S081</td>
<td>Enhanced object support</td>
<td>Subtables</td>
<td></td>
</tr>
<tr>
<td>S091</td>
<td>Basic array support</td>
<td>Basic array support</td>
<td></td>
</tr>
<tr>
<td>S091-01</td>
<td></td>
<td>Arrays of built-in data types</td>
<td></td>
</tr>
<tr>
<td>S091-02</td>
<td></td>
<td>Arrays of distinct types</td>
<td></td>
</tr>
<tr>
<td>S091-03</td>
<td></td>
<td>Array expressions</td>
<td></td>
</tr>
<tr>
<td>S092</td>
<td></td>
<td>Arrays of user-defined types</td>
<td></td>
</tr>
<tr>
<td>S094</td>
<td></td>
<td>Arrays of reference types</td>
<td></td>
</tr>
<tr>
<td>S095</td>
<td></td>
<td>Array constructors by query</td>
<td></td>
</tr>
<tr>
<td>S096</td>
<td></td>
<td>Optional array bounds</td>
<td></td>
</tr>
<tr>
<td>S097</td>
<td></td>
<td>Array element assignment</td>
<td></td>
</tr>
<tr>
<td>S151</td>
<td>Basic object support</td>
<td>Type predicate</td>
<td></td>
</tr>
<tr>
<td>S161</td>
<td>Enhanced object support</td>
<td>Subtype treatment</td>
<td></td>
</tr>
<tr>
<td>S162</td>
<td></td>
<td>Subtype treatment for references</td>
<td></td>
</tr>
<tr>
<td>S201</td>
<td></td>
<td>SQL-invoked routines on arrays</td>
<td></td>
</tr>
<tr>
<td>S201-01</td>
<td></td>
<td>Array parameters</td>
<td></td>
</tr>
<tr>
<td>S201-02</td>
<td></td>
<td>Array as result type of functions</td>
<td></td>
</tr>
<tr>
<td>S202</td>
<td></td>
<td>SQL-invoked routines on multisets</td>
<td></td>
</tr>
<tr>
<td>S231</td>
<td>Enhanced object support</td>
<td>Structured type locators</td>
<td></td>
</tr>
<tr>
<td>S232</td>
<td></td>
<td>Array locators</td>
<td></td>
</tr>
<tr>
<td>S233</td>
<td></td>
<td>Multiset locators</td>
<td></td>
</tr>
<tr>
<td>S241</td>
<td></td>
<td>Transform functions</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>S242</td>
<td></td>
<td>Alter transform statement</td>
<td></td>
</tr>
<tr>
<td>S251</td>
<td></td>
<td>User-defined orderings</td>
<td></td>
</tr>
<tr>
<td>S261</td>
<td></td>
<td>Specific type method</td>
<td></td>
</tr>
<tr>
<td>S271</td>
<td></td>
<td>Basic multiset support</td>
<td></td>
</tr>
<tr>
<td>S272</td>
<td></td>
<td>Multisets of user-defined types</td>
<td></td>
</tr>
<tr>
<td>S274</td>
<td></td>
<td>Multisets of reference types</td>
<td></td>
</tr>
<tr>
<td>S275</td>
<td></td>
<td>Advanced multiset support</td>
<td></td>
</tr>
<tr>
<td>S281</td>
<td></td>
<td>Nested collection types</td>
<td></td>
</tr>
<tr>
<td>S291</td>
<td></td>
<td>Unique constraint on entire row</td>
<td></td>
</tr>
<tr>
<td>T011</td>
<td></td>
<td>Timestamp in Information Schema</td>
<td></td>
</tr>
<tr>
<td>T041</td>
<td>Basic object support</td>
<td>Basic LOB data type support</td>
<td></td>
</tr>
<tr>
<td>T041-01</td>
<td>Basic object support</td>
<td>BLOB data type</td>
<td></td>
</tr>
<tr>
<td>T041-02</td>
<td>Basic object support</td>
<td>CLOB data type</td>
<td></td>
</tr>
<tr>
<td>T041-03</td>
<td>Basic object support</td>
<td>POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types</td>
<td></td>
</tr>
<tr>
<td>T041-04</td>
<td>Basic object support</td>
<td>Concatenation of LOB data types</td>
<td></td>
</tr>
<tr>
<td>T041-05</td>
<td>Basic object support</td>
<td>LOB locator: non-holdable</td>
<td></td>
</tr>
<tr>
<td>T042</td>
<td></td>
<td>Extended LOB data type support</td>
<td></td>
</tr>
<tr>
<td>T051</td>
<td></td>
<td>Row types</td>
<td></td>
</tr>
<tr>
<td>T052</td>
<td></td>
<td>MAX and MIN for row types</td>
<td></td>
</tr>
<tr>
<td>T053</td>
<td></td>
<td>Explicit aliases for all-fields reference</td>
<td></td>
</tr>
<tr>
<td>T061</td>
<td></td>
<td>UCS support</td>
<td></td>
</tr>
<tr>
<td>T111</td>
<td></td>
<td>Updatable joins, unions, and columns</td>
<td></td>
</tr>
<tr>
<td>T121</td>
<td></td>
<td>WITH (excluding RECURSIVE) in query expression</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>T122</td>
<td></td>
<td>WITH (excluding RECURSIVE) in subquery</td>
<td></td>
</tr>
<tr>
<td>T131</td>
<td></td>
<td>Recursive query</td>
<td></td>
</tr>
<tr>
<td>T132</td>
<td></td>
<td>Recursive query in subquery</td>
<td></td>
</tr>
<tr>
<td>T152</td>
<td></td>
<td>DISTINCT predicate with negation</td>
<td></td>
</tr>
<tr>
<td>T172</td>
<td></td>
<td>AS subquery clause in table definition</td>
<td></td>
</tr>
<tr>
<td>T173</td>
<td></td>
<td>Extended LIKE clause in table definition</td>
<td></td>
</tr>
<tr>
<td>T174</td>
<td></td>
<td>Identity columns</td>
<td></td>
</tr>
<tr>
<td>T175</td>
<td></td>
<td>Generated columns</td>
<td></td>
</tr>
<tr>
<td>T176</td>
<td></td>
<td>Sequence generator support</td>
<td></td>
</tr>
<tr>
<td>T211</td>
<td></td>
<td>Basic trigger capability</td>
<td></td>
</tr>
<tr>
<td>T211-05</td>
<td></td>
<td>Ability to specify a search condition that must be true before the trigger is invoked</td>
<td></td>
</tr>
<tr>
<td>T211-06</td>
<td></td>
<td>Support for run-time rules for the interaction of triggers and constraints</td>
<td></td>
</tr>
<tr>
<td>T211-08</td>
<td></td>
<td>Multiple triggers for the same event are executed in the order in which they were created in the catalog</td>
<td>intentionally omitted</td>
</tr>
<tr>
<td>T251</td>
<td></td>
<td>SET TRANSACTION statement: LOCAL option</td>
<td></td>
</tr>
<tr>
<td>T261</td>
<td></td>
<td>Chained transactions</td>
<td></td>
</tr>
<tr>
<td>T272</td>
<td></td>
<td>Enhanced savepoint management</td>
<td></td>
</tr>
<tr>
<td>T281</td>
<td></td>
<td>SELECT privilege with column granularity</td>
<td></td>
</tr>
<tr>
<td>T301</td>
<td></td>
<td>Functional dependencies</td>
<td></td>
</tr>
<tr>
<td>T321</td>
<td>Core</td>
<td>Basic SQL-invoked routines</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>T321-02</td>
<td>Core</td>
<td>User-defined stored procedures with no overloading</td>
<td></td>
</tr>
<tr>
<td>T321-04</td>
<td>Core</td>
<td>CALL statement</td>
<td></td>
</tr>
<tr>
<td>T321-05</td>
<td>Core</td>
<td>RETURN statement</td>
<td></td>
</tr>
<tr>
<td>T324</td>
<td></td>
<td>Explicit security for SQL routines</td>
<td></td>
</tr>
<tr>
<td>T325</td>
<td></td>
<td>Qualified SQL parameter references</td>
<td></td>
</tr>
<tr>
<td>T326</td>
<td></td>
<td>Table functions</td>
<td></td>
</tr>
<tr>
<td>T331</td>
<td></td>
<td>Basic roles</td>
<td></td>
</tr>
<tr>
<td>T332</td>
<td></td>
<td>Extended roles</td>
<td></td>
</tr>
<tr>
<td>T401</td>
<td></td>
<td>INSERT into a cursor</td>
<td></td>
</tr>
<tr>
<td>T411</td>
<td></td>
<td>UPDATE statement: SET ROW option</td>
<td></td>
</tr>
<tr>
<td>T431</td>
<td>OLAP</td>
<td>Extended grouping capabilities</td>
<td></td>
</tr>
<tr>
<td>T432</td>
<td></td>
<td>Nested and concatenated GROUPING SETS</td>
<td></td>
</tr>
<tr>
<td>T433</td>
<td></td>
<td>Multiargument GROUPING function</td>
<td></td>
</tr>
<tr>
<td>T434</td>
<td></td>
<td>GROUP BY DISTINCT</td>
<td></td>
</tr>
<tr>
<td>T471</td>
<td></td>
<td>Result sets return value</td>
<td></td>
</tr>
<tr>
<td>T491</td>
<td></td>
<td>LATERAL derived table</td>
<td></td>
</tr>
<tr>
<td>T511</td>
<td></td>
<td>Transaction counts</td>
<td></td>
</tr>
<tr>
<td>T541</td>
<td></td>
<td>Updatable table references</td>
<td></td>
</tr>
<tr>
<td>T561</td>
<td></td>
<td>Holdable locators</td>
<td></td>
</tr>
<tr>
<td>T571</td>
<td></td>
<td>Array-returning external SQL-invoked functions</td>
<td></td>
</tr>
<tr>
<td>T572</td>
<td></td>
<td>Multiset-returning external SQL-invoked functions</td>
<td></td>
</tr>
<tr>
<td>T601</td>
<td></td>
<td>Local cursor references</td>
<td></td>
</tr>
<tr>
<td>T611</td>
<td>OLAP</td>
<td>Elementary OLAP operations</td>
<td></td>
</tr>
<tr>
<td>T612</td>
<td></td>
<td>Advanced OLAP operations</td>
<td></td>
</tr>
<tr>
<td>T613</td>
<td></td>
<td>Sampling</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>T621</td>
<td></td>
<td>Enhanced numeric functions</td>
<td></td>
</tr>
<tr>
<td>T631</td>
<td>Core</td>
<td>IN predicate with one list element</td>
<td></td>
</tr>
<tr>
<td>T641</td>
<td></td>
<td>Multiple column assignment</td>
<td></td>
</tr>
<tr>
<td>T651</td>
<td></td>
<td>SQL-schema statements in SQL routines</td>
<td></td>
</tr>
<tr>
<td>T652</td>
<td></td>
<td>SQL-dynamic statements in SQL routines</td>
<td></td>
</tr>
<tr>
<td>T653</td>
<td></td>
<td>SQL-schema statements in external routines</td>
<td></td>
</tr>
<tr>
<td>T654</td>
<td></td>
<td>SQL-dynamic statements in external routines</td>
<td></td>
</tr>
<tr>
<td>T655</td>
<td></td>
<td>Cyclically dependent routines</td>
<td></td>
</tr>
<tr>
<td>X012</td>
<td></td>
<td>Multisets of XML type</td>
<td></td>
</tr>
<tr>
<td>X013</td>
<td></td>
<td>Distinct types of XML type</td>
<td></td>
</tr>
<tr>
<td>X014</td>
<td></td>
<td>Attributes of XML type</td>
<td></td>
</tr>
<tr>
<td>X015</td>
<td></td>
<td>Fields of XML type</td>
<td></td>
</tr>
<tr>
<td>X035</td>
<td></td>
<td>XMLAgg: ORDER BY option</td>
<td></td>
</tr>
<tr>
<td>X080</td>
<td></td>
<td>Namespaces in XML publishing</td>
<td></td>
</tr>
<tr>
<td>X081</td>
<td></td>
<td>Query-level XML namespace declarations</td>
<td></td>
</tr>
<tr>
<td>X082</td>
<td></td>
<td>XML namespace declarations in DML</td>
<td></td>
</tr>
<tr>
<td>X083</td>
<td></td>
<td>XML namespace declarations in DDL</td>
<td></td>
</tr>
<tr>
<td>X084</td>
<td></td>
<td>XML namespace declarations in compound statements</td>
<td></td>
</tr>
<tr>
<td>X100</td>
<td></td>
<td>Host language support for XML: CONTENT option</td>
<td></td>
</tr>
<tr>
<td>X101</td>
<td></td>
<td>Host language support for XML: DOCUMENT option</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Package</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>X110</td>
<td></td>
<td>Host language support for XML: VARCHAR mapping</td>
<td></td>
</tr>
<tr>
<td>X111</td>
<td></td>
<td>Host language support for XML: CLOB mapping</td>
<td></td>
</tr>
<tr>
<td>X131</td>
<td></td>
<td>Query-level XMLBINARY clause</td>
<td></td>
</tr>
<tr>
<td>X132</td>
<td></td>
<td>XMLBINARY clause in DML</td>
<td></td>
</tr>
<tr>
<td>X133</td>
<td></td>
<td>XMLBINARY clause in DDL</td>
<td></td>
</tr>
<tr>
<td>X134</td>
<td></td>
<td>XMLBINARY clause in compound statements</td>
<td></td>
</tr>
<tr>
<td>X135</td>
<td></td>
<td>XMLBINARY clause in subqueries</td>
<td></td>
</tr>
</tbody>
</table>
Appendix E. Release Notes

The release notes contain the significant changes in each PostgreSQL release, with major features and migration issues listed at the top. The release notes do not contain changes that affect only a few users or changes that are internal and therefore not user-visible. For example, the optimizer is improved in almost every release, but the improvements are usually observed by users as simply faster queries.

A complete list of changes for each release can be obtained by viewing the Git logs for each release. The pgsql-committers email list\(^1\) records all source code changes as well. There is also a web interface\(^2\) that shows changes to specific files.

The name appearing next to each item represents the major developer for that item. Of course all changes involve community discussion and patch review, so each item is truly a community effort.

E.1. Release 8.3.23

Release Date: 2013-02-07

This release contains a variety of fixes from 8.3.22. For information about new features in the 8.3 major release, see Section E.24.

This is expected to be the last PostgreSQL release in the 8.3.X series. Users are encouraged to update to a newer release branch soon.

E.1.1. Migration to Version 8.3.23

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see the release notes for 8.3.17.

E.1.2. Changes

- Prevent execution of `enum_recv` from SQL (Tom Lane)
  The function was misdeclared, allowing a simple SQL command to crash the server. In principle an attacker might be able to use it to examine the contents of server memory. Our thanks to Sumit Soni (via Secunia SVCRP) for reporting this issue. (CVE-2013-0255)

- Fix SQL grammar to allow subscripting or field selection from a sub-SELECT result (Tom Lane)

- Protect against race conditions when scanning `pg_tablespace` (Stephen Frost, Tom Lane)

  `CREATE DATABASE` and `DROP DATABASE` could misbehave if there were concurrent updates of `pg_tablespace` entries.

- Prevent `DROP OWNED` from trying to drop whole databases or tablespaces (Álvaro Herrera)

\(^1\) [http://archives.postgresql.org/pgsql-committers/](http://archives.postgresql.org/pgsql-committers/)

\(^2\) [http://git.postgresql.org/gitweb/?p=postgresql.git;a=summary](http://git.postgresql.org/gitweb/?p=postgresql.git;a=summary)
Appendix E. Release Notes

For safety, ownership of these objects must be reassigned, not dropped.

- Prevent misbehavior when a RowExpr or XmlExpr is parse-analyzed twice (Andres Freund, Tom Lane)
  This mistake could be user-visible in contexts such as CREATE TABLE LIKE INCLUDING INDEXES.
- Improve defenses against integer overflow in hashtable sizing calculations (Jeff Davis)
- Ensure that non-ASCII prompt strings are translated to the correct code page on Windows (Alexander Law, Noah Misch)
  This bug affected psql and some other client programs.
- Fix possible crash in psql’s `?` command when not connected to a database (Meng Qingzhong)
- Fix one-byte buffer overrun in libpq’s PQprintTuples (Xi Wang)
  This ancient function is not used anywhere by PostgreSQL itself, but it might still be used by some client code.
- Rearrange configure’s tests for supplied functions so it is not fooled by bogus exports from libedit/libreadline (Christoph Berg)
- Ensure Windows build number increases over time (Magnus Hagander)
- Make pgxs build executables with the right .exe suffix when cross-compiling for Windows (Zoltan Boszormenyi)
- Add new timezone abbreviation FET (Tom Lane)
  This is now used in some eastern-European time zones.

E.2. Release 8.3.22

Release Date: 2012-12-06

This release contains a variety of fixes from 8.3.21. For information about new features in the 8.3 major release, see Section E.24.

The PostgreSQL community will stop releasing updates for the 8.3.X release series in February 2013. Users are encouraged to update to a newer release branch soon.

E.2.1. Migration to Version 8.3.22

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see the release notes for 8.3.17.

E.2.2. Changes

- Fix multiple bugs associated with CREATE INDEX CONCURRENTLY (Andres Freund, Tom Lane)
Fix `CREATE INDEX CONCURRENTLY` to use in-place updates when changing the state of an index’s `pg_index` row. This prevents race conditions that could cause concurrent sessions to miss updating the target index, thus resulting in corrupt concurrently-created indexes.

Also, fix various other operations to ensure that they ignore invalid indexes resulting from a failed `CREATE INDEX CONCURRENTLY` command. The most important of these is `VACUUM`, because an auto-vacuum could easily be launched on the table before corrective action can be taken to fix or remove the invalid index.

- Avoid corruption of internal hash tables when out of memory (Hitoshi Harada)
- Fix planning of non-strict equivalence clauses above outer joins (Tom Lane)
  
  The planner could derive incorrect constraints from a clause equating a non-strict construct to something else, for example `WHERE COALESCE(foo, 0) = 0` when `foo` is coming from the nullable side of an outer join.
- Improve planner’s ability to prove exclusion constraints from equivalence classes (Tom Lane)
- Fix partial-row matching in hashed subplans to handle cross-type cases correctly (Tom Lane)
  
  This affects multicolumn `NOT IN` subplans, such as `WHERE (a, b) NOT IN (SELECT x, y FROM ...)` when for instance `b` and `y` are `int4` and `int8` respectively. This mistake led to wrong answers or crashes depending on the specific datatypes involved.
- Acquire buffer lock when re-fetching the old tuple for an `AFTER ROW UPDATE/DELETE` trigger (Andres Freund)
  
  In very unusual circumstances, this oversight could result in passing incorrect data to the precheck logic for a foreign-key enforcement trigger. That could result in a crash, or in an incorrect decision about whether to fire the trigger.
- Fix `REASSIGN OWNED` to handle grants on tablespaces (Álvaro Herrera)
- Ignore incorrect `pg_attribute` entries for system columns for views (Tom Lane)
  
  Views do not have any system columns. However, we forgot to remove such entries when converting a table to a view. That’s fixed properly for 9.3 and later, but in previous branches we need to defend against existing mis-converted views.
- Fix rule printing to dump `INSERT INTO` table `DEFAULT VALUES` correctly (Tom Lane)
- Guard against stack overflow when there are too many `UNION/INTERSECT/EXCEPT` clauses in a query (Tom Lane)
- Prevent platform-dependent failures when dividing the minimum possible integer value by -1 (Xi Wang, Tom Lane)
- Fix possible access past end of string in date parsing (Hitoshi Harada)
- Produce an understandable error message if the length of the path name for a Unix-domain socket exceeds the platform-specific limit (Tom Lane, Andrew Dunstan)
  
  Formerly, this would result in something quite unhelpful, such as “Non-recoverable failure in name resolution”.
- Fix memory leaks when sending composite column values to the client (Tom Lane)
- Make `pg_ctl` more robust about reading the `postmaster.pid` file (Heikki Linnakangas)
  
  Fix race conditions and possible file descriptor leakage.
- Fix possible crash in `psql` if incorrectly-encoded data is presented and the `client_encoding` setting is a client-only encoding, such as SJIS (Jiang Guiqing)
- Fix bugs in the `restore.sql` script emitted by `pg_dump` in `tar` output format (Tom Lane)
The script would fail outright on tables whose names include upper-case characters. Also, make the script capable of restoring data in --inserts mode as well as the regular COPY mode.

- Fix pg_restore to accept POSIX-conformant tar files (Brian Weaver, Tom Lane)

The original coding of pg_dump’s tar output mode produced files that are not fully conformant with the POSIX standard. This has been corrected for version 9.3. This patch updates previous branches so that they will accept both the incorrect and the corrected formats, in hopes of avoiding compatibility problems when 9.3 comes out.

- Fix pg_resetxlog to locate postmaster.pid correctly when given a relative path to the data directory (Tom Lane)

This mistake could lead to pg_resetxlog not noticing that there is an active postmaster using the data directory.

- Fix libpq’s lo_import() and lo_export() functions to report file I/O errors properly (Tom Lane)

- Fix ecpg’s processing of nested structure pointer variables (Muhammad Usama)

- Make contrib/pageinspect’s btree page inspection functions take buffer locks while examining pages (Tom Lane)

- Fix pgxs support for building loadable modules on AIX (Tom Lane)

Building modules outside the original source tree didn’t work on AIX.

- Update time zone data files to tzdata release 2012j for DST law changes in Cuba, Israel, Jordan, Libya, Palestine, Western Samoa, and portions of Brazil.

E.3. Release 8.3.21

Release Date: 2012-09-24

This release contains a variety of fixes from 8.3.20. For information about new features in the 8.3 major release, see Section E.24.

The PostgreSQL community will stop releasing updates for the 8.3.X release series in February 2013. Users are encouraged to update to a newer release branch soon.

E.3.1. Migration to Version 8.3.21

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see the release notes for 8.3.17.

E.3.2. Changes

- Improve page-splitting decisions in GiST indexes (Alexander Korotkov, Robert Haas, Tom Lane)

  Multi-column GiST indexes might suffer unexpected bloat due to this error.
• Fix cascading privilege revoke to stop if privileges are still held (Tom Lane)
  If we revoke a grant option from some role $x$, but $x$ still holds that option via a grant from someone else, we should not recursively revoke the corresponding privilege from role(s) $y$ that $x$ had granted it to.
• Fix handling of SIGFPE when PL/Perl is in use (Andres Freund)
  Perl resets the process’s SIGFPE handler to SIG_IGN, which could result in crashes later on. Restore the normal Postgres signal handler after initializing PL/Perl.
• Prevent PL/Perl from crashing if a recursive PL/Perl function is redefined while being executed (Tom Lane)
• Work around possible misoptimization in PL/Perl (Tom Lane)
  Some Linux distributions contain an incorrect version of pthread.h that results in incorrect compiled code in PL/Perl, leading to crashes if a PL/Perl function calls another one that throws an error.
• Update time zone data files to tzdata release 2012f for DST law changes in Fiji

E.4. Release 8.3.20

Release Date: 2012-08-17

This release contains a variety of fixes from 8.3.19. For information about new features in the 8.3 major release, see Section E.24.

The PostgreSQL community will stop releasing updates for the 8.3.X release series in February 2013. Users are encouraged to update to a newer release branch soon.

E.4.1. Migration to Version 8.3.20

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see the release notes for 8.3.17.

E.4.2. Changes

• Prevent access to external files/URLs via XML entity references (Noah Misch, Tom Lane)
  xml_parse() would attempt to fetch external files or URLs as needed to resolve DTD and entity references in an XML value, thus allowing unprivileged database users to attempt to fetch data with the privileges of the database server. While the external data wouldn’t get returned directly to the user, portions of it could be exposed in error messages if the data didn’t parse as valid XML; and in any case the mere ability to check existence of a file might be useful to an attacker. (CVE-2012-3489)
• Prevent access to external files/URLs via contrib/xml2\'s xslt_process() (Peter Eisentraut)
libxslt offers the ability to read and write both files and URLs through stylesheet commands, thus allowing unprivileged database users to both read and write data with the privileges of the database server. Disable that through proper use of libxslt’s security options. (CVE-2012-3488)

Also, remove xslt_process()’s ability to fetch documents and stylesheets from external files/URLs. While this was a documented “feature”, it was long regarded as a bad idea. The fix for CVE-2012-3489 broke that capability, and rather than expend effort on trying to fix it, we’re just going to summarily remove it.

- Prevent too-early recycling of btree index pages (Noah Misch)
  When we allowed read-only transactions to skip assigning XIDs, we introduced the possibility that a deleted btree page could be recycled while a read-only transaction was still in flight to it. This would result in incorrect index search results. The probability of such an error occurring in the field seems very low because of the timing requirements, but nonetheless it should be fixed.

- Fix crash-safety bug with newly-created-or-reset sequences (Tom Lane)
  If ALTER SEQUENCE was executed on a freshly created or reset sequence, and then precisely one nextval() call was made on it, and then the server crashed, WAL replay would restore the sequence to a state in which it appeared that no nextval() had been done, thus allowing the first sequence value to be returned again by the next nextval() call. In particular this could manifest for serial columns, since creation of a serial column’s sequence includes an ALTER SEQUENCE OWNED BY step.

- Ensure the backup_label file is fsync’ed after pg_start_backup() (Dave Kerr)

- Back-patch 9.1 improvement to compress the fsync request queue (Robert Haas)
  This improves performance during checkpoints. The 9.1 change has now seen enough field testing to seem safe to back-patch.

- Only allow autovacuum to be auto-canceled by a directly blocked process (Tom Lane)
  The original coding could allow inconsistent behavior in some cases; in particular, an autovacuum could get canceled after less than deadlock_timeout grace period.

- Improve logging of autovacuum cancels (Robert Haas)

- Fix log collector so that log_truncate_on_rotation works during the very first log rotation after server start (Tom Lane)

- Ensure that a whole-row reference to a subquery doesn’t include any extra GROUP BY or ORDER BY columns (Tom Lane)

- Disallow copying whole-row references in CHECK constraints and index definitions during CREATE TABLE (Tom Lane)
  This situation can arise in CREATE TABLE with LIKE or INHERITS. The copied whole-row variable was incorrectly labeled with the row type of the original table not the new one. Rejecting the case seems reasonable for LIKE, since the row types might well diverge later. For INHERITS we should ideally allow it, with an implicit coercion to the parent table’s row type; but that will require more work than seems safe to back-patch.

- Fix memory leak in ARRAY(SELECT ...) subqueries (Heikki Linnakangas, Tom Lane)

- Fix extraction of common prefixes from regular expressions (Tom Lane)
  The code could get confused by quantified parenthesized subexpressions, such as ^\(foo\)?bar. This would lead to incorrect index optimization of searches for such patterns.

- Report errors properly in contrib/xml2’s xslt_process() (Tom Lane)

- Update time zone data files to tzdata release 2012e for DST law changes in Morocco and Tokelau
Appendix E. Release Notes

E.5. Release 8.3.19

Release Date: 2012-06-04

This release contains a variety of fixes from 8.3.18. For information about new features in the 8.3 major release, see Section E.24.

E.5.1. Migration to Version 8.3.19

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see the release notes for 8.3.17.

E.5.2. Changes

• Fix incorrect password transformation in contrib/pgcrypto’s DES crypt() function (Solar Designer)

  If a password string contained the byte value 0x80, the remainder of the password was ignored, causing the password to be much weaker than it appeared. With this fix, the rest of the string is properly included in the DES hash. Any stored password values that are affected by this bug will thus no longer match, so the stored values may need to be updated. (CVE-2012-2143)

• Ignore SECURITY DEFINER and SET attributes for a procedural language’s call handler (Tom Lane)

  Applying such attributes to a call handler could crash the server. (CVE-2012-2655)

• Allow numeric timezone offsets in timestamp input to be up to 16 hours away from UTC (Tom Lane)

  Some historical time zones have offsets larger than 15 hours, the previous limit. This could result in dumped data values being rejected during reload.

• Fix timestamp conversion to cope when the given time is exactly the last DST transition time for the current timezone (Tom Lane)

  This oversight has been there a long time, but was not noticed previously because most DST-using zones are presumed to have an indefinite sequence of future DST transitions.

• Fix text to name and char to name casts to perform string truncation correctly in multibyte encodings (Karl Schnaitter)

• Fix memory copying bug in to_tsquery() (Heikki Linnakangas)

• Fix slow session startup when pg_attribute is very large (Tom Lane)

  If pg_attribute exceeds one-fourth of shared_buffers, cache rebuilding code that is sometimes needed during session start would trigger the synchronized-scan logic, causing it to take many times longer than normal. The problem was particularly acute if many new sessions were starting at once.

• Ensure sequential scans check for query cancel reasonably often (Merlin Moncure)

  A scan encountering many consecutive pages that contain no live tuples would not respond to interrupts meanwhile.
• Ensure the Windows implementation of PGSemaphoreLock() clears ImmediateInterruptOK before returning (Tom Lane)

This oversight meant that a query-cancel interrupt received later in the same query could be accepted at an unsafe time, with unpredictable but not good consequences.

• Show whole-row variables safely when printing views or rules (Abbas Butt, Tom Lane)

Corner cases involving ambiguous names (that is, the name could be either a table or column name of the query) were printed in an ambiguous way, risking that the view or rule would be interpreted differently after dump and reload. Avoid the ambiguous case by attaching a no-op cast.

• Ensure autovacuum worker processes perform stack depth checking properly (Heikki Linnakangas)

Previously, infinite recursion in a function invoked by auto-Analyze could crash worker processes.

• Fix logging collector to not lose log coherency under high load (Andrew Dunstan)

The collector previously could fail to reassemble large messages if it got too busy.

• Fix logging collector to ensure it will restart file rotation after receiving SIGHUP (Tom Lane)

• Fix PL/pgSQL’s GET DIAGNOSTICS command when the target is the function’s first variable (Tom Lane)

• Fix several performance problems in pg_dump when the database contains many objects (Jeff Janes, Tom Lane)

pg_dump could get very slow if the database contained many schemas, or if many objects are in dependency loops, or if there are many owned sequences.

• Fix contrib/dblink's dblink_exec() to not leak temporary database connections upon error (Tom Lane)

• Update time zone data files to tzdata release 2012c for DST law changes in Antarctica, Armenia, Chile, Cuba, Falkland Islands, Gaza, Haiti, Hebron, Morocco, Syria, and Tokelau Islands; also historical corrections for Canada.

E.6. Release 8.3.18

Release Date: 2012-02-27

This release contains a variety of fixes from 8.3.17. For information about new features in the 8.3 major release, see Section E.24.

E.6.1. Migration to Version 8.3.18

A dump/restore is not required for those running 8.3.X.

However, if you are upgrading from a version earlier than 8.3.17, see the release notes for 8.3.17.
E.6.2. Changes

- Require execute permission on the trigger function for `CREATE TRIGGER` (Robert Haas)
  This missing check could allow another user to execute a trigger function with forged input data, by installing it on a table he owns. This is only of significance for trigger functions marked `SECURITY DEFINER`, since otherwise trigger functions run as the table owner anyway. (CVE-2012-0866)

- Convert newlines to spaces in names written in `pg_dump` comments (Robert Haas)
  `pg_dump` was incautious about sanitizing object names that are emitted within SQL comments in its output script. A name containing a newline would at least render the script syntactically incorrect. Maliciously crafted object names could present a SQL injection risk when the script is reloaded. (CVE-2012-0868)

- Fix btree index corruption from insertions concurrent with vacuuming (Tom Lane)
  An index page split caused by an insertion could sometimes cause a concurrently-running `VACUUM` to miss removing index entries that it should remove. After the corresponding table rows are removed, the dangling index entries would cause errors (such as “could not read block N in file ...”) or worse, silently wrong query results after unrelated rows are re-inserted at the now-free table locations. This bug has been present since release 8.2, but occurs so infrequently that it was not diagnosed until now. If you have reason to suspect that it has happened in your database, reindexing the affected index will fix things.

- Allow non-existent values for some settings in `ALTER USER/DATABASE SET` (Heikki Linnakangas)
  Allow `default_text_search_config`, `default_tablespace`, and `temp_tablespaces` to be set to names that are not known. This is because they might be known in another database where the setting is intended to be used, or for the tablespace cases because the tablespace might not be created yet. The same issue was previously recognized for `search_path`, and these settings now act like that one.

- Track the OID counter correctly during WAL replay, even when it wraps around (Tom Lane)
  Previously the OID counter would remain stuck at a high value until the system exited replay mode. The practical consequences of that are usually nil, but there are scenarios wherein a standby server that’s been promoted to master might take a long time to advance the OID counter to a reasonable value once values are needed.

- Fix regular expression back-references with `*` attached (Tom Lane)
  Rather than enforcing an exact string match, the code would effectively accept any string that satisfies the pattern sub-expression referenced by the back-reference symbol.
  A similar problem still affects back-references that are embedded in a larger quantified expression, rather than being the immediate subject of the quantifier. This will be addressed in a future PostgreSQL release.

- Fix recently-introduced memory leak in processing of `inet/cidr` values (Heikki Linnakangas)
  A patch in the December 2011 releases of PostgreSQL caused memory leakage in these operations, which could be significant in scenarios such as building a btree index on such a column.

- Avoid double close of file handle in `syslogger` on Windows (MauMau)
  Ordinarily this error was invisible, but it would cause an exception when running on a debug version of Windows.

- Fix I/O-conversion-related memory leaks in `plpgsql` (Andres Freund, Jan Urbanski, Tom Lane)
Appendix E. Release Notes

Certain operations would leak memory until the end of the current function.

- Improve pg_dump’s handling of inherited table columns (Tom Lane)
  
  pg_dump mishandled situations where a child column has a different default expression than its
  parent column. If the default is textually identical to the parent’s default, but not actually the same
  (for instance, because of schema search path differences) it would not be recognized as different, so
  that after dump and restore the child would be allowed to inherit the parent’s default. Child columns
  that are NOT NULL where their parent is not could also be restored subtly incorrectly.

- Fix pg_restore’s direct-to-database mode for INSERT-style table data (Tom Lane)
  
  Direct-to-database restores from archive files made with --inserts or --column-inserts options fail
  when using pg_restore from a release dated September or December 2011, as a result of an
  oversight in a fix for another problem. The archive file itself is not at fault, and text-mode output
  is okay.

- Fix error in contrib/intarray’s int[] & int[] operator (Guillaume Lelarge)
  
  If the smallest integer the two input arrays have in common is 1, and there are smaller values in
  either array, then 1 would be incorrectly omitted from the result.

- Fix error detection in contrib/pgcrypto’s encrypt_iv() and decrypt_iv() (Marko Kreen)
  
  These functions failed to report certain types of invalid-input errors, and would instead return
  random garbage values for incorrect input.

- Fix one-byte buffer overrun in contrib/test_parser (Paul Guyot)
  
  The code would try to read one more byte than it should, which would crash in corner cases. Since
  contrib/test_parser is only example code, this is not a security issue in itself, but bad example
  code is still bad.

- Use __sync_lock_test_and_set() for spinlocks on ARM, if available (Martin Pitt)
  
  This function replaces our previous use of the SWPB instruction, which is deprecated and not avail-
  able on ARMv6 and later. Reports suggest that the old code doesn’t fail in an obvious way on
  recent ARM boards, but simply doesn’t interlock concurrent accesses, leading to bizarre failures in
  multiprocess operation.

- Use -fexcess-precision=standard option when building with gcc versions that accept it (An-
  drew Dunstan)
  
  This prevents assorted scenarios wherein recent versions of gcc will produce creative results.

- Allow use of threaded Python on FreeBSD (Chris Rees)
  
  Our configure script previously believed that this combination wouldn’t work; but FreeBSD fixed
  the problem, so remove that error check.

E.7. Release 8.3.17

Release Date: 2011-12-05

This release contains a variety of fixes from 8.3.16. For information about new features in the 8.3
major release, see Section E.24.
E.7.1. Migration to Version 8.3.17

A dump/restore is not required for those running 8.3.X.

However, a longstanding error was discovered in the definition of the `information_schema.referential_constraints` view. If you rely on correct results from that view, you should replace its definition as explained in the first changelog item below.

Also, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.

E.7.2. Changes

- **Fix bugs in `information_schema.referential_constraints` view (Tom Lane)**
  
  This view was being insufficiently careful about matching the foreign-key constraint to the depended-on primary or unique key constraint. That could result in failure to show a foreign key constraint at all, or showing it multiple times, or claiming that it depends on a different constraint than the one it really does.

  Since the view definition is installed by initdb, merely upgrading will not fix the problem. If you need to fix this in an existing installation, you can (as a superuser) drop the `information_schema` schema then re-create it by sourcing `SHAREDIR/information_schema.sql` (Run `pg_config --sharedir` if you’re uncertain where `SHAREDIR` is). This must be repeated in each database to be fixed.

- **Fix TOAST-related data corruption during `CREATE TABLE dest AS SELECT * FROM src` or `INSERT INTO dest SELECT * FROM src` (Tom Lane)**

  If a table has been modified by `ALTER TABLE ADD COLUMN`, attempts to copy its data verbatim to another table could produce corrupt results in certain corner cases. The problem can only manifest in this precise form in 8.4 and later, but we patched earlier versions as well in case there are other code paths that could trigger the same bug.

- **Fix race condition during toast table access from stale syscache entries (Tom Lane)**

  The typical symptom was transient errors like “missing chunk number 0 for toast value NNNNN in pg_toast_2619”, where the cited toast table would always belong to a system catalog.

- **Make `DatumGetInetP()` unpack inet datums that have a 1-byte header, and add a new macro, `DatumGetInetPP()`., that does not (Heikki Linnakangas)**

  This change affects no core code, but might prevent crashes in add-on code that expects `DatumGetInetP()` to produce an unpacked datum as per usual convention.

- **Improve locale support in `money` type’s input and output (Tom Lane)**

  Aside from not supporting all standard `lc_monetary` formatting options, the input and output functions were inconsistent, meaning there were locales in which dumped `money` values could not be re-read.

- **Don’t let `transform_null_equals` affect `CASE foo WHEN NULL ...` constructs (Heikki Linnakangas)**

  `transform_null_equals` is only supposed to affect `foo = NULL` expressions written directly by the user, not equality checks generated internally by this form of `CASE`.

- **Change foreign-key trigger creation order to better support self-referential foreign keys (Tom Lane)**

  For a cascading foreign key that references its own table, a row update will fire both the `ON UPDATE` trigger and the `CHECK` trigger as one event. The `ON UPDATE` trigger must execute first,
else the CHECK will check a non-final state of the row and possibly throw an inappropriate error. However, the firing order of these triggers is determined by their names, which generally sort in creation order since the triggers have auto-generated names following the convention "RI_ConstraintTrigger_NNNN". A proper fix would require modifying that convention, which we will do in 9.2, but it seems risky to change it in existing releases. So this patch just changes the creation order of the triggers. Users encountering this type of error should drop and re-create the foreign key constraint to get its triggers into the right order.

• Avoid floating-point underflow while tracking buffer allocation rate (Greg Matthews)
  While harmless in itself, on certain platforms this would result in annoying kernel log messages.
• Preserve blank lines within commands in psql's command history (Robert Haas)
  The former behavior could cause problems if an empty line was removed from within a string literal, for example.
• Fix pg_dump to dump user-defined casts between auto-generated types, such as table rowtypes (Tom Lane)
• Use the preferred version of xsubpp to build PL/Perl, not necessarily the operating system's main copy (David Wheeler and Alex Hunsaker)
• Fix incorrect coding in contrib/dict_int and contrib/dict_xsyn (Tom Lane)
  Some functions incorrectly assumed that memory returned by palloc() is guaranteed zeroed.
• Honor query cancel interrupts promptly in pgstatindex() (Robert Haas)
• Ensure VPATH builds properly install all server header files (Peter Eisentraut)
• Shorten file names reported in verbose error messages (Peter Eisentraut)
  Regular builds have always reported just the name of the C file containing the error message call, but VPATH builds formerly reported an absolute path name.
• Fix interpretation of Windows timezone names for Central America (Tom Lane)
  Map “Central America Standard Time” to CST6, not CST6CDT, because DST is generally not observed anywhere in Central America.
• Update time zone data files to tzdata release 2011n for DST law changes in Brazil, Cuba, Fiji, Palestine, Russia, and Samoa; also historical corrections for Alaska and British East Africa.

E.8. Release 8.3.16

Release Date: 2011-09-26

This release contains a variety of fixes from 8.3.15. For information about new features in the 8.3 major release, see Section E.24.

E.8.1. Migration to Version 8.3.16

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.
E.8.2. Changes

- Fix bugs in indexing of in-doubt HOT-updated tuples (Tom Lane)
  These bugs could result in index corruption after reindexing a system catalog. They are not believed to affect user indexes.
- Fix multiple bugs in GiST index page split processing (Heikki Linnakangas)
  The probability of occurrence was low, but these could lead to index corruption.
- Fix possible buffer overrun in \texttt{tsvector\_concat()} (Tom Lane)
  The function could underestimate the amount of memory needed for its result, leading to server crashes.
- Fix crash in \texttt{xml\_recv} when processing a “standalone” parameter (Tom Lane)
- Avoid possibly accessing off the end of memory in \texttt{ANALYZE} and in SJIS-2004 encoding conversion (Noah Misch)
  This fixes some very-low-probability server crash scenarios.
- Fix race condition in relcache init file invalidation (Tom Lane)
  There was a window wherein a new backend process could read a stale init file but miss the inval messages that would tell it the data is stale. The result would be bizarre failures in catalog accesses, typically “could not read block 0 in file ...” later during startup.
- Fix memory leak at end of a GiST index scan (Tom Lane)
  Commands that perform many separate GiST index scans, such as verification of a new GiST-based exclusion constraint on a table already containing many rows, could transiently require large amounts of memory due to this leak.
- Fix performance problem when constructing a large, lossy bitmap (Tom Lane)
- Fix array- and path-creating functions to ensure padding bytes are zeroes (Tom Lane)
  This avoids some situations where the planner will think that semantically-equal constants are not equal, resulting in poor optimization.
- Work around gcc 4.6.0 bug that breaks WAL replay (Tom Lane)
  This could lead to loss of committed transactions after a server crash.
- Fix dump bug for \texttt{VALUES} in a view (Tom Lane)
- \textbf{Disallow} \texttt{SELECT FOR UPDATE/SHARE} on sequences (Tom Lane)
  This operation doesn’t work as expected and can lead to failures.
- Defend against integer overflow when computing size of a hash table (Tom Lane)
- Fix cases where \texttt{CLUSTER} might attempt to access already-removed TOAST data (Tom Lane)
- Fix portability bugs in use of credentials control messages for “peer” authentication (Tom Lane)
- Fix SSPI login when multiple roundtrips are required (Ahmed Shinwari, Magnus Hagander)
  The typical symptom of this problem was “The function requested is not supported” errors during SSPI login.
- Fix typo in \texttt{pg\_srand48} seed initialization (Andres Freund)
This led to failure to use all bits of the provided seed. This function is not used on most platforms (only those without `srandom`), and the potential security exposure from a less-random-than-expected seed seems minimal in any case.

- Avoid integer overflow when the sum of `LIMIT` and `OFFSET` values exceeds $2^{63}$ (Heikki Linnakangas)
- Add overflow checks to `int4` and `int8` versions of `generate_series()` (Robert Haas)
- Fix trailing-zero removal in `to_char()` (Marti Raudsepp)
  In a format with `FM` and no digit positions after the decimal point, zeroes to the left of the decimal point could be removed incorrectly.
- Fix `pg_size_pretty()` to avoid overflow for inputs close to $2^{63}$ (Tom Lane)
- In `pg_ctl`, support silent mode for service registrations on Windows (MauMau)
- Fix sql’s counting of script file line numbers during `COPY` from a different file (Tom Lane)
- Fix `pg_restore`’s direct-to-database mode for `standard_conforming_strings` (Tom Lane)
  `pg_restore` could emit incorrect commands when restoring directly to a database server from an archive file that had been made with `standard_conforming_strings` set to on.
- Fix write-past-buffer-end and memory leak in libpq’s LDAP service lookup code (Albe Laurenz)
- In libpq, avoid failures when using nonblocking I/O and an SSL connection (Martin Pihlak, Tom Lane)
- Improve libpq’s handling of failures during connection startup (Tom Lane)
  In particular, the response to a server report of `fork()` failure during SSL connection startup is now saner.
- Improve libpq’s error reporting for SSL failures (Tom Lane)
- Make ecpglib write `double` values with 15 digits precision (Akira Kurosawa)
- In ecpglib, be sure `LC_NUMERIC` setting is restored after an error (Michael Meskes)
- Apply upstream fix for blowfish signed-character bug (CVE-2011-2483) (Tom Lane)
  `contrib/pg_crypto`’s blowfish encryption code could give wrong results on platforms where `char` is signed (which is most), leading to encrypted passwords being weaker than they should be.
- Fix memory leak in `contrib/seg` (Heikki Linnakangas)
- Fix `pgstatindex()` to give consistent results for empty indexes (Tom Lane)
- Allow building with perl 5.14 (Alex Hunsaker)
- Update configure script’s method for probing existence of system functions (Tom Lane)
  The version of autoconf we used in 8.3 and 8.2 could be fooled by compilers that perform link-time optimization.
- Fix assorted issues with build and install file paths containing spaces (Tom Lane)
- Update time zone data files to tzdata release 2011i for DST law changes in Canada, Egypt, Russia, Samoa, and South Sudan.
E.9. Release 8.3.15

**Release Date:** 2011-04-18

This release contains a variety of fixes from 8.3.14. For information about new features in the 8.3 major release, see Section E.24.

### E.9.1. Migration to Version 8.3.15

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.

### E.9.2. Changes

- **Disallow including a composite type in itself (Tom Lane)**
  This prevents scenarios wherein the server could recurse infinitely while processing the composite type. While there are some possible uses for such a structure, they don’t seem compelling enough to justify the effort required to make sure it always works safely.

- **Avoid potential deadlock during catalog cache initialization (Nikhil Sontakke)**
  In some cases the cache loading code would acquire share lock on a system index before locking the index’s catalog. This could deadlock against processes trying to acquire exclusive locks in the other, more standard order.

- **Fix dangling-pointer problem in BEFORE ROW UPDATE trigger handling when there was a concurrent update to the target tuple (Tom Lane)**
  This bug has been observed to result in intermittent “cannot extract system attribute from virtual tuple” failures while trying to do UPDATE RETURNING ctid. There is a very small probability of more serious errors, such as generating incorrect index entries for the updated tuple.

- **Disallow DROP TABLE when there are pending deferred trigger events for the table (Tom Lane)**
  Formerly the DROP would go through, leading to “could not open relation with OID nnn” errors when the triggers were eventually fired.

- **Fix PL/Python memory leak involving array slices (Daniel Popowich)**

- **Fix pg_restore to cope with long lines (over 1KB) in TOC files (Tom Lane)**

- **Put in more safeguards against crashing due to division-by-zero with overly enthusiastic compiler optimization (Aurelien Jarno)**

- **Support use of dlopen() in FreeBSD and OpenBSD on MIPS (Tom Lane)**
  There was a hard-wired assumption that this system function was not available on MIPS hardware on these systems. Use a compile-time test instead, since more recent versions have it.

- **Fix compilation failures on HP-UX (Heikki Linnakangas)**

- **Fix version-incompatibility problem with libintl on Windows (Hiroshi Inoue)**

- **Fix usage of xcopy in Windows build scripts to work correctly under Windows 7 (Andrew Dunstan)**
  This affects the build scripts only, not installation or usage.
Appendix E. Release Notes

- Fix path separator used by pg_regress on Cygwin (Andrew Dunstan)
- Update time zone data files to tzdata release 2011f for DST law changes in Chile, Cuba, Falkland Islands, Morocco, Samoa, and Turkey; also historical corrections for South Australia, Alaska, and Hawaii.

E.10. Release 8.3.14

Release Date: 2011-01-31

This release contains a variety of fixes from 8.3.13. For information about new features in the 8.3 major release, see Section E.24.

E.10.1. Migration to Version 8.3.14

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.

E.10.2. Changes

- Avoid failures when EXPLAIN tries to display a simple-form CASE expression (Tom Lane)
  If the CASE’s test expression was a constant, the planner could simplify the CASE into a form that confused the expression-display code, resulting in “unexpected CASE WHEN clause” errors.
- Fix assignment to an array slice that is before the existing range of subscripts (Tom Lane)
  If there was a gap between the newly added subscripts and the first pre-existing subscript, the code miscalculated how many entries needed to be copied from the old array’s null bitmap, potentially leading to data corruption or crash.
- Avoid unexpected conversion overflow in planner for very distant date values (Tom Lane)
  The date type supports a wider range of dates than can be represented by the timestamp types, but the planner assumed it could always convert a date to timestamp with impunity.
- Fix pg_restore’s text output for large objects (BLOBs) when standard_conforming_strings is on (Tom Lane)
  Although restoring directly to a database worked correctly, string escaping was incorrect if pg_restore was asked for SQL text output and standard_conforming_strings had been enabled in the source database.
- Fix erroneous parsing of tsquery values containing ... & !(subexpression) | ... (Tom Lane)
  Queries containing this combination of operators were not executed correctly. The same error existed in contrib/intarray’s query_int type and contrib/ltree’s ltxtquery type.
- Fix buffer overrun in contrib/intarray’s input function for the query_int type (Apple)
This bug is a security risk since the function’s return address could be overwritten. Thanks to Apple Inc’s security team for reporting this issue and supplying the fix. (CVE-2010-4015)

• Fix bug in contrib/seg’s GiST picksplit algorithm (Alexander Korotkov)
  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a \texttt{seg} column. If you have such an index, consider \texttt{REINDEXing} it after installing this update. (This is identical to the bug that was fixed in contrib/cube in the previous update.)

\section*{E.11. Release 8.3.13}

\textbf{Release Date:} 2010-12-16

This release contains a variety of fixes from 8.3.12. For information about new features in the 8.3 major release, see Section E.24.

\subsection*{E.11.1. Migration to Version 8.3.13}

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.

\subsection*{E.11.2. Changes}

• Force the default \texttt{walync\_method} to be \texttt{fdatasync} on Linux (Tom Lane, Marti Raudsepp)
  The default on Linux has actually been \texttt{fdatasync} for many years, but recent kernel changes caused PostgreSQL to choose \texttt{open\_datasync} instead. This choice did not result in any performance improvement, and caused outright failures on certain filesystems, notably \texttt{ext4} with \texttt{data=journal} mount option.

• Fix assorted bugs in WAL replay logic for GIN indexes (Tom Lane)
  This could result in “bad buffer id: 0” failures or corruption of index contents during replication.

• Fix recovery from base backup when the starting checkpoint WAL record is not in the same WAL segment as its redo point (Jeff Davis)

• Fix persistent slowdown of autovacuum workers when multiple workers remain active for a long time (Tom Lane)
  The effective \texttt{vacuum\_cost\_limit} for an autovacuum worker could drop to nearly zero if it processed enough tables, causing it to run extremely slowly.

• Add support for detecting register-stack overrun on IA64 (Tom Lane)
  The IA64 architecture has two hardware stacks. Full prevention of stack-overrun failures requires checking both.

• Add a check for stack overflow in \texttt{copyObject()} (Tom Lane)
  Certain code paths could crash due to stack overflow given a sufficiently complex query.

• Fix detection of page splits in temporary GiST indexes (Heikki Linnakangas)
Appendix E. Release Notes

It is possible to have a “concurrent” page split in a temporary index, if for example there is an open cursor scanning the index when an insertion is done. GiST failed to detect this case and hence could deliver wrong results when execution of the cursor continued.

- Avoid memory leakage while `ANALYZE`ing complex index expressions (Tom Lane)
- Ensure an index that uses a whole-row `Var` still depends on its table (Tom Lane)
  An index declared like `create index i on t (foo(t.*))` would not automatically get dropped when its table was dropped.
- Do not “inline” a SQL function with multiple `OUT` parameters (Tom Lane)
  This avoids a possible crash due to loss of information about the expected result rowtype.
- Behave correctly if `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `WITH` is attached to the `VALUES` part of `INSERT ... VALUES` (Tom Lane)
- Fix constant-folding of `COALESCE()` expressions (Tom Lane)
  The planner would sometimes attempt to evaluate sub-expressions that in fact could never be reached, possibly leading to unexpected errors.
- Fix postmaster crash when connection acceptance (`accept()` or one of the calls made immediately after it) fails, and the postmaster was compiled with GSSAPI support (Alexander Chernikov)
- Fix missed unlink of temporary files when `log_temp_files` is active (Tom Lane)
  If an error occurred while attempting to emit the log message, the unlink was not done, resulting in accumulation of temp files.
- Add print functionality for `InhRelation` nodes (Tom Lane)
  This avoids a failure when `debug_print_parse` is enabled and certain types of query are executed.
- Fix incorrect calculation of distance from a point to a horizontal line segment (Tom Lane)
  This bug affected several different geometric distance-measurement operators.
- Fix PL/pgSQL’s handling of “simple” expressions to not fail in recursion or error-recovery cases (Tom Lane)
- Fix PL/Python’s handling of set-returning functions (Jan Urbanski)
  Attempts to call SPI functions within the iterator generating a set result would fail.
- Fix bug in `contrib/cube`’s GiST picksplit algorithm (Alexander Korotkov)
  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a `cube` column. If you have such an index, consider `REINDEX`ing it after installing this update.
- Don’t emit “identifier will be truncated” notices in `contrib/dblink` except when creating new connections (Itagaki Takahiro)
- Fix potential coredump on missing public key in `contrib/pgcrypto` (Marti Raudsepp)
- Fix memory leak in `contrib/xml2`’s XPath query functions (Tom Lane)
- Update time zone data files to `tzdata` release 2010o for DST law changes in Fiji and Samoa; also historical corrections for Hong Kong.
Appendix E. Release Notes

E.12. Release 8.3.12

Release Date: 2010-10-04

This release contains a variety of fixes from 8.3.11. For information about new features in the 8.3 major release, see Section E.24.

E.12.1. Migration to Version 8.3.12

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.

E.12.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a SECURITY DEFINER function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function’s owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).

- Prevent possible crashes in pg_get_expr() by disallowing it from being called with an argument that is not one of the system catalog columns it’s intended to be used with (Heikki Linnakangas, Tom Lane)

- Treat exit code 128 (ERROR_WAIT_NO_CHILDREN) as non-fatal on Windows (Magnus Hagander)

  Under high load, Windows processes will sometimes fail at startup with this error code. Formerly the postmaster treated this as a panic condition and restarted the whole database, but that seems to be an overreaction.

- Fix incorrect usage of non-strict OR join clauses in Append indexscans (Tom Lane)

  This is a back-patch of an 8.4 fix that was missed in the 8.3 branch. This corrects an error introduced in 8.3.8 that could cause incorrect results for outer joins when the inner relation is an inheritance tree or UNION ALL subquery.

- Fix possible duplicate scans of UNION ALL member relations (Tom Lane)

- Fix “cannot handle unplanned sub-select” error (Tom Lane)
This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.

- Fix failure to mark cached plans as transient (Tom Lane)
  
  If a plan is prepared while `CREATE INDEX CONCURRENTLY` is in progress for one of the referenced tables, it is supposed to be re-planned once the index is ready for use. This was not happening reliably.

- Reduce PANIC to ERROR in some occasionally-reported btree failure cases, and provide additional detail in the resulting error messages (Tom Lane)
  
  This should improve the system’s robustness with corrupted indexes.

- Prevent `show_session_authorization()` from crashing within autovacuum processes (Tom Lane)

- Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)

- Fix possible failure when hashing a pass-by-reference function result (Tao Ma, Tom Lane)

- Improve merge join’s handling of NULLs in the join columns (Tom Lane)
  
  A merge join can now stop entirely upon reaching the first NULL, if the sort order is such that NULLs sort high.

- Take care to fsync the contents of lockfiles (both `postmaster.pid` and the socket lockfile) while writing them (Tom Lane)
  
  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.

- Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)
  
  The original coding could result in a crash if there was limited stack space.

- Avoid holding open old WAL segments in the walwriter process (Magnus Hagander, Heikki Linnakangas)
  
  The previous coding would prevent removal of no-longer-needed segments.

- Fix `log_line_prefix`'s `%i` escape, which could produce junk early in backend startup (Tom Lane)

- Fix possible data corruption in `ALTER TABLE ... SET TABLESPACE` when archiving is enabled (Jeff Davis)

- Allow `CREATE DATABASE` and `ALTER DATABASE ... SET TABLESPACE` to be interrupted by query-cancel (Guillaume Lelarge)

- Fix `REASSIGN OWNED` to handle operator classes and families (Asko Tiidumaa)

- Fix possible core dump when comparing two empty `tsquery` values (Tom Lane)

- Fix `LIKE`'s handling of patterns containing `%` followed by `_` (Tom Lane)
  
  We’ve fixed this before, but there were still some incorrectly-handled cases.

- In PL/Python, defend against null pointer results from `PyObject_AsVoidPtr` and `PyObject_FromVoidPtr` (Peter Eisentraut)

- Make `psql` recognize `DISCARD ALL` as a command that should not be encased in a transaction block in autocommit-off mode (Itagaki Takahiro)

- Fix `ecpg` to process data from `RETURNING` clauses correctly (Michael Meskes)
Appendix E. Release Notes

• Improve contrib/dblink’s handling of tables containing dropped columns (Tom Lane)
• Fix connection leak after “duplicate connection name” errors in contrib/dblink (Itagaki Takahiro)
• Fix contrib/dblink to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)
• Add hstore(text, text) function to contrib/hstore (Robert Haas)
  This function is the recommended substitute for the now-deprecated => operator. It was back-patched so that future-proofed code can be used with older server versions. Note that the patch will be effective only after contrib/hstore is installed or reinstalled in a particular database. Users might prefer to execute the CREATE FUNCTION command by hand, instead.
• Update build infrastructure and documentation to reflect the source code repository’s move from CVS to Git (Magnus Hagander and others)
• Update time zone data files to tzdata release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.
  This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.
• Make Windows’ “N. Central Asia Standard Time” timezone map to Asia/Novosibirsk, not Asia/Almaty (Magnus Hagander)
  Microsoft changed the DST behavior of this zone in the timezone update from KB976098. Asia/Novosibirsk is a better match to its new behavior.

E.13. Release 8.3.11

Release Date: 2010-05-17

This release contains a variety of fixes from 8.3.10. For information about new features in the 8.3 major release, see Section E.24.

E.13.1. Migration to Version 8.3.11

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.

E.13.2. Changes

• Enforce restrictions in plperl using an opmask applied to the whole interpreter, instead of using Safe.pm (Tim Bunce, Andrew Dunstan)
  Recent developments have convinced us that Safe.pm is too insecure to rely on for making plperl trustable. This change removes use of Safe.pm altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is
Appendix E. Release Notes

now possible to use Perl’s strict pragma in a natural way in plperl, and that Perl’s $a and $b variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

• Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

PL/Tcl’s feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted “normal” Tcl interpreter unless we are really going to execute a pltclu function. (CVE-2010-1170)

• Fix possible crash if a cache reset message is received during rebuild of a relcache entry (Heikki)

This error was introduced in 8.3.10 while fixing a related failure.

• Apply per-function GUC settings while running the language validator for the function (Itagaki Takahiro)

This avoids failures if the function’s code is invalid without the setting; an example is that SQL functions may not parse if the search_path is not correct.

• Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

Previously, if an unprivileged user ran ALTER USER ... RESET ALL for himself, or ALTER DATABASE ... RESET ALL for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the ALTER will only remove the parameters that the user has permission to change.

• Avoid possible crash during backend shutdown if shutdown occurs when a CONTEXT addition would be made to log entries (Tom)

In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

• Ensure the archiver process responds to changes in archive_command as soon as possible (Tom)

• Update pl/perl’s ppport.h for modern Perl versions (Andrew)

• Fix assorted memory leaks in pl/python (Andreas Freund, Tom)

• Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)

• Fix psql’s \copy to not add spaces around a dot within \copy (select ...) (Tom)

Addition of spaces around the decimal point in a numeric literal would result in a syntax error.

• Fix unnecessary “GIN indexes do not support whole-index scans” errors for unsatisfiable queries using contrib/intarray operators (Tom)

• Ensure that contrib/pgstattuple functions respond to cancel interrupts promptly (Tatsuhito Kasahara)

• Make server startup deal properly with the case that shmget() returns EINVAL for an existing shared memory segment (Tom)

This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.

• Avoid possible crashes in syslogger process on Windows (Heikki)

• Deal more robustly with incomplete time zone information in the Windows registry (Magnus)

• Update the set of known Windows time zone names (Magnus)
Appendix E. Release Notes

• Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.

Also, add PKST (Pakistan Summer Time) to the default set of timezone abbreviations.

E.14. Release 8.3.10

Release Date: 2010-03-15

This release contains a variety of fixes from 8.3.9. For information about new features in the 8.3 major release, see Section E.24.

E.14.1. Migration to Version 8.3.10

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.

E.14.2. Changes

• Add new configuration parameter ssl_renegotiation_limit to control how often we do session key renegotiation for an SSL connection (Magnus)

This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.

• Fix possible deadlock during backend startup (Tom)

• Fix possible crashes due to not handling errors during relcache reload cleanly (Tom)

• Fix possible crash due to use of dangling pointer to a cached plan (Tatsuo)

• Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)

• Fix server memory leak associated with use of savepoints and a client encoding different from server’s encoding (Tom)

• Fix incorrect WAL data emitted during end-of-recovery cleanup of a GIST index page split (Yoichi Hirai)

This would result in index corruption, or even more likely an error during WAL replay, if we were unlucky enough to crash during end-of-recovery cleanup after having completed an incomplete GIST insertion.

• Make substring() for bit types treat any negative length as meaning “all the rest of the string” (Tom)

The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).
Appendix E. Release Notes

- Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)
- Fix some cases of pathologically slow regular expression matching (Tom)
- Fix assorted crashes in xml processing caused by sloppy memory management (Tom)
  This is a back-patch of changes first applied in 8.4. The 8.3 code was known buggy, but the new code was sufficiently different to not want to back-patch it until it had gotten some field testing.
- Fix bug with trying to update a field of an element of a composite-type array column (Tom)
- Fix the STOP WAL LOCATION entry in backup history files to report the next WAL segment’s name when the end location is exactly at a segment boundary (Itagaki Takahiro)
- Fix some more cases of temporary-file leakage (Heikki)
  This corrects a problem introduced in the previous minor release. One case that failed is when a plpgsql function returning set is called within another function’s exception handler.
- Improve constraint exclusion processing of boolean-variable cases, in particular make it possible to exclude a partition that has a "bool_column = false" constraint (Tom)
- When reading pg_hba.conf and related files, do not treat @something as a file inclusion request if the @ appears inside quote marks; also, never treat @ by itself as a file inclusion request (Tom)
  This prevents erratic behavior if a role or database name starts with @. If you need to include a file whose path name contains spaces, you can still do so, but you must write @"/path to/file" rather than putting the quotes around the whole construct.
- Prevent infinite loop on some platforms if a directory is named as an inclusion target in pg_hba.conf and related files (Tom)
- Fix possible infinite loop if SSL_read or SSL_write fails without setting errno (Tom)
  This is reportedly possible with some Windows versions of openssl.
- Disallow GSSAPI authentication on local connections, since it requires a hostname to function correctly (Magnus)
- Make ecpg report the proper SQLSTATE if the connection disappears (Michael)
- Fix psql's numericlocale option to not format strings it shouldn’t in latex and troff output formats (Heikki)
- Make psql return the correct exit status (3) when ON_ERROR_STOP and --single-transaction are both specified and an error occurs during the implied COMMIT (Bruce)
- Fix plpgsql failure in one case where a composite column is set to NULL (Tom)
- Fix possible failure when calling PL/Perl functions from PL/PerlU or vice versa (Tim Bunce)
- Add volatile markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)
- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)
  The only known symptom of this oversight is that the Tcl clock command misbehaves if using Tcl 8.5 or later.
- Prevent crash in contrib/dblink when too many key columns are specified to a dblink_build_sql_* function (Rushabh Lathia, Joe Conway)
- Allow zero-dimensional arrays in contrib/ltree operations (Tom)
This case was formerly rejected as an error, but it’s more convenient to treat it the same as a zero-element array. In particular this avoids unnecessary failures when an \texttt{ltree} operation is applied to the result of \texttt{ARRAY(SELECT ...)} and the sub-select returns no rows.

- Fix assorted crashes in \texttt{contrib/xml2} caused by sloppy memory management (Tom)
- Make building of \texttt{contrib/xml2} more robust on Windows (Andrew)
- Fix race condition in Windows signal handling (Radu Ilie)

One known symptom of this bug is that rows in \texttt{pg_listener} could be dropped under heavy load.

- Update time zone data files to \texttt{tzdata} release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

\section*{E.15. Release 8.3.9}

\textbf{Release Date:} 2009-12-14

This release contains a variety of fixes from 8.3.8. For information about new features in the 8.3 major release, see Section E.24.

\subsection*{E.15.1. Migration to Version 8.3.9}

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.8, see the release notes for 8.3.8.

\subsection*{E.15.2. Changes}

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)

This change prevents allegedly-immutable index functions from possibly subverting a superuser’s session (CVE-2009-4136).

- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)

This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).

- Fix possible crash during backend-startup-time cache initialization (Tom)

- Avoid crash on empty thesaurus dictionary (Tom)

- Prevent signals from interrupting \texttt{VACUUM} at unsafe times (Alvaro)

This fix prevents a PANIC if a \texttt{VACUUM FULL} is canceled after it’s already committed its tuple movements, as well as transient errors if a plain \texttt{VACUUM} is interrupted after having truncated the table.

- Fix possible crash due to integer overflow in hash table size calculation (Tom)

This could occur with extremely large planner estimates for the size of a hashjoin’s result.
Appendix E. Release Notes

- Fix very rare crash in `inet/cidr` comparisons (Chris Mikkelson)
- Ensure that shared tuple-level locks held by prepared transactions are not ignored (Heikki)
- Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction (Heikki)
- Fix memory leak in syslogger process when rotating to a new CSV logfile (Tom)
- Fix Windows permission-downgrade logic (Jesse Morris)

This fixes some cases where the database failed to start on Windows, often with misleading error messages such as “could not locate matching postgres executable”.
- Fix incorrect logic for GiST index page splits, when the split depends on a non-first column of the index (Paul Ramsey)
- Don’t error out if recycling or removing an old WAL file fails at the end of checkpoint (Heikki)

It’s better to treat the problem as non-fatal and allow the checkpoint to complete. Future checkpoints will retry the removal. Such problems are not expected in normal operation, but have been seen to be caused by misdesigned Windows anti-virus and backup software.
- Ensure WAL files aren’t repeatedly archived on Windows (Heikki)

This is another symptom that could happen if some other process interfered with deletion of a no-longer-needed file.
- Fix PAM password processing to be more robust (Tom)

The previous code is known to fail with the combination of the Linux `pam_krb5` PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.
- Raise the maximum authentication token (Kerberos ticket) size in GSSAPI and SSPI authentication methods (Ian Turner)

While the old 2000-byte limit was more than enough for Unix Kerberos implementations, tickets issued by Windows Domain Controllers can be much larger.
- Re-enable collection of access statistics for sequences (Akira Kurosawa)

This used to work but was broken in 8.3.
- Fix processing of ownership dependencies during `CREATE OR REPLACE FUNCTION` (Tom)
- Fix incorrect handling of WHERE `x=x` conditions (Tom)

In some cases these could get ignored as redundant, but they aren’t — they’re equivalent to `x IS NOT NULL`.
- Make text search parser accept underscores in XML attributes (Peter)
- Fix encoding handling in XML binary input (Heikki)

If the XML header doesn’t specify an encoding, we now assume UTF-8 by default; the previous handling was inconsistent.
- Fix bug with calling `plperl` from `plperlu` or vice versa (Tom)

An error exit from the inner function could result in crashes due to failure to re-select the correct Perl interpreter for the outer function.
- Fix session-lifespan memory leak when a PL/Perl function is redefined (Tom)
- Ensure that Perl arrays are properly converted to PostgreSQL arrays when returned by a set-returning PL/Perl function (Andrew Dunstan, Abhijit Menon-Sen)
This worked correctly already for non-set-returning functions.

- Fix rare crash in exception processing in PL/Python (Peter)
- In contrib/pg_standby, disable triggering failover with a signal on Windows (Fujii Masao)
  This never did anything useful, because Windows doesn’t have Unix-style signals, but recent
  changes made it actually crash.
- Ensure psql’s flex module is compiled with the correct system header definitions (Tom)
  This fixes build failures on platforms where --enable-largefile causes incompatible changes
  in the generated code.
- Make the postmaster ignore any application_name parameter in connection request packets, to
  improve compatibility with future libpq versions (Tom)
- Update the timezone abbreviation files to match current reality (Joachim Wieland)
  This includes adding IDT and SGT to the default timezone abbreviation set.
- Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina,
  Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for
  Hong Kong.

E.16. Release 8.3.8

Release Date: 2009-09-09

This release contains a variety of fixes from 8.3.7. For information about new features in the 8.3 major
release, see Section E.24.

E.16.1. Migration to Version 8.3.8

A dump/restore is not required for those running 8.3.X. However, if you have any hash indexes on
interval columns, you must REINDEX them after updating to 8.3.8. Also, if you are upgrading from
a version earlier than 8.3.5, see the release notes for 8.3.5.

E.16.2. Changes

- Fix Windows shared-memory allocation code (Tsutomu Yamada, Magnus)
  This bug led to the often-reported “could not reattach to shared memory” error message.
- Force WAL segment switch during pg_start_backup() (Heikki)
  This avoids corner cases that could render a base backup unusable.
- Disallow RESET ROLE and RESET SESSION AUTHORIZATION inside security-definer functions
  (Tom, Heikki)
  This covers a case that was missed in the previous patch that disallowed SET ROLE and SET
  SESSION AUTHORIZATION inside security-definer functions. (See CVE-2007-6600)
• Make `LOAD` of an already-loaded loadable module into a no-op (Tom)
  Formerly, `LOAD` would attempt to unload and re-load the module, but this is unsafe and not all that useful.
• Disallow empty passwords during LDAP authentication (Magnus)
• Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)
• Fix bugs associated with fetching a whole-row value from the output of a Sort or Materialize plan node (Tom)
• Prevent `synchronize_seqscans` from changing the results of scrollable and WITH HOLD cursors (Tom)
• Revert planner change that disabled partial-index and constraint exclusion optimizations when there were more than 100 clauses in an AND or OR list (Tom)
• Fix hash calculation for data type `interval` (Tom)
  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must `REINDEX` them after updating.
• Treat `to_char(..., 'TH')` as an uppercase ordinal suffix with ‘HH’/’HH12’ (Heikki)
  It was previously handled as ‘th’ (lowercase).
• Fix overflow for `INTERVAL ‘x ms’` when x is more than 2 million and integer datetimes are in use (Alex Hunsaker)
• Fix calculation of distance between a point and a line segment (Tom)
  This led to incorrect results from a number of geometric operators.
• Fix `money` data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)
• Fix `LIKE` for case where pattern contains `%_` (Tom)
• Properly round datetime input like `00:12:57.9999999999999999999999999999` (Tom)
• Fix memory leaks in XML operations (Tom)
• Fix poor choice of page split point in GiST R-tree operator classes (Teodor)
• Ensure that a “fast shutdown” request will forcibly terminate open sessions, even if a “smart shut-
down” was already in progress (Fujii Masao)
• Avoid performance degradation in bulk inserts into GIN indexes when the input values are (nearly) in sorted order (Tom)
• Correctly enforce NOT NULL domain constraints in some contexts in PL/pgSQL (Tom)
• Fix portability issues in plperl initialization (Andrew Dunstan)
• Fix `pg_ctl` to not go into an infinite loop if `postgresql.conf` is empty (Jeff Davis)
• Improve `pg_dump`’s efficiency when there are many large objects (Tamas Vincze)
• Use `SIGUSR1`, not `SIGQUIT`, as the failover signal for `pg_standby` (Heikki)
• Make `pg_standby`’s `maxretries` option behave as documented (Fujii Masao)
• Make `contrib/hstore` throw an error when a key or value is too long to fit in its data structure, rather than silently truncating it (Andrew Gierth)
• Fix `contrib/xml2`’s `xslt_process()` to properly handle the maximum number of parameters (twenty) (Tom)
• Improve robustness of libpq’s code to recover from errors during COPY FROM STDIN (Tom)
• Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)
• Update time zone data files to tzdata release 2009l for DST law changes in Bangladesh, Egypt, Jordan, Pakistan, Argentina/San Luis, Cuba, Jordan (historical correction only), Mauritius, Morocco, Palestine, Syria, Tunisia.

E.17. Release 8.3.7

Release Date: 2009-03-16

This release contains a variety of fixes from 8.3.6. For information about new features in the 8.3 major release, see Section E.24.

E.17.1. Migration to Version 8.3.7

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.5, see the release notes for 8.3.5.

E.17.2. Changes

• Prevent error recursion crashes when encoding conversion fails (Tom)
  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that any error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)

•Disallow CREATE CONVERSION with the wrong encodings for the specified conversion function (Heikki)
  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.

• Fix xpath() to not modify the path expression unless necessary, and to make a saner attempt at it when necessary (Andrew)
  The SQL standard suggests that xpath should work on data that is a document fragment, but libxml doesn’t support that, and indeed it’s not clear that this is sensible according to the XPath standard. xpath attempted to work around this mismatch by modifying both the data and the path expression, but the modification was buggy and could cause valid searches to fail. Now, xpath checks whether the data is in fact a well-formed document, and if so invokes libxml with no change to the data or path expression. Otherwise, a different modification method that is somewhat less likely to fail is used.
Appendix E. Release Notes

Note: The new modification method is still not 100% satisfactory, and it seems likely that no real solution is possible. This patch should therefore be viewed as a band-aid to keep from breaking existing applications unnecessarily. It is likely that PostgreSQL 8.4 will simply reject use of xpath on data that is not a well-formed document.

- Fix core dump when `to_char()` is given format codes that are inappropriate for the type of the data argument (Tom)
- Fix possible failure in text search when C locale is used with a multi-byte encoding (Teodor)
  Crashes were possible on platforms where wchar_t is narrower than int; Windows in particular.
- Fix extreme inefficiency in text search parser’s handling of an email-like string containing multiple @ characters (Heikki)
- Fix planner problem with sub-SELECT in the output list of a larger subquery (Tom)
  The known symptom of this bug is a “failed to locate grouping columns” error that is dependent on the datatype involved; but there could be other issues as well.
- Fix decompilation of CASE WHEN with an implicit coercion (Tom)
  This mistake could lead to Assert failures in an Assert-enabled build, or an “unexpected CASE WHEN clause” error message in other cases, when trying to examine or dump a view.
- Fix possible misassignment of the owner of a TOAST table’s rowtype (Tom)
  If CLUSTER or a rewriting variant of ALTER TABLE were executed by someone other than the table owner, the pg_type entry for the table’s TOAST table would end up marked as owned by that someone. This caused no immediate problems, since the permissions on the TOAST rowtype aren’t examined by any ordinary database operation. However, it could lead to unexpected failures if one later tried to drop the role that issued the command (in 8.1 or 8.2), or “owner of data type appears to be invalid” warnings from pg_dump after having done so (in 8.3).
- Change UNLISTEN to exit quickly if the current session has never executed any LISTEN command (Tom)
  Most of the time this is not a particularly useful optimization, but since DISCARD ALL invokes UNLISTEN, the previous coding caused a substantial performance problem for applications that made heavy use of DISCARD ALL.
- Fix PL/pgSQL to not treat INTO after INSERT as an INTO-variables clause anywhere in the string, not only at the start; in particular, don’t fail for INSERT INTO within CREATE RULE (Tom)
- Clean up PL/pgSQL error status variables fully at block exit (Ashesh Vashi and Dave Page)
  This is not a problem for PL/pgSQL itself, but the omission could cause the PL/pgSQL Debugger to crash while examining the state of a function.
- Retry failed calls to CallNamedPipe() on Windows (Steve Marshall, Magnus)
  It appears that this function can sometimes fail transiently; we previously treated any failure as a hard error, which could confuse LISTEN/NOTIFY as well as other operations.
- Add MUST (Mauritius Island Summer Time) to the default list of known timezone abbreviations (Xavier Bugaud)
Appendix E. Release Notes

E.18. Release 8.3.6

Release Date: 2009-02-02

This release contains a variety of fixes from 8.3.5. For information about new features in the 8.3 major release, see Section E.24.

E.18.1. Migration to Version 8.3.6

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.5, see the release notes for 8.3.5.

E.18.2. Changes

• Make \texttt{DISCARD ALL} release advisory locks, in addition to everything it already did (Tom)
  This was decided to be the most appropriate behavior. This could affect existing applications, however.

• Fix whole-index GiST scans to work correctly (Teodor)
  This error could cause rows to be lost if a table is clustered on a GiST index.

• Fix crash of \texttt{xmlconcat(NULL)} (Peter)

• Fix possible crash in \texttt{ispell} dictionary if high-bit-set characters are used as flags (Teodor)
  This is known to be done by one widely available Norwegian dictionary, and the same condition may exist in others.

• Fix misordering of pg\_dump output for composite types (Tom)
  The most likely problem was for user-defined operator classes to be dumped after indexes or views that needed them.

• Improve handling of URLs in \texttt{headline()} function (Teodor)

• Improve handling of overlength headlines in \texttt{headline()} function (Teodor)

• Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)

• Fix possible Assert failure if a statement executed in PL/pgSQL is rewritten into another kind of statement, for example if an \texttt{INSERT} is rewritten into an \texttt{UPDATE} (Heikki)

• Ensure that a snapshot is available to datatype input functions (Tom)
  This primarily affects domains that are declared with \texttt{CHECK} constraints involving user-defined stable or immutable functions. Such functions typically fail if no snapshot has been set.

• Make it safer for SPI-using functions to be used within datatype I/O; in particular, to be used in domain check constraints (Tom)

• Avoid unnecessary locking of small tables in \texttt{VACUUM} (Heikki)

• Fix a problem that sometimes kept \texttt{ALTER TABLE ENABLE/DISABLE RULE} from being recognized by active sessions (Tom)
• Fix a problem that made UPDATE RETURNING tableoid return zero instead of the correct OID (Tom)

• Allow functions declared as taking ANYARRAY to work on the pg_statistic columns of that type (Tom)
  This used to work, but was unintentionally broken in 8.3.

• Fix planner misestimation of selectivity when transitive equality is applied to an outer-join clause (Tom)
  This could result in bad plans for queries like ...

  from a left join b on a.a1 = b.b1
  where a.a1 = 42 ...

• Improve optimizer's handling of long IN lists (Tom)
  This change avoids wasting large amounts of time on such lists when constraint exclusion is en-
  abled.

• Prevent synchronous scan during GIN index build (Tom)
  Because GIN is optimized for inserting tuples in increasing TID order, choosing to use a syn-
  chronous scan could slow the build by a factor of three or more.

• Ensure that the contents of a holdable cursor don’t depend on the contents of TOAST tables (Tom)
  Previously, large field values in a cursor result might be represented as TOAST pointers, which
  would fail if the referenced table got dropped before the cursor is read, or if the large value is
  deleted and then vacuumed away. This cannot happen with an ordinary cursor, but it could with a
cursor that is held past its creating transaction.

• Fix memory leak when a set-returning function is terminated without reading its whole result (Tom)

• Fix encoding conversion problems in XML functions when the database encoding isn’t UTF-8 (Tom)
  • Fix contrib/dblink’s dblink_get_result (text, bool) function (Joe)
  • Fix possible garbage output from contrib/sslinfo functions (Tom)
  • Fix incorrect behavior of contrib/tsearch2 compatibility trigger when it’s fired more than once
    in a command (Teodor)
  • Fix possible mis-signaling in autovacuum (Heikki)
  • Support running as a service on Windows 7 beta (Dave and Magnus)
  • Fix ecpg’s handling of varchar structs (Michael)
  • Fix configure script to properly report failure when unable to obtain linkage information for PL/Perl
    (Andrew)
  • Make all documentation reference pgsq1-bugs and/or pgsq1-hackers as appropriate, instead
    of the now-decommissioned pgsq1-ports and pgsq1-patches mailing lists (Tom)
  • Update time zone data files to tzdata release 2009a (for Kathmandu and historical DST corrections
    in Switzerland, Cuba)

  E.19. Release 8.3.5

  Release Date: 2008-11-03
Appendix E. Release Notes

This release contains a variety of fixes from 8.3.4. For information about new features in the 8.3 major release, see Section E.24.

E.19.1. Migration to Version 8.3.5

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.1, see the release notes for 8.3.1. Also, if you were running a previous 8.3.X release, it is recommended to \texttt{REINDEX} all GiST indexes after the upgrade.

E.19.2. Changes

- Fix GiST index corruption due to marking the wrong index entry “dead” after a deletion (Teodor)
  This would result in index searches failing to find rows they should have found. Corrupted indexes can be fixed with \texttt{REINDEX}.
- Fix backend crash when the client encoding cannot represent a localized error message (Tom)
  We have addressed similar issues before, but it would still fail if the “character has no equivalent” message itself couldn’t be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.
- Fix possible crash in bytea-to-XML mapping (Michael McMaster)
- Fix possible crash when deeply nested functions are invoked from a trigger (Tom)
- Improve optimization of \texttt{expression IN (expression-list)} queries (Tom, per an idea from Robert Haas)
  Cases in which there are query variables on the right-hand side had been handled less efficiently in 8.2.x and 8.3.x than in prior versions. The fix restores 8.1 behavior for such cases.
- Fix mis-expansion of rule queries when a sub-\texttt{SELECT} appears in a function call in \texttt{FROM}, a multi-row \texttt{VALUES} list, or a \texttt{RETURNING} list (Tom)
  The usual symptom of this problem is an “unrecognized node type” error.
- Fix Assert failure during rescan of an \texttt{IS NULL} search of a GiST index (Teodor)
- Fix memory leak during rescan of a hashed aggregation plan (Neil)
- Ensure an error is reported when a newly-defined PL/pgSQL trigger function is invoked as a normal function (Tom)
- Force a checkpoint before \texttt{CREATE DATABASE} starts to copy files (Heikki)
  This prevents a possible failure if files had recently been deleted in the source database.
- Prevent possible collision of \texttt{relfilenode} numbers when moving a table to another tablespace with \texttt{ALTER SET TABLESPACE} (Heikki)
  The command tried to re-use the existing filename, instead of picking one that is known unused in the destination directory.
- Fix incorrect text search headline generation when single query item matches first word of text (Sushant Sinha)
• Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an
--enable-integer-datetimes build (Ron Mayer)
• Make ILIKE compare characters case-insensitively even when they’re escaped (Andrew)
• Ensure DISCARD is handled properly by statement logging (Tom)
• Fix incorrect logging of last-completed-transaction time during PITR recovery (Tom)
• Ensure SPI_getvalue and SPI_getbinval behave correctly when the passed tuple and tuple
descriptor have different numbers of columns (Tom)
  This situation is normal when a table has had columns added or removed, but these two functions
didn’t handle it properly. The only likely consequence is an incorrect error indication.
• Mark SessionReplicationRole as PGDLLIMPORT so it can be used by Slony on Windows
  (Magnus)
• Fix small memory leak when using libpq’s gsslib parameter (Magnus)
  The space used by the parameter string was not freed at connection close.
• Ensure libgssapi is linked into libpq if needed (Markus Schaaf)
• Fix ecpg’s parsing of CREATE ROLE (Michael)
• Fix recent breakage of pg_ctl restart (Tom)
• Ensure pg_control is opened in binary mode (Itagaki Takahiro)
  pg_controldata and pg_resetxlog did this incorrectly, and so could fail on Windows.
• Update time zone data files to tzdata release 2008i (for DST law changes in Argentina, Brazil,
  Mauritius, Syria)

E.20. Release 8.3.4

Release Date: 2008-09-22

This release contains a variety of fixes from 8.3.3. For information about new features in the 8.3 major
release, see Section E.24.

E.20.1. Migration to Version 8.3.4

A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version
earlier than 8.3.1, see the release notes for 8.3.1.

E.20.2. Changes

• Fix bug in btree WAL recovery code (Heikki)
  Recovery failed if the WAL ended partway through a page split operation.
• Fix potential use of wrong cutoff XID for HOT page pruning (Alvaro)
This error created a risk of corruption in system catalogs that are consulted by `VACUUM`: dead tuple versions might be removed too soon. The impact of this on actual database operations would be minimal, since the system doesn’t follow MVCC rules while examining catalogs, but it might result in transiently wrong output from `pg_dump` or other client programs.

- **Fix potential miscalculation of `datfrozenxid` (Alvaro)**
  This error may explain some recent reports of failure to remove old `pg_clog` data.

- **Fix incorrect HOT updates after `pg_class` is reindexed (Tom)**
  Corruption of `pg_class` could occur if `REINDEX TABLE pg_class` was followed in the same session by an `ALTER TABLE RENAME` or `ALTER TABLE SET SCHEMA` command.

- **Fix missed “combo cid” case (Karl Schnaitter)**
  This error made rows incorrectly invisible to a transaction in which they had been deleted by multiple subtransactions that all aborted.

- **Prevent autovacuum from crashing if the table it’s currently checking is deleted at just the wrong time (Alvaro)**

- **Widen local lock counters from 32 to 64 bits (Tom)**
  This responds to reports that the counters could overflow in sufficiently long transactions, leading to unexpected “lock is already held” errors.

- **Fix possible duplicate output of tuples during a GiST index scan (Teodor)**

- **Regenerate foreign key checking queries from scratch when either table is modified (Tom)**
  Previously, 8.3 would attempt to replan the query, but would work from previously generated query text. This led to failures if a table or column was renamed.

- **Fix missed permissions checks when a view contains a simple `UNION ALL` construct (Heikki)**
  Permissions for the referenced tables were checked properly, but not permissions for the view itself.

- **Add checks in executor startup to ensure that the tuples produced by an `INSERT` or `UPDATE` will match the target table’s current rowtype (Tom)**
  This situation is believed to be impossible in 8.3, but it can happen in prior releases, so a check seems prudent.

- **Fix possible repeated drops during `DROP OWNED` (Tom)**
  This would typically result in strange errors such as “cache lookup failed for relation NNN”.

- **Fix several memory leaks in XML operations (Kris Jurka, Tom)**

- **Fix `xmlserialize()` to raise error properly for unacceptable target data type (Tom)**

- **Fix a couple of places that mis-handled multibyte characters in text search configuration file parsing (Tom)**
  Certain characters occurring in configuration files would always cause “invalid byte sequence for encoding” failures.

- **Provide file name and line number location for all errors reported in text search configuration files (Tom)**

- **Fix `AT TIME ZONE` to first try to interpret its timezone argument as a timezone abbreviation, and only try it as a full timezone name if that fails, rather than the other way around as formerly (Tom)**
  The timestamp input functions have always resolved ambiguous zone names in this order. Making `AT TIME ZONE` do so as well improves consistency, and fixes a compatibility bug introduced in
Appendix E. Release Notes

8.1: in ambiguous cases we now behave the same as 8.0 and before did, since in the older versions AT TIME ZONE accepted only abbreviations.

- Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)
- Prevent integer overflows during units conversion when displaying a configuration parameter that has units (Tom)
- Improve performance of writing very long log messages to syslog (Tom)
- Allow spaces in the suffix part of an LDAP URL in pg_hba.conf (Tom)
- Fix bug in backwards scanning of a cursor on a SELECT DISTINCT ON query (Tom)
- Fix planner bug that could improperly push down IS NULL tests below an outer join (Tom)
  This was triggered by occurrence of IS NULL tests for the same relation in all arms of an upper OR clause.
- Fix planner bug with nested sub-select expressions (Tom)
  If the outer sub-select has no direct dependency on the parent query, but the inner one does, the outer value might not get recalculated for new parent query rows.
- Fix planner to estimate that GROUP BY expressions yielding boolean results always result in two groups, regardless of the expressions’ contents (Tom)
  This is very substantially more accurate than the regular GROUP BY estimate for certain boolean tests like col IS NULL.
- Fix pgSQL to not fail when a FOR loop’s target variable is a record containing composite-type fields (Tom)
- Fix PL/Tcl to behave correctly with Tcl 8.5, and to be more careful about the encoding of data sent to or from Tcl (Tom)
- Improve performance of PQescapeBytea() (Rudolf Leitgeb)
- On Windows, work around a Microsoft bug by preventing libpq from trying to send more than 64kB per system call (Magnus)
- Fix ecpg to handle variables properly in SET commands (Michael)
- Improve pg_dump and pg_restore’s error reporting after failure to send a SQL command (Tom)
- Fix pg_ctl to properly preserve postmaster command-line arguments across a restart (Bruce)
- Fix erroneous WAL file cutoff point calculation in pg_standby (Simon)
- Update time zone data files to tzdata release 2008f (for DST law changes in Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine, and Paraguay)

E.21. Release 8.3.3

Release Date: 2008-06-12

This release contains one serious and one minor bug fix over 8.3.2. For information about new features in the 8.3 major release, see Section E.24.
Appendix E. Release Notes

E.21.1. Migration to Version 8.3.3
A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.1, see the release notes for 8.3.1.

E.21.2. Changes

- Make `pg_get_ruledef()` parenthesize negative constants (Tom)
  Before this fix, a negative constant in a view or rule might be dumped as, say, `-42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL to reject what had been a valid `SELECT DISTINCT` view query. Since this could result in pg_dump output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.

- Make `ALTER AGGREGATE ... OWNER TO` update `pg_shdepend` (Tom)
  This oversight could lead to problems if the aggregate was later involved in a `DROP OWNED` or `REASSIGN OWNED` operation.

E.22. Release 8.3.2

**Release Date:** never released

This release contains a variety of fixes from 8.3.1. For information about new features in the 8.3 major release, see Section E.24.

E.22.1. Migration to Version 8.3.2
A dump/restore is not required for those running 8.3.X. However, if you are upgrading from a version earlier than 8.3.1, see the release notes for 8.3.1.

E.22.2. Changes

- Fix `ERRORDATA_STACK_SIZE exceeded` crash that occurred on Windows when using UTF-8 database encoding and a different client encoding (Tom)

- Fix incorrect archive truncation point calculation for the `%r` macro in `recovery_command parameters` (Simon)
  This could lead to data loss if a warm-standby script relied on `%r` to decide when to throw away WAL segment files.

- Fix `ALTER TABLE ADD COLUMN ... PRIMARY KEY` so that the new column is correctly checked to see if it’s been initialized to all non-nulls (Brendan Jurd)
Appendix E. Release Notes

Previous versions neglected to check this requirement at all.

- Fix REASSIGN OWNED so that it works on procedural languages too (Alvaro)
- Fix problems with SELECT FOR UPDATE/SHARE occurring as a subquery in a query with a non-SELECT top-level operation (Tom)
- Fix possible CREATE TABLE failure when inheriting the “same” constraint from multiple parent relations that inherited that constraint from a common ancestor (Tom)
- Fix pg_get_ruledef() to show the alias, if any, attached to the target table of an UPDATE or DELETE (Tom)
- Restore the pre-8.3 behavior that an out-of-range block number in a TID being used in a TidScan plan results in silently not matching any rows (Tom)

8.3.0 and 8.3.1 threw an error instead.

- Fix GIN bug that could result in a too many LWLocks taken failure (Teodor)
- Fix broken GiST comparison function for tsquery (Teodor)
- Fix tsvector_update_trigger() and ts_stat() to accept domains over the types they expect to work with (Tom)
- Fix failure to support enum data types as foreign keys (Tom)
- Avoid possible crash when decompressing corrupted data (Zdenek Kotala)
- Fix race conditions between delayed unlinks and DROP DATABASE (Heikki)

In the worst case this could result in deleting a newly created table in a new database that happened to get the same OID as the recently-dropped one; but of course that is an extremely low-probability scenario.

- Repair two places where SIGTERM exit of a backend could leave corrupted state in shared memory (Tom)

Neither case is very important if SIGTERM is used to shut down the whole database cluster together, but there was a problem if someone tried to SIGTERM individual backends.

- Fix possible crash due to incorrect plan generated for an \( x \) IN (SELECT \( y \) FROM ...) clause when \( x \) and \( y \) have different data types; and make sure the behavior is semantically correct when the conversion from \( y \)’s type to \( x \)’s type is lossy (Tom)

- Fix oversight that prevented the planner from substituting known Param values as if they were constants (Tom)

This mistake partially disabled optimization of unnamed extended-Query statements in 8.3.0 and 8.3.1: in particular the LIKE-to-indexscan optimization would never be applied if the LIKE pattern was passed as a parameter, and constraint exclusion depending on a parameter value didn’t work either.

- Fix planner failure when an indexable MIN or MAX aggregate is used with DISTINCT or ORDER BY (Tom)

- Fix planner to ensure it never uses a “physical tlist” for a plan node that is feeding a Sort node (Tom)

This led to the sort having to push around more data than it really needed to, since unused column values were included in the sorted data.

- Avoid unnecessary copying of query strings (Tom)

This fixes a performance problem introduced in 8.3.0 when a very large number of commands are submitted as a single query string.

1510
Appendix E. Release Notes

- Make TransactionIdIsCurrentTransactionId() use binary search instead of linear search when checking child-transaction XIDs (Heikki)

  This fixes some cases in which 8.3.0 was significantly slower than earlier releases.

- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic “Yo” characters (e and E with two dots) (Sergey Burladyan)

- Fix several datatype input functions, notably array_in(), that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)

  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched ORDER BY and DISTINCT expressions.

- Fix a corner case in regular-expression substring matching (substring(string from pattern)) (Tom)

  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn’t got a match. An example is substring(‘foo’ from ‘foo(bar)?’). This should return NULL, since (bar) isn’t matched, but it was mistakenly returning the whole-pattern match instead (ie, foo).

- Prevent cancellation of an auto-vacuum that was launched to prevent XID wraparound (Alvaro)

- Improve ANALYZE’s handling of in-doubt tuples (those inserted or deleted by a not-yet-committed transaction) so that the counts it reports to the stats collector are more likely to be correct (Pavan Deolasee)

- Fix initdb to reject a relative path for its --xlogdir (-X) option (Tom)

- Make psql print tab characters as an appropriate number of spaces, rather than \x09 as was done in 8.3.0 and 8.3.1 (Bruce)

- Update time zone data files to tzdata release 2008c (for DST law changes in Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, and Argentina/San_Luis)

- Add ECPGget_PGconn() function to ecpglib (Michael)

- Fix incorrect result from ecpg’s PGTYPEStimestamp_sub() function (Michael)

- Fix handling of continuation line markers in ecpg (Michael)

- Fix possible crashes in contrib/cube functions (Tom)

- Fix core dump in contrib/xml2’s xpath_table() function when the input query returns a NULL value (Tom)

- Fix contrib/xml2’s makefile to not override CFLAGS, and make it auto-configure properly for libxslt present or not (Tom)

E.23. Release 8.3.1

Release Date: 2008-03-17

This release contains a variety of fixes from 8.3.0. For information about new features in the 8.3 major release, see Section E.24.
E.23.1. Migration to Version 8.3.1

A dump/restore is not required for those running 8.3.X. However, you might need to **REINDEX** indexes on textual columns after updating, if you are affected by the Windows locale issue described below.

E.23.2. Changes

- Fix character string comparison for Windows locales that consider different character combinations as equal (Tom)
  
  This fix applies only on Windows and only when using UTF-8 database encoding. The same fix was made for all other cases over two years ago, but Windows with UTF-8 uses a separate code path that was not updated. If you are using a locale that considers some non-identical strings as equal, you may need to **REINDEX** to fix existing indexes on textual columns.

- Repair corner-case bugs in **VACUUM FULL** (Tom)
  
  A potential deadlock between concurrent **VACUUM FULL** operations on different system catalogs was introduced in 8.2. This has now been corrected. 8.3 made this worse because the deadlock could occur within a critical code section, making it a PANIC rather than just ERROR condition.

- Fix misbehavior of foreign key checks involving character or bit columns (Tom)
  
  If the referencing column were of a different but compatible type (for instance `varchar`), the constraint was enforced incorrectly.

- Avoid needless deadlock failures in no-op foreign-key checks (Stephan Szabo, Tom)

- Fix possible core dump when re-planning a prepared query (Tom)
  
  This bug affected only protocol-level prepare operations, not SQL `PREPARE`, and so tended to be seen only with JDBC, DBI, and other client-side drivers that use prepared statements heavily.

- Fix possible failure when re-planning a query that calls an SPI-using function (Tom)

- Fix failure in row-wise comparisons involving columns of different datatypes (Tom)

- Fix longstanding **LISTEN**/**NOTIFY** race condition (Tom)
  
  In rare cases a session that had just executed a **LISTEN** might not get a notification, even though one would be expected because the concurrent transaction executing **NOTIFY** was observed to commit later.

  A side effect of the fix is that a transaction that has executed a not-yet-committed **LISTEN** command will not see any row in `pg_listener` for the **LISTEN**, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

- Disallow **LISTEN** and **UNLISTEN** within a prepared transaction (Tom)
  
  This was formerly allowed but trying to do it had various unpleasant consequences, notably that the originating backend could not exit as long as an **UNLISTEN** remained uncommitted.

- Disallow dropping a temporary table within a prepared transaction (Heikki)
  
  This was correctly disallowed by 8.1, but the check was inadvertently broken in 8.2 and 8.3.
Appendix E. Release Notes

• Fix rare crash when an error occurs during a query using a hash index (Heikki)
• Fix incorrect comparison of tsquery values (Teodor)
• Fix incorrect behavior of LIKE with non-ASCII characters in single-byte encodings (Rolf Jentsch)
• Disable xmlvalidate (Tom)
  This function should have been removed before 8.3 release, but was inadvertently left in the source code. It poses a small security risk since unprivileged users could use it to read the first few characters of any file accessible to the server.
• Fix memory leaks in certain usages of set-returning functions (Neil)
• Make encode(bytea, 'escape') convert all high-bit-set byte values into \nnn octal escape sequences (Tom)
  This is necessary to avoid encoding problems when the database encoding is multi-byte. This change could pose compatibility issues for applications that are expecting specific results from encode.
• Fix input of datetime values for February 29 in years BC (Tom)
  The former coding was mistaken about which years were leap years.
• Fix “unrecognized node type” error in some variants of ALTER OWNER (Tom)
• Avoid tablespace permissions errors in CREATE TABLE LIKE INCLUDING INDEXES (Tom)
• Ensure pg_stat_activity.waiting flag is cleared when a lock wait is aborted (Tom)
• Fix handling of process permissions on Windows Vista (Dave, Magnus)
  In particular, this fix allows starting the server as the Administrator user.
• Update time zone data files to tzdata release 2008a (in particular, recent Chile changes); adjust timezone abbreviation VET (Venezuela) to mean UTC-4:30, not UTC-4:00 (Tom)
• Fix ecpg problems with arrays (Michael)
• Fix pg_ctl to correctly extract the postmaster’s port number from command-line options (Itagaki Takahiro, Tom)
  Previously, pg_ctl start -w could try to contact the postmaster on the wrong port, leading to bogus reports of startup failure.
• Use -fwrapv to defend against possible misoptimization in recent gcc versions (Tom)
  This is known to be necessary when building PostgreSQL with gcc 4.3 or later.
• Enable building contrib/uuid-ossp with MSVC (Hirosi Saito)

E.24. Release 8.3

Release Date: 2008-02-04
E.24.1. Overview

With significant new functionality and performance enhancements, this release represents a major leap forward for PostgreSQL. This was made possible by a growing community that has dramatically accelerated the pace of development. This release adds the following major features:

- Full text search is integrated into the core database system
- Support for the SQL/XML standard, including new operators and an XML data type
- Enumerated data types (ENUM)
- Arrays of composite types
- Universally Unique Identifier (UUID) data type
- Add control over whether NULLs sort first or last
- Updatable cursors
- Server configuration parameters can now be set on a per-function basis
- User-defined types can now have type modifiers
- Automatically re-plan cached queries when table definitions change or statistics are updated
- Numerous improvements in logging and statistics collection
- Support Security Service Provider Interface (SSPI) for authentication on Windows
- Support multiple concurrent autovacuum processes, and other autovacuum improvements
- Allow the whole PostgreSQL distribution to be compiled with Microsoft Visual C++

Major performance improvements are listed below. Most of these enhancements are automatic and do not require user changes or tuning:

- Asynchronous commit delays writes to WAL during transaction commit
- Checkpoint writes can be spread over a longer time period to smooth the I/O spike during each checkpoint
- Heap-Only Tuples (HOT) accelerate space reuse for most UPDATEs and DELETEs
- Just-in-time background writer strategy improves disk write efficiency
- Using non-persistent transaction IDs for read-only transactions reduces overhead and VACUUM requirements
- Per-field and per-row storage overhead has been reduced
- Large sequential scans no longer force out frequently used cached pages
- Concurrent large sequential scans can now share disk reads
- ORDER BY ... LIMIT can be done without sorting

The above items are explained in more detail in the sections below.

E.24.2. Migration to Version 8.3

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:
E.24.2.1. General

- Non-character data types are no longer automatically cast to TEXT (Peter, Tom)

Previously, if a non-character value was supplied to an operator or function that requires text input, it was automatically cast to text, for most (though not all) built-in data types. This no longer happens: an explicit cast to text is now required for all non-character-string types. For example, these expressions formerly worked:

```
substr(current_date, 1, 4)
23 LIKE '2%'
```

but will now draw “function does not exist” and “operator does not exist” errors respectively. Use an explicit cast instead:

```
substr(current_date::text, 1, 4)
23::text LIKE '2%'
```

(Of course, you can use the more verbose CAST() syntax too.) The reason for the change is that these automatic casts too often caused surprising behavior. An example is that in previous releases, this expression was accepted but did not do what was expected:

```
current_date < 2017-11-17
```

This is actually comparing a date to an integer, which should be (and now is) rejected — but in the presence of automatic casts both sides were cast to text and a textual comparison was done, because the text < text operator was able to match the expression when no other < operator could.

Types char(n) and varchar(n) still cast to text automatically. Also, automatic casting to text still works for inputs to the concatenation (||) operator, so long as least one input is a character-string type.

- Full text search features from contrib/tsearch2 have been moved into the core server, with some minor syntax changes

```
contrib/tsearch2 now contains a compatibility interface.
```

- ARRAY(SELECT ...), where the SELECT returns no rows, now returns an empty array, rather than NULL (Tom)

- The array type name for a base data type is no longer always the base type’s name with an underscore prefix

The old naming convention is still honored when possible, but application code should no longer depend on it. Instead use the new pg_type.typarray column to identify the array data type associated with a given type.

- ORDER BY ... USING operator must now use a less-than or greater-than operator that is defined in a btree operator class

This restriction was added to prevent inconsistent results.

- SET LOCAL changes now persist until the end of the outermost transaction, unless rolled back (Tom)

Previously SET LOCAL’s effects were lost after subtransaction commit (RELEASE SAVEPOINT or exit from a PL/pgSQL exception block).

- Commands rejected in transaction blocks are now also rejected in multiple-statement query strings (Tom)

For example, "BEGIN; DROP DATABASE; COMMIT" will now be rejected even if submitted as a single query message.
• **ROLLBACK** outside a transaction block now issues **NOTICE** instead of **WARNING** (Bruce)

• Prevent **NOTIFY**/LISTEN/UNLISTEN from accepting schema-qualified names (Bruce)

  Formerly, these commands accepted `schema.relation` but ignored the schema part, which was confusing.

• **ALTER SEQUENCE** no longer affects the sequence’s `currval()` state (Tom)

• Foreign keys now must match indexable conditions for cross-data-type references (Tom)

  This improves semantic consistency and helps avoid performance problems.

• Restrict object size functions to users who have reasonable permissions to view such information (Tom)

  For example, `pg_database_size()` now requires CONNECT permission, which is granted to everyone by default. `pg_tablespace_size()` requires CREATE permission in the tablespace, or is allowed if the tablespace is the default tablespace for the database.

• Remove the undocumented `!!= (not in)` operator (Tom)

  `NOT IN (SELECT ...)` is the proper way to perform this operation.

• Internal hashing functions are now more uniformly-distributed (Tom)

  If application code was computing and storing hash values using internal PostgreSQL hashing functions, the hash values must be regenerated.

• C-code conventions for handling variable-length data values have changed (Greg Stark, Tom)

  The new `SET_VARSIZE()` macro must be used to set the length of generated `varlena` values.

  Also, it might be necessary to expand (“de-TOAST”) input values in more cases.

• Continuous archiving no longer reports each successful archive operation to the server logs unless `DEBUG` level is used (Simon)

---

**E.24.2.2. Configuration Parameters**

• Numerous changes in administrative server parameters

  `bgwriter_lru_percent`, `bgwriter_all_percent`, `bgwriter_all_maxpages`, `stats_start_collector`, and `stats_reset_on_server_start` are removed. `redirect_stderr` is renamed to `logging_collector`. `stats_command_string` is renamed to `track_activities`. `stats_block_level` and `stats_row_level` are merged into `track_counts`. A new boolean configuration parameter, `archive_mode`, controls archiving. Autovacuum’s default settings have changed.

• Remove `stats_start_collector` parameter (Tom)

  We now always start the collector process, unless UDP socket creation fails.

• Remove `stats_reset_on_server_start` parameter (Tom)

  This was removed because `pg_stat_reset()` can be used for this purpose.

• Commenting out a parameter in `postgresql.conf` now causes it to revert to its default value (Joachim Wieland)

  Previously, commenting out an entry left the parameter’s value unchanged until the next server restart.
E.24.2.3. Character Encodings

- Add more checks for invalidly-encoded data (Andrew)
  This change plugs some holes that existed in literal backslash escape string processing and \COPY escape processing. Now the de-escaped string is rechecked to see if the result created an invalid multi-byte character.
- Disallow database encodings that are inconsistent with the server’s locale setting (Tom)
  On most platforms, \c locale is the only locale that will work with any database encoding. Other locale settings imply a specific encoding and will misbehave if the database encoding is something different. (Typical symptoms include bogus textual sort order and wrong results from \texttt{upper()} or \texttt{lower()}.) The server now rejects attempts to create databases that have an incompatible encoding.
- Ensure that \texttt{chr()} cannot create invalidly-encoded values (Andrew)
  In UTF8-encoded databases the argument of \texttt{chr()} is now treated as a Unicode code point. In other multi-byte encodings \texttt{chr()}’s argument must designate a 7-bit ASCII character. Zero is no longer accepted. \texttt{ascii()} has been adjusted to match.
- Adjust \texttt{convert()} behavior to ensure encoding validity (Andrew)
  The two argument form of \texttt{convert()} has been removed. The three argument form now takes a \texttt{bytea} first argument and returns a \texttt{bytea}. To cover the loss of functionality, three new functions have been added:
  - \texttt{convert_from(bytea, name)} returns \texttt{text} — converts the first argument from the named encoding to the database encoding
  - \texttt{convert_to(text, name)} returns \texttt{bytea} — converts the first argument from the database encoding to the named encoding
  - \texttt{length(bytea, name)} returns \texttt{integer} — gives the length of the first argument in characters in the named encoding
- Remove \texttt{convert(argument USING conversion_name)} (Andrew)
  Its behavior did not match the SQL standard.
- Make JOHAB encoding client-only (Tatsuo)
  JOHAB is not safe as a server-side encoding.

E.24.3. Changes

Below you will find a detailed account of the changes between PostgreSQL 8.3 and the previous major release.

E.24.3.1. Performance

- Asynchronous commit delays writes to WAL during transaction commit (Simon)
  This feature dramatically increases performance for short data-modifying transactions. The disadvantage is that because disk writes are delayed, if the database or operating system crashes before data is written to the disk, committed data will be lost. This feature is useful for applications that can
Appendix E. Release Notes

accept some data loss. Unlike turning off *fsync*, using asynchronous commit does not put database consistency at risk; the worst case is that after a crash the last few reportedly-committed transactions might not be committed after all. This feature is enabled by turning off synchronous_commit (which can be done per-session or per-transaction, if some transactions are critical and others are not). *wal_writer_delay* can be adjusted to control the maximum delay before transactions actually reach disk.

- Checkpoint writes can be spread over a longer time period to smooth the I/O spike during each checkpoint (Itagaki Takahiro and Heikki Linnakangas)

  Previously all modified buffers were forced to disk as quickly as possible during a checkpoint, causing an I/O spike that decreased server performance. This new approach spreads out disk writes during checkpoints, reducing peak I/O usage. (User-requested and shutdown checkpoints are still written as quickly as possible.)

- Heap-Only Tuples (HOT) accelerate space reuse for most updates and deletes (Pavan Deolasee, with ideas from many others)

  Updates and deletes leave dead tuples behind, as do failed inserts. Previously only vacuum could reclaim space taken by dead tuples. With HOT dead tuple space can be automatically reclaimed at the time of insert or update if no changes are made to indexed columns. This allows for more consistent performance. Also, HOT avoids adding duplicate index entries.

- Just-in-time background writer strategy improves disk write efficiency (Greg Smith, Itagaki Takahiro)

  This greatly reduces the need for manual tuning of the background writer.

- Per-field and per-row storage overhead have been reduced (Greg Stark, Heikki Linnakangas)

  Variable-length data types with data values less than 128 bytes long will see a storage decrease of 3 to 6 bytes. For example, two adjacent char(1) fields now use 4 bytes instead of 16. Row headers are also 4 bytes shorter than before.

- Using non-persistent transaction IDs for read-only transactions reduces overhead and vacuum requirements (Florian Pflug)

  Non-persistent transaction IDs do not increment the global transaction counter. Therefore, they reduce the load on *pg_clog* and increase the time between forced vacuum to prevent transaction ID wraparound. Other performance improvements were also made that should improve concurrency.

- Avoid incrementing the command counter after a read-only command (Tom)

  There was formerly a hard limit of $2^{32}$ (4 billion) commands per transaction. Now only commands that actually changed the database count, so while this limit still exists, it should be significantly less annoying.

- Create a dedicated WAL writer process to off-load work from backends (Simon)

- Skip unnecessary WAL writes for cluster and copy (Simon)

  Unless WAL archiving is enabled, the system now avoids WAL writes for cluster and just *fsync()*s the table at the end of the command. It also does the same for copy if the table was created in the same transaction.

- Large sequential scans no longer force out frequently used cached pages (Simon, Heikki, Tom)

- Concurrent large sequential scans can now share disk reads (Jeff Davis)

  This is accomplished by starting the new sequential scan in the middle of the table (where another sequential scan is already in-progress) and wrapping around to the beginning to finish. This can affect the order of returned rows in a query that does not specify *order by*. The synchronize_seqscans configuration parameter can be used to disable this if necessary.
Appendix E. Release Notes

• ORDER BY ... LIMIT can be done without sorting (Greg Stark)
  This is done by sequentially scanning the table and tracking just the “top N” candidate rows, rather
  than performing a full sort of the entire table. This is useful when there is no matching index and
  the LIMIT is not large.
• Put a rate limit on messages sent to the statistics collector by backends (Tom)
  This reduces overhead for short transactions, but might sometimes increase the delay before statistics
  are tallied.
• Improve hash join performance for cases with many NULLs (Tom)
• Speed up operator lookup for cases with non-exact datatype matches (Tom)

E.24.3.2. Server

• Autovacuum is now enabled by default (Alvaro)
  Several changes were made to eliminate disadvantages of having autovacuum enabled, thereby
  justifying the change in default. Several other autovacuum parameter defaults were also modified.
• Support multiple concurrent autovacuum processes (Alvaro, Itagaki Takahiro)
  This allows multiple vacuums to run concurrently. This prevents vacuuming of a large table from
  delaying vacuuming of smaller tables.
• Automatically re-plan cached queries when table definitions change or statistics are updated (Tom)
  Previously PL/pgSQL functions that referenced temporary tables would fail if the temporary table
  was dropped and recreated between function invocations, unless EXECUTE was used. This improve-
  ment fixes that problem and many related issues.
• Add a temp_tablespaces parameter to control the tablespaces for temporary tables and files
  (Jaime Casanova, Albert Cervera, Bernd Helmle)
  This parameter defines a list of tablespaces to be used. This enables spreading the I/O load across
  multiple tablespaces. A random tablespace is chosen each time a temporary object is created. Tem-
  porary files are no longer stored in per-database pgsql_tmp/ directories but in per-tablespace
  directories.
• Place temporary tables’ TOAST tables in special schemas named pg_toast_temp_nnn (Tom)
  This allows low-level code to recognize these tables as temporary, which enables various optimiza-
  tions such as not WAL-logging changes and using local rather than shared buffers for access. This
  also fixes a bug wherein backends unexpectedly held open file references to temporary TOAST
  tables.
• Fix problem that a constant flow of new connection requests could indefinitely delay the postmaster
  from completing a shutdown or a crash restart (Tom)
• Guard against a very-low-probability data loss scenario by preventing re-use of a deleted table’s
  relfilenode until after the next checkpoint (Heikki)
• Fix CREATE CONSTRAINT TRIGGER to convert old-style foreign key trigger definitions into regu-
  lar foreign key constraints (Tom)
  This will ease porting of foreign key constraints carried forward from pre-7.3 databases, if they
  were never converted using contrib/adddepend.
• Fix DEFAULT NULL to override inherited defaults (Tom)
Appendix E. Release Notes

DEFAULT NULL was formerly considered a noise phrase, but it should (and now does) override non-null defaults that would otherwise be inherited from a parent table or domain.

- Add new encodings EUC_JIS_2004 and SHIFT_JIS_2004 (Tatsuo)
  These new encodings can be converted to and from UTF-8.
- Change server startup log message from “database system is ready” to “database system is ready to accept connections”, and adjust its timing.
  The message now appears only when the postmaster is really ready to accept connections.

E.24.3.3. Monitoring

- Add \texttt{log\_autovacuum\_min\_duration} parameter to support configurable logging of autovacuum activity (Simon, Alvaro)
- Add \texttt{log\_lock\_waits} parameter to log lock waiting (Simon)
- Add \texttt{log\_temp\_files} parameter to log temporary file usage (Bill Moran)
- Add \texttt{log\_checkpoints} parameter to improve logging of checkpoints (Greg Smith, Heikki)
- \texttt{log\_line\_prefix} now supports \texttt{%s} and \texttt{%c} escapes in all processes (Andrew)
  Previously these escapes worked only for user sessions, not for background database processes.
- Add \texttt{log\_restartpoints} to control logging of point-in-time recovery restart points (Simon)
- Last transaction end time is now logged at end of recovery and at each logged restart point (Simon)
- Autovacuum now reports its activity start time in \texttt{pg\_stat\_activity} (Tom)
- Allow server log output in comma-separated value (CSV) format (Arul Shaji, Greg Smith, Andrew Dunstan)
  CSV-format log files can easily be loaded into a database table for subsequent analysis.
- Use PostgreSQL-supplied timezone support for formatting timestamps displayed in the server log (Tom)
  This avoids Windows-specific problems with localized time zone names that are in the wrong encoding. There is a new \texttt{log\_timezone} parameter that controls the timezone used in log messages, independently of the client-visible \texttt{timezone} parameter.
- New system view \texttt{pg\_stat\_bgwriter} displays statistics about background writer activity (Magnus)
- Add new columns for database-wide tuple statistics to \texttt{pg\_stat\_database} (Magnus)
- Add \texttt{xact\_start} (transaction start time) column to \texttt{pg\_stat\_activity} (Neil)
  This makes it easier to identify long-running transactions.
- Add \texttt{n\_live\_tuples} and \texttt{n\_dead\_tuples} columns to \texttt{pg\_stat\_all\_tables} and related views (Glen Parker)
- Merge \texttt{stats\_block\_level} and \texttt{stats\_row\_level} parameters into a single parameter \texttt{track\_counts}, which controls all messages sent to the statistics collector process (Tom)
- Rename \texttt{stats\_command\_string} parameter to \texttt{track\_activities} (Tom)
- Fix statistical counting of live and dead tuples to recognize that committed and aborted transactions have different effects (Tom)
E.24.3.4. Authentication

- Support Security Service Provider Interface (SSPI) for authentication on Windows (Magnus)
- Support GSSAPI authentication (Henry Hotz, Magnus)
  This should be preferred to native Kerberos authentication because GSSAPI is an industry standard.
- Support a global SSL configuration file (Victor Wagner)
- Add `ssl_ciphers` parameter to control accepted SSL ciphers (Victor Wagner)
- Add a Kerberos realm parameter, `krb_realm` (Magnus)

E.24.3.5. Write-Ahead Log (WAL) and Continuous Archiving

- Change the timestamps recorded in transaction WAL records from `time_t` to `TimestampTz` representation (Tom)
  This provides sub-second resolution in WAL, which can be useful for point-in-time recovery.
- Reduce WAL disk space needed by warm standby servers (Simon)
  This change allows a warm standby server to pass the name of the earliest still-needed WAL file to the recovery script, allowing automatic removal of no-longer-needed WAL files. This is done using `%r` in the `restore_command` parameter of `recovery.conf`.
- New boolean configuration parameter, `archive_mode`, controls archiving (Simon)
  Previously setting `archive_command` to an empty string turned off archiving. Now `archive_mode` turns archiving on and off, independently of `archive_command`. This is useful for stopping archiving temporarily.

E.24.3.6. Queries

- Full text search is integrated into the core database system (Teodor, Oleg)
  Text search has been improved, moved into the core code, and is now installed by default. `contrib/tsearch2` now contains a compatibility interface.
- Add control over whether ` NULLs` sort first or last (Teodor, Tom)
  The syntax is `ORDER BY ... NULLS FIRST/LAST`.
- Allow per-column ascending/descending (`ASC/DESC`) ordering options for indexes (Teodor, Tom)
  Previously a query using `ORDER BY` with mixed `ASC/DESC` specifiers could not fully use an index. Now an index can be fully used in such cases if the index was created with matching `ASC/DESC` specifications. `NULL` sort order within an index can be controlled, too.
- Allow `col IS NULL` to use an index (Teodor)
- Updatable cursors (Arul Shaji, Tom)
  This eliminates the need to reference a primary key to `UPDATE` or `DELETE` rows returned by a cursor. The syntax is `UPDATE/DELETE WHERE CURRENT OF`.
- Allow `FOR UPDATE` in cursors (Arul Shaji, Tom)
• Create a general mechanism that supports casts to and from the standard string types (TEXT, VARCHAR, CHAR) for every datatype, by invoking the datatype’s I/O functions (Tom)

Previously, such casts were available only for types that had specialized function(s) for the purpose. These new casts are assignment-only in the to-string direction, explicit-only in the other direction, and therefore should create no surprising behavior.

• Allow UNION and related constructs to return a domain type, when all inputs are of that domain type (Tom)

Formerly, the output would be considered to be of the domain’s base type.

• Allow limited hashing when using two different data types (Tom)

This allows hash joins, hash indexes, hashed subplans, and hash aggregation to be used in situations involving cross-data-type comparisons, if the data types have compatible hash functions. Currently, cross-data-type hashing support exists for smallint/integer/bigint, and for float4/float8.

• Improve optimizer logic for detecting when variables are equal in a WHERE clause (Tom)

This allows mergejoins to work with descending sort orders, and improves recognition of redundant sort columns.

• Improve performance when planning large inheritance trees in cases where most tables are excluded by constraints (Tom)

E.24.3.7. Object Manipulation

• Arrays of composite types (David Fetter, Andrew, Tom)

In addition to arrays of explicitly-declared composite types, arrays of the rowtypes of regular tables and views are now supported, except for rowtypes of system catalogs, sequences, and TOAST tables.

• Server configuration parameters can now be set on a per-function basis (Tom)

For example, functions can now set their own search_path to prevent unexpected behavior if a different search_path exists at run-time. Security definer functions should set search_path to avoid security loopholes.

• CREATE/ALTER FUNCTION now supports COST and ROWS options (Tom)

COST allows specification of the cost of a function call. ROWS allows specification of the average number or rows returned by a set-returning function. These values are used by the optimizer in choosing the best plan.

• Implement CREATE TABLE LIKE ... INCLUDING INDEXES (Trevor Hardcastle, Nikhil Sontakke, Neil)

• Allow CREATE INDEX CONCURRENTLY to ignore transactions in other databases (Simon)

• Add ALTER VIEW ... RENAME TO and ALTER SEQUENCE ... RENAME TO (David Fetter, Neil)

Previously this could only be done via ALTER TABLE ... RENAME TO.

• Make CREATE/DROP/RENAME DATABASE wait briefly for conflicting backends to exit before failing (Tom)

This increases the likelihood that these commands will succeed.
Appendix E. Release Notes

• Allow triggers and rules to be deactivated in groups using a configuration parameter, for replication purposes (Jan)

This allows replication systems to disable triggers and rewrite rules as a group without modifying the system catalogs directly. The behavior is controlled by `ALTER TABLE` and a new parameter `session_replication_role`.

• User-defined types can now have type modifiers (Teodor, Tom)

This allows a user-defined type to take a modifier, like `ssnum(7)`. Previously only built-in data types could have modifiers.

E.24.3.8. Utility Commands

• Non-superuser database owners now are able to add trusted procedural languages to their databases by default (Jeremy Drake)

While this is reasonably safe, some administrators might wish to revoke the privilege. It is controlled by `pg_pltemplate tmpldbacreate`.

• Allow a session’s current parameter setting to be used as the default for future sessions (Tom)

This is done with `SET ... FROM CURRENT` in `CREATE/ALTER FUNCTION, ALTER DATABASE, or ALTER ROLE`.

• Implement new commands `DISCARD ALL`, `DISCARD PLANS`, `DISCARD TEMPORARY`, `CLOSE ALL`, and `DEALLOCATE ALL` (Marko Kreen, Neil)

These commands simplify resetting a database session to its initial state, and are particularly useful for connection-pooling software.

• Make `CLUSTER` MVCC-safe (Heikki Linnakangas)

Formerly, `CLUSTER` would discard all tuples that were committed dead, even if there were still transactions that should be able to see them under MVCC visibility rules.

• Add new `CLUSTER` syntax: `CLUSTER table USING index` (Holger Schurig)

The old `CLUSTER` syntax is still supported, but the new form is considered more logical.

• Fix `EXPLAIN` so it can show complex plans more accurately (Tom)

References to subplan outputs are now always shown correctly, instead of using `?columnN?` for complicated cases.

• Limit the amount of information reported when a user is dropped (Alvaro)

Previously, dropping (or attempting to drop) a user who owned many objects could result in large `NOTICE` or `ERROR` messages listing all these objects; this caused problems for some client applications. The length of the message is now limited, although a full list is still sent to the server log.

E.24.3.9. Data Types

• Support for the SQL/XML standard, including new operators and an `XML` data type (Nikolay Samokhvalov, Pavel Stehule, Peter)

• Enumerated data types (`ENUM`) (Tom Dunstan)
This feature provides convenient support for fields that have a small, fixed set of allowed values. An example of creating an `ENUM` type is `CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy')`.

- Universally Unique Identifier (UUID) data type (Gevik Babakhani, Neil)
  This closely matches RFC 4122.
- Widen the `MONEY` data type to 64 bits (D’Arcy Cain)
  This greatly increases the range of supported `MONEY` values.
- Fix `float4`/`float8` to handle `Infinity` and `NAN` (Not A Number) consistently (Bruce)
  The code formerly was not consistent about distinguishing `Infinity` from overflow conditions.
- Allow leading and trailing whitespace during input of `boolean` values (Neil)
- Prevent `COPY` from using digits and lowercase letters as delimiters (Tom)

**E.24.3.10. Functions**

- Add new regular expression functions `regexp_matches()`, `regexp_split_to_array()`, and `regexp_split_to_table()` (Jeremy Drake, Neil)
  These functions provide extraction of regular expression subexpressions and allow splitting a string using a POSIX regular expression.
- Add `lo_truncate()` for large object truncation (Kris Jurka)
- Implement `width_bucket()` for the `float8` data type (Neil)
- Add `pg_stat_clear_snapshot()` to discard statistics snapshots collected during the current transaction (Tom)
  The first request for statistics in a transaction takes a statistics snapshot that does not change during the transaction. This function allows the snapshot to be discarded and a new snapshot loaded during the next statistics query. This is particularly useful for PL/pgSQL functions, which are confined to a single transaction.
- Add `isodow` option to `EXTRACT()` and `date_part()` (Bruce)
  This returns the day of the week, with Sunday as seven. (`dow` returns Sunday as zero.)
- Add `ID` (ISO day of week) and `IDDD` (ISO day of year) format codes for `to_char()`, `to_date()`, and `to_timestamp()` (Brendan Jurd)
- Make `to_timestamp()` and `to_date()` assume TM (trim) option for potentially variable-width fields (Bruce)
  This matches Oracle’s behavior.
- Fix off-by-one conversion error in `to_date()`/`to_timestamp()` D (non-ISO day of week) fields (Bruce)
- Make `setseed()` return void, rather than a useless integer value (Neil)
- Add a hash function for `NUMERIC` (Neil)
  This allows hash indexes and hash-based plans to be used with `NUMERIC` columns.
- Improve efficiency of `LIKE/ILIKE`, especially for multi-byte character sets like UTF-8 (Andrew, Itagaki Takahiro)
- Make `currtid()` functions require `SELECT` privileges on the target table (Tom)
• Add several \texttt{txid\_\_\_}() functions to query active transaction IDs (Jan)
  This is useful for various replication solutions.

\textbf{E.24.3.11. PL/pgSQL Server-Side Language}

• Add scrollable cursor support, including directional control in \texttt{FETCH} (Pavel Stehule)
• Allow \texttt{IN} as an alternative to \texttt{FROM} in PL/pgSQL's \texttt{FETCH} statement, for consistency with the backend's \texttt{FETCH} command (Pavel Stehule)
• Add \texttt{MOVE} to PL/pgSQL (Magnus, Pavel Stehule, Neil)
• Implement \texttt{RETURN QUERY} (Pavel Stehule, Neil)
  This adds convenient syntax for PL/pgSQL set-returning functions that want to return the result of a query. \texttt{RETURN QUERY} is easier and more efficient than a loop around \texttt{RETURN NEXT}.
• Allow function parameter names to be qualified with the function's name (Tom)
  For example, \texttt{myfunc.myvar}. This is particularly useful for specifying variables in a query where the variable name might match a column name.
• Make qualification of variables with block labels work properly (Tom)
  Formerly, outer-level block labels could unexpectedly interfere with recognition of inner-level record or row references.
• Tighten requirements for \texttt{FOR} loop \texttt{STEP} values (Tom)
  Prevent non-positive \texttt{STEP} values, and handle loop overflows.
• Improve accuracy when reporting syntax error locations (Tom)

\textbf{E.24.3.12. Other Server-Side Languages}

• Allow type-name arguments to PL/Perl \texttt{spi\_\_prepare()} to be data type aliases in addition to names found in \texttt{pg\_\_type} (Andrew)
• Allow type-name arguments to PL/Python \texttt{plpy\_\_prepare()} to be data type aliases in addition to names found in \texttt{pg\_\_type} (Andrew)
• Allow type-name arguments to PL/Tcl \texttt{spi\_\_prepare} to be data type aliases in addition to names found in \texttt{pg\_\_type} (Andrew)
• Enable PL/PythonU to compile on Python 2.5 (Marko Kreen)
• Support a true PL/Python boolean type in compatible Python versions (Python 2.3 and later) (Marko Kreen)
• Fix PL/Tcl problems with thread-enabled \texttt{libtcl} spawning multiple threads within the backend (Steve Marshall, Paul Bayer, Doug Knight)
  This caused all sorts of unpleasantness.

\textbf{E.24.3.13. psql}

• List disabled triggers separately in \texttt{\_\_d output} (Brendan Jurd)
Appendix E. Release Notes

- In \d patterns, always match $ literally (Tom)
- Show aggregate return types in \da output (Greg Sabino Mullane)
- Add the function’s volatility status to the output of \df+ (Neil)
- Add \prompt capability (Chad Wagner)
- Allow \pset, \t, and \x to specify on or off, rather than just toggling (Chad Wagner)
- Add \sleep capability (Jan)
- Enable \timing output for \copy (Andrew)
- Improve \timing resolution on Windows (Itagaki Takahiro)
- Flush \o output after each backslash command (Tom)
- Correctly detect and report errors while reading a -f input file (Peter)
- Remove -u option (this option has long been deprecated) (Tom)

E.24.3.14. pg_dump

- Add --tablespaces-only and --roles-only options to pg_dumpall (Dave Page)
- Add an output file option to pg_dumpall (Dave Page)
  This is primarily useful on Windows, where output redirection of child pg_dump processes does not work.
- Allow pg_dumpall to accept an initial-connection database name rather than the default
  template1 (Dave Page)
- In -n and -t switches, always match $ literally (Tom)
- Improve performance when a database has thousands of objects (Tom)
- Remove -u option (this option has long been deprecated) (Tom)

E.24.3.15. Other Client Applications

- In initdb, allow the location of the pg_xlog directory to be specified (Euler Taveira de Oliveira)
- Enable server core dump generation in pg_regress on supported operating systems (Andrew)
- Add a -t (timeout) parameter to pg_ctl (Bruce)
  This controls how long pg_ctl will wait when waiting for server startup or shutdown. Formerly the
  timeout was hard-wired as 60 seconds.
- Add a pg_ctl option to control generation of server core dumps (Andrew)
- Allow Control-C to cancel clusterdb, reindexdb, and vacuumdb (Itagaki Takahiro, Magnus)
- Suppress command tag output for createdb, createuser, droppedb, and dropuser (Peter)
  The --quiet option is ignored and will be removed in 8.4. Progress messages when acting on all
  databases now go to stdout instead of stderr because they are not actually errors.
E.24.3.16. libpq

- Interpret the dbName parameter of PQsetdbLogin() as a conninfo string if it contains an equals sign (Andrew)
  This allows use of conninfo strings in client programs that still use PQsetdbLogin().
- Support a global SSL configuration file (Victor Wagner)
- Add environment variable PGSSLKEY to control SSL hardware keys (Victor Wagner)
- Add lo_truncate() for large object truncation (Kris Jurka)
- Add PQconnectionNeedsPassword() that returns true if the server required a password but none was supplied (Joe Conway, Tom)
  If this returns true after a failed connection attempt, a client application should prompt the user for a password. In the past applications have had to check for a specific error message string to decide whether a password is needed; that approach is now deprecated.
- Add PQconnectionUsedPassword() that returns true if the supplied password was actually used (Joe Conway, Tom)
  This is useful in some security contexts where it is important to know whether a user-supplied password is actually valid.

E.24.3.17. ecpg

- Use V3 frontend/backend protocol (Michael)
  This adds support for server-side prepared statements.
- Use native threads, instead of pthreads, on Windows (Magnus)
- Improve thread-safety of ecpglib (Itagaki Takahiro)
- Make the ecpg libraries export only necessary API symbols (Michael)

E.24.3.18. Windows Port

- Allow the whole PostgreSQL distribution to be compiled with Microsoft Visual C++ (Magnus and others)
  This allows Windows-based developers to use familiar development and debugging tools. Windows executables made with Visual C++ might also have better stability and performance than those made with other tool sets. The client-only Visual C++ build scripts have been removed.
- Drastically reduce postmaster’s memory usage when it has many child processes (Magnus)
- Allow regression tests to be started by an administrative user (Magnus)
- Add native shared memory implementation (Magnus)

E.24.3.19. Server Programming Interface (SPI)

- Add cursor-related functionality in SPI (Pavel Stehule)
Appendix E. Release Notes

Allow access to the cursor-related planning options, and add `FETCH/MOVE` routines.

- Allow execution of cursor commands through `SPI_execute` (Tom)
  The macro `SPI_ERROR_CURSOR` still exists but will never be returned.
- SPI plan pointers are now declared as `SPIPlanPtr` instead of `void *` (Tom)
  This does not break application code, but switching is recommended to help catch simple programming mistakes.

E.24.3.20. Build Options

- Add configure option `--enable-profiling` to enable code profiling (works only with gcc) (Korry Douglas and Nikhil Sontakke)
- Add configure option `--with-system-tzdata` to use the operating system’s time zone database (Peter)
- Fix PGXS so extensions can be built against PostgreSQL installations whose `pg_config` program does not appear first in the `PATH` (Tom)
- Support `gmake draft` when building the SGML documentation (Bruce)
  Unless `draft` is used, the documentation build will now be repeated if necessary to ensure the index is up-to-date.

E.24.3.21. Source Code

- Rename macro `DLLIMPORT` to `PGDLLIMPORT` to avoid conflicting with third party includes (like Tcl) that define `DLLIMPORT` (Magnus)
- Create “operator families” to improve planning of queries involving cross-data-type comparisons (Tom)
- Update GIN `extractQuery()` API to allow signalling that nothing can satisfy the query (Teodor)
- Move `NAMEDATALEN` definition from `postgres_ext.h` to `pg_config_manual.h` (Peter)
- Provide `strlcpy()` and `strlcat()` on all platforms, and replace error-prone uses of `strncpy()`, `strncat()`, etc (Peter)
- Create hooks to let an external plugin monitor (or even replace) the planner and create plans for hypothetical situations (Gurjeet Singh, Tom)
- Create a function variable `join_search_hook` to let plugins override the join search order portion of the planner (Julius Stroffek)
- Add `tas()` support for Renesas’ M32R processor (Kazuhiro Inaoka)
- `quote_identifier()` and `pg_dump` no longer quote keywords that are unreserved according to the grammar (Tom)
- Change the on-disk representation of the `NUMERIC` data type so that the `sign_dscale` word comes before the weight (Tom)
- Use SYSV semaphores rather than POSIX on Darwin >= 6.0, i.e., OS X 10.2 and up (Chris Marcellino)
- Add acronym and NFS documentation sections (Bruce)
Appendix E. Release Notes

- "Postgres" is now documented as an accepted alias for "PostgreSQL" (Peter)
- Add documentation about preventing database server spoofing when the server is down (Bruce)

E.24.3.22. Contrib

- Move contrib README content into the main PostgreSQL documentation (Albert Cervera i Areny)
- Add contrib/pageinspect module for low-level page inspection (Simon, Heikki)
- Add contrib/pg_standby module for controlling warm standby operation (Simon)
- Add contrib/uuid-osspp module for generating UUID values using the OSSP UUID library (Peter)
  
  Use configure --with-osspp-uuid to activate. This takes advantage of the new UUID builtin type.
- Add contrib/dict_int, contrib/dict_xsyn, and contrib/test_parser modules to provide sample add-on text search dictionary templates and parsers (Sergey Karpov)
- Allow contrib/pgbench to set the fillfactor (Pavan Deolasee)
- Add timestamps to contrib/pgbench -l (Greg Smith)
- Add usage count statistics to contrib/pgbuffercache (Greg Smith)
- Add GIN support for contrib/hstore (Teodor)
- Add GIN support for contrib/pg_trgm (Guillaume Smet, Teodor)
- Update OS/X startup scripts in contrib/start-scripts (Mark Cotner, David Fetter)
- Restrict pgrowlocks() and dblink_get_pkey() to users who have SELECT privilege on the target table (Tom)
- Restrict contrib/pgstattuple functions to superusers (Tom)
- contrib/xml2 is deprecated and planned for removal in 8.4 (Peter)

  The new XML support in core PostgreSQL supersedes this module.

E.25. Release 8.2.23

Release Date: 2011-12-05

This release contains a variety of fixes from 8.2.22. For information about new features in the 8.2 major release, see Section E.48.

This is expected to be the last PostgreSQL release in the 8.2.X series. Users are encouraged to update to a newer release branch soon.

E.25.1. Migration to Version 8.2.23

A dump/restore is not required for those running 8.2.X.
However, a longstanding error was discovered in the definition of the information_schema.referential_constraints view. If you rely on correct results from that view, you should replace its definition as explained in the first changelog item below.

Also, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

E.25.2. Changes

- Fix bugs in information_schema.referential_constraints view (Tom Lane)
  
  This view was being insufficiently careful about matching the foreign-key constraint to the depended-on primary or unique key constraint. That could result in failure to show a foreign key constraint at all, or showing it multiple times, or claiming that it depends on a different constraint than the one it really does.

  Since the view definition is installed by initdb, merely upgrading will not fix the problem. If you need to fix this in an existing installation, you can (as a superuser) drop the information_schema schema then re-create it by sourcing $SHAREDIR/information_schema.sql. (Run pg_config --sharedir if you’re uncertain where $SHAREDIR is.) This must be repeated in each database to be fixed.

- Fix TOAST-related data corruption during CREATE TABLE dest AS SELECT * FROM src or INSERT INTO dest SELECT * FROM src (Tom Lane)

  If a table has been modified by ALTER TABLE ADD COLUMN, attempts to copy its data verbatim to another table could produce corrupt results in certain corner cases. The problem can only manifest in this precise form in 8.4 and later, but we patched earlier versions as well in case there are other code paths that could trigger the same bug.

- Fix race condition during toast table access from stale syscache entries (Tom Lane)

  The typical symptom was transient errors like "missing chunk number 0 for toast value NNNNN in pg_toast_2619", where the cited toast table would always belong to a system catalog.

- Improve locale support in money type’s input and output (Tom Lane)

  Aside from not supporting all standard lc_monetary formatting options, the input and output functions were inconsistent, meaning there were locales in which dumped money values could not be re-read.

- Don’t let transform_null_equals affect CASE foo WHEN NULL ... constructs (Heikki Lin-nakangas)

  transform_null_equals is only supposed to affect foo = NULL expressions written directly by the user, not equality checks generated internally by this form of CASE.

- Change foreign-key trigger creation order to better support self-referential foreign keys (Tom Lane)

  For a cascading foreign key that references its own table, a row update will fire both the ON UPDATE trigger and the CHECK trigger as one event. The ON UPDATE trigger must execute first, else the CHECK will check a non-final state of the row and possibly throw an inappropriate error. However, the firing order of these triggers is determined by their names, which generally sort in creation order since the triggers have auto-generated names following the convention “RI_ConstrainTrigger_NNNNN”. A proper fix would require modifying that convention, which we will do in 9.2, but it seems risky to change it in existing releases. So this patch just changes the creation order of the triggers. Users encountering this type of error should drop and re-create the foreign key constraint to get its triggers into the right order.
Appendix E. Release Notes

- Preserve blank lines within commands in psql’s command history (Robert Haas)
  The former behavior could cause problems if an empty line was removed from within a string literal, for example.
- Use the preferred version of xsubpp to build PL/Perl, not necessarily the operating system’s main copy (David Wheeler and Alex Hunsaker)
- Honor query cancel interrupts promptly in pgstaindex() (Robert Haas)
- Ensure VPATH builds properly install all server header files (Peter Eisentraut)
- Shorten file names reported in verbose error messages (Peter Eisentraut)
  Regular builds have always reported just the name of the C file containing the error message call, but VPATH builds formerly reported an absolute path name.
- Fix interpretation of Windows timezone names for Central America (Tom Lane)
  Map “Central America Standard Time” to CST6, not CST6CDT, because DST is generally not observed anywhere in Central America.
- Update time zone data files to tzdata release 2011n for DST law changes in Brazil, Cuba, Fiji, Palestine, Russia, and Samoa; also historical corrections for Alaska and British East Africa.

E.26. Release 8.2.22

Release Date: 2011-09-26

This release contains a variety of fixes from 8.2.21. For information about new features in the 8.2 major release, see Section E.48.

The PostgreSQL community will stop releasing updates for the 8.2.X release series in December 2011. Users are encouraged to update to a newer release branch soon.

E.26.1. Migration to Version 8.2.22

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

E.26.2. Changes

- Fix multiple bugs in GiST index page split processing (Heikki Linnakangas)
  The probability of occurrence was low, but these could lead to index corruption.
- Avoid possibly accessing off the end of memory in ANALYZE (Noah Misch)
  This fixes a very-low-probability server crash scenario.
- Fix race condition in relcache init file invalidation (Tom Lane)
Appendix E. Release Notes

There was a window wherein a new backend process could read a stale init file but miss the inval messages that would tell it the data is stale. The result would be bizarre failures in catalog accesses, typically “could not read block 0 in file ...” later during startup.

- Fix memory leak at end of a GiST index scan (Tom Lane)

Commands that perform many separate GiST index scans, such as verification of a new GiST-based exclusion constraint on a table already containing many rows, could transiently require large amounts of memory due to this leak.

- Fix performance problem when constructing a large, lossy bitmap (Tom Lane)

This avoids some situations where the planner will think that semantically-equal constants are not equal, resulting in poor optimization.

- Work around gcc 4.6.0 bug that breaks WAL replay (Tom Lane)

This could lead to loss of committed transactions after a server crash.

- Fix dump bug for VALUES in a view (Tom Lane)

- Disallow SELECT FOR UPDATE/SHARE on sequences (Tom Lane)

This operation doesn’t work as expected and can lead to failures.

- Defend against integer overflow when computing size of a hash table (Tom Lane)

- Fix portability bugs in use of credentials control messages for “peer” authentication (Tom Lane)

- Fix typo in pg_srand48 seed initialization (Andres Freund)

This led to failure to use all bits of the provided seed. This function is not used on most platforms (only those without srand), and the potential security exposure from a less-random-than-expected seed seems minimal in any case.

- Avoid integer overflow when the sum of LIMIT and OFFSET values exceeds 2^63 (Heikki Linnakangas)

- Add overflow checks to int4 and int8 versions of generate_series() (Robert Haas)

- Fix trailing-zero removal in to_char() (Marti Raudsepp)

In a format with FM and no digit positions after the decimal point, zeroes to the left of the decimal point could be removed incorrectly.

- Fix pg_size_pretty() to avoid overflow for inputs close to 2^63 (Tom Lane)

- Fix psql’s counting of script file line numbers during COPY from a different file (Tom Lane)

- Fix pg_restore’s direct-to-database mode for standard_conforming_strings (Tom Lane)

pg_restore could emit incorrect commands when restoring directly to a database server from an archive file that had been made with standard_conforming_strings set to on.

- Fix write-past-buffer-end and memory leak in libpq’s LDAP service lookup code (Albe Laurenz)

- In libpq, avoid failures when using nonblocking I/O and an SSL connection (Martin Pihlak, Tom Lane)

- Improve libpq’s handling of failures during connection startup (Tom Lane)

In particular, the response to a server report of fork() failure during SSL connection startup is now saner.

- Make ecpglib write double values with 15 digits precision (Akira Kurosawa)

- Apply upstream fix for blowfish signed-character bug (CVE-2011-2483) (Tom Lane)
Appendix E. Release Notes

contrib/pg_crypto's blowfish encryption code could give wrong results on platforms where char is signed (which is most), leading to encrypted passwords being weaker than they should be.

- Fix memory leak in contrib/seg (Heikki Linnakangas)
- Fix pgstatindex() to give consistent results for empty indexes (Tom Lane)
- Allow building with perl 5.14 (Alex Hunsaker)
- Update configure script's method for probing existence of system functions (Tom Lane)
  The version of autoconf we used in 8.3 and 8.2 could be fooled by compilers that perform link-time optimization.
- Fix assorted issues with build and install file paths containing spaces (Tom Lane)
- Update time zone data files to tzdata release 2011i for DST law changes in Canada, Egypt, Russia, Samoa, and South Sudan.

E.27. Release 8.2.21

Release Date: 2011-04-18

This release contains a variety of fixes from 8.2.20. For information about new features in the 8.2 major release, see Section E.48.

E.27.1. Migration to Version 8.2.21

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

E.27.2. Changes

- Avoid potential deadlock during catalog cache initialization (Nikhil Sontakke)
  In some cases the cache loading code would acquire share lock on a system index before locking the index's catalog. This could deadlock against processes trying to acquire exclusive locks in the other, more standard order.
- Fix dangling-pointer problem in BEFORE ROW UPDATE trigger handling when there was a concurrent update to the target tuple (Tom Lane)
  This bug has been observed to result in intermittent “cannot extract system attribute from virtual tuple” failures while trying to do UPDATE RETURNING ctid. There is a very small probability of more serious errors, such as generating incorrect index entries for the updated tuple.
- Disallow DROP TABLE when there are pending deferred trigger events for the table (Tom Lane)
  Formerly the DROP would go through, leading to “could not open relation with OID nnn” errors when the triggers were eventually fired.
- Fix PL/Python memory leak involving array slices (Daniel Popowich)
• Fix pg_restore to cope with long lines (over 1KB) in TOC files (Tom Lane)
• Put in more safeguards against crashing due to division-by-zero with overly enthusiastic compiler optimization (Aurelien Jarno)
• Support use of dlopen() in FreeBSD and OpenBSD on MIPS (Tom Lane)
  There was a hard-wired assumption that this system function was not available on MIPS hardware on these systems. Use a compile-time test instead, since more recent versions have it.
• Fix compilation failures on HP-UX (Heikki Linnakangas)
• Fix path separator used by pg_regress on Cygwin (Andrew Dunstan)
• Update time zone data files to tzdata release 2011f for DST law changes in Chile, Cuba, Falkland Islands, Morocco, Samoa, and Turkey; also historical corrections for South Australia, Alaska, and Hawaii.

E.28. Release 8.2.20

Release Date: 2011-01-31

This release contains a variety of fixes from 8.2.19. For information about new features in the 8.2 major release, see Section E.48.

E.28.1. Migration to Version 8.2.20

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

E.28.2. Changes

• Avoid failures when EXPLAIN tries to display a simple-form CASE expression (Tom Lane)
  If the CASE’s test expression was a constant, the planner could simplify the CASE into a form that confused the expression-display code, resulting in “unexpected CASE WHEN clause” errors.
• Fix assignment to an array slice that is before the existing range of subscripts (Tom Lane)
  If there was a gap between the newly added subscripts and the first pre-existing subscript, the code miscalculated how many entries needed to be copied from the old array’s null bitmap, potentially leading to data corruption or crash.
• Avoid unexpected conversion overflow in planner for very distant date values (Tom Lane)
  The date type supports a wider range of dates than can be represented by the timestamp types, but the planner assumed it could always convert a date to timestamp with impunity.
• Fix pg_restore’s text output for large objects (BLOBs) when standard_conforming_strings is on (Tom Lane)
Although restoring directly to a database worked correctly, string escaping was incorrect if pg_restore was asked for SQL text output and standard_conforming_strings had been enabled in the source database.

- Fix erroneous parsing of tsquery values containing ... & !(subexpression) | ... (Tom Lane)

Queries containing this combination of operators were not executed correctly. The same error existed in contrib/intarray's query_int type and contrib/ltree's ltxquery type.

- Fix buffer overrun in contrib/intarray's input function for the query_int type (Apple)

This bug is a security risk since the function's return address could be overwritten. Thanks to Apple Inc's security team for reporting this issue and supplying the fix. (CVE-2010-4015)

- Fix bug in contrib/seg's GiST picksplit algorithm (Alexander Korotkov)

This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a seg column. If you have such an index, consider REINDEXing it after installing this update. (This is identical to the bug that was fixed in contrib/cube in the previous update.)

---

**E.29. Release 8.2.19**

**Release Date:** 2010-12-16

This release contains a variety of fixes from 8.2.18. For information about new features in the 8.2 major release, see Section E.48.

**E.29.1. Migration to Version 8.2.19**

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

**E.29.2. Changes**

- **Force the default wal_sync_method to be fdatasync on Linux** (Tom Lane, Marti Raudsepp)

  The default on Linux has actually been fdatasync for many years, but recent kernel changes caused PostgreSQL to choose open_datasync instead. This choice did not result in any performance improvement, and caused outright failures on certain filesystems, notably ext4 with the data=journal mount option.

- **Fix assorted bugs in WAL replay logic for GIN indexes** (Tom Lane)

  This could result in "bad buffer id: 0" failures or corruption of index contents during replication.

- **Fix recovery from base backup when the starting checkpoint WAL record is not in the same WAL segment as its redo point** (Jeff Davis)

- **Add support for detecting register-stack overrun on IA64** (Tom Lane)
Appendix E. Release Notes

The IA64 architecture has two hardware stacks. Full prevention of stack-overrun failures requires checking both.

- Add a check for stack overflow in `copyObject()` (Tom Lane)
  Certain code paths could crash due to stack overflow given a sufficiently complex query.

- Fix detection of page splits in temporary GiST indexes (Heikki Linnakangas)
  It is possible to have a “concurrent” page split in a temporary index, if for example there is an open cursor scanning the index when an insertion is done. GiST failed to detect this case and hence could deliver wrong results when execution of the cursor continued.

- Avoid memory leakage while `ANALYZE`'ing complex index expressions (Tom Lane)

- Ensure an index that uses a whole-row Var still depends on its table (Tom Lane)
  An index declared like `create index i on t (foo(t.*))` would not automatically get dropped when its table was dropped.

- Do not “inline” a SQL function with multiple `OUT` parameters (Tom Lane)
  This avoids a possible crash due to loss of information about the expected result rowtype.

- Behave correctly if `ORDER BY`, `LIMIT`, `FOR UPDATE`, or `WITH` is attached to the `VALUES` part of `INSERT ... VALUES` (Tom Lane)

- Fix constant-folding of `COALESCE()` expressions (Tom Lane)
  The planner would sometimes attempt to evaluate sub-expressions that in fact could never be reached, possibly leading to unexpected errors.

- Add print functionality for `InhRelation` nodes (Tom Lane)
  This avoids a failure when `debug_print_parse` is enabled and certain types of query are executed.

- Fix incorrect calculation of distance from a point to a horizontal line segment (Tom Lane)
  This bug affected several different geometric distance-measurement operators.

- Fix PL/pgSQL’s handling of “simple” expressions to not fail in recursion or error-recovery cases (Tom Lane)

- Fix PL/Python’s handling of set-returning functions (Jan Urbanski)
  Attempts to call SPI functions within the iterator generating a set result would fail.

- Fix bug in `contrib/cube`’s GiST picksplit algorithm (Alexander Korotkov)
  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a `cube` column. If you have such an index, consider `REINDEX`ing it after installing this update.

- Don’t emit “identifier will be truncated” notices in `contrib/dblink` except when creating new connections (Itagaki Takahiro)

- Fix potential coredump on missing public key in `contrib/pgcrypto` (Marti Raudsepp)

- Fix memory leak in `contrib/xml2`’s XPath query functions (Tom Lane)

- Update time zone data files to `tzdata` release 2010o for DST law changes in Fiji and Samoa; also historical corrections for Hong Kong.
E.30. Release 8.2.18

Release Date: 2010-10-04

This release contains a variety of fixes from 8.2.17. For information about new features in the 8.2 major release, see Section E.48.

E.30.1. Migration to Version 8.2.18

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

E.30.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)
  
  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a SECURITY DEFINER function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function’s owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).

- Prevent possible crashes in pg_get_expr() by disallowing it from being called with an argument that is not one of the system catalog columns it’s intended to be used with (Heikki Linnakangas, Tom Lane)

- Fix Windows shared-memory allocation code (Tsutomu Yamada, Magnus Hagander)

  This bug led to the often-reported “could not reattach to shared memory” error message. This is a back-patch of a fix that was applied to newer branches some time ago.

- Treat exit code 128 (ERROR_WAIT_NO_CHILDREN) as non-fatal on Windows (Magnus Hagander)

  Under high load, Windows processes will sometimes fail at startup with this error code. Formerly the postmaster treated this as a panic condition and restarted the whole database, but that seems to be an overreaction.

- Fix possible duplicate scans of UNION ALL member relations (Tom Lane)

- Fix “cannot handle unplanned sub-select” error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.
Appendix E. Release Notes

- Reduce PANIC to ERROR in some occasionally-reported btree failure cases, and provide additional detail in the resulting error messages (Tom Lane)
  This should improve the system’s robustness with corrupted indexes.
- Prevent show_session_authorization() from crashing within autovacuum processes (Tom Lane)
- Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)
- Fix possible failure when hashing a pass-by-reference function result (Tao Ma, Tom Lane)
- Take care to fsync the contents of lockfiles (both postmaster.pid and the socket lockfile) while writing them (Tom Lane)
  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.
- Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)
  The original coding could result in a crash if there was limited stack space.
- Fix log_line_prefix’s %i escape, which could produce junk early in backend startup (Tom Lane)
- Fix possible data corruption in ALTER TABLE ... SET TABLESPACE when archiving is enabled (Jeff Davis)
- Allow CREATE DATABASE and ALTER DATABASE ... SET TABLESPACE to be interrupted by query-cancel (Guillaume Lelarge)
- In PL/Python, defend against null pointer results from PyObject_AsVoidPtr and PyObject_FromVoidPtr (Peter Eisentraut)
- Improve contrib/dblink’s handling of tables containing dropped columns (Tom Lane)
- Fix connection leak after “duplicate connection name” errors in contrib/dblink (Itagaki Takahiro)
- Fix contrib/dblink to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)
- Add hstore(text, text) function to contrib/hstore (Robert Haas)
  This function is the recommended substitute for the now-deprecated = operator. It was back-patched so that future-proofed code can be used with older server versions. Note that the patch will be effective only after contrib/hstore is installed or reinstalled in a particular database. Users might prefer to execute the CREATE FUNCTION command by hand, instead.
- Update build infrastructure and documentation to reflect the source code repository’s move from CVS to Git (Magnus Hagander and others)
- Update time zone data files to tzdata release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.
  This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.
- Make Windows’ “N. Central Asia Standard Time” timezone map to Asia/Novosibirsk, not Asia/Almaty (Magnus Hagander)
Microsoft changed the DST behavior of this zone in the timezone update from KB976098. Asia/Novosibirsk is a better match to its new behavior.

E.31. Release 8.2.17

Release Date: 2010-05-17

This release contains a variety of fixes from 8.2.16. For information about new features in the 8.2 major release, see Section E.48.

E.31.1. Migration to Version 8.2.17

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

E.31.2. Changes

- Enforce restrictions in plperl using an opmask applied to the whole interpreter, instead of using Safe.pm (Tim Bunce, Andrew Dunstan)

  Recent developments have convinced us that Safe.pm is too insecure to rely on for making plperl trustable. This change removes use of Safe.pm altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl's strict pragma in a natural way in plperl, and that Perl's $a and $b variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

  PL/Tcl's feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted "normal" Tcl interpreter unless we are really going to execute a pltclu function. (CVE-2010-1170)

- Fix possible crash if a cache reset message is received during rebuild of a relcache entry (Heikki)

  This error was introduced in 8.2.16 while fixing a related failure.

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran ALTER USER ... RESET ALL for himself, or ALTER DATABASE ... RESET ALL for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the ALTER will only remove the parameters that the user has permission to change.

- Avoid possible crash during backend shutdown if shutdown occurs when a CONTEXT addition would be made to log entries (Tom)
In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

- Update pl/perl’s ppport.h for modern Perl versions (Andrew)
- Fix assorted memory leaks in pl/python (Andreas Freund, Tom)
- Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)
- Fix psql’s \copy to not add spaces around a dot within \copy (select ...) (Tom)
  Addition of spaces around the decimal point in a numeric literal would result in a syntax error.
- Ensure that contrib/pgstattuple functions respond to cancel interrupts promptly (Tatsuhito Kasahara)
- Make server startup deal properly with the case that shmget() returns EINVAL for an existing shared memory segment (Tom)
  This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.
- Avoid possible crashes in syslogger process on Windows (Heikki)
- Deal more robustly with incomplete time zone information in the Windows registry (Magnus)
- Update the set of known Windows time zone names (Magnus)
- Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.
  Also, add PKST (Pakistan Summer Time) to the default set of timezone abbreviations.

### E.32. Release 8.2.16

**Release Date:** 2010-03-15

This release contains a variety of fixes from 8.2.15. For information about new features in the 8.2 major release, see Section E.48.

#### E.32.1. Migration to Version 8.2.16

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

#### E.32.2. Changes

- Add new configuration parameter ssl_renegotiation_limit to control how often we do session key renegotiation for an SSL connection (Magnus)
This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.

- Fix possible deadlock during backend startup (Tom)
- Fix possible crashes due to not handling errors during relcache reload cleanly (Tom)
- Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)
- Fix server memory leak associated with use of savepoints and a client encoding different from server’s encoding (Tom)
- Fix incorrect WAL data emitted during end-of-recovery cleanup of a GIST index page split (Yoichi Hirai)
  This would result in index corruption, or even more likely an error during WAL replay, if we were unlucky enough to crash during end-of-recovery cleanup after having completed an incomplete GIST insertion.
- Make substring() for bit types treat any negative length as meaning “all the rest of the string” (Tom)
  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).
- Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)
- Fix some cases of pathologically slow regular expression matching (Tom)
- Fix the STOP WAL LOCATION entry in backup history files to report the next WAL segment’s name when the end location is exactly at a segment boundary (Itagaki Takahiro)
- Fix some more cases of temporary-file leakage (Heikki)
  This corrects a problem introduced in the previous minor release. One case that failed is when a plpgsql function returning set is called within another function’s exception handler.
- Improve constraint exclusion processing of boolean-variable cases, in particular make it possible to exclude a partition that has a “bool_column = false” constraint (Tom)
- When reading pg_hba.conf and related files, do not treat @something as a file inclusion request if the @ appears inside quote marks; also, never treat @ by itself as a file inclusion request (Tom)
  This prevents erratic behavior if a role or database name starts with @. If you need to include a file whose path name contains spaces, you can still do so, but you must write @"/path to/file" rather than putting the quotes around the whole construct.
- Prevent infinite loop on some platforms if a directory is named as an inclusion target in pg_hba.conf and related files (Tom)
- Fix possible infinite loop if SSL_read or SSL_write fails without setting errno (Tom)
  This is reportedly possible with some Windows versions of openssl.
- Fix psql’s numericlocale option to not format strings it shouldn’t in latex and troff output formats (Heikki)
- Make psql return the correct exit status (3) when ON_ERROR_STOP and --single-transaction are both specified and an error occurs during the implied COMMIT (Bruce)
- Fix plpgsql failure in one case where a composite column is set to NULL (Tom)
- Fix possible failure when calling PL/Perl functions from PL/PerlU or vice versa (Tim Bunce)
Appendix E. Release Notes

- Add volatile markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)
- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)
  The only known symptom of this oversight is that the Tcl `clock` command misbehaves if using Tcl 8.5 or later.
- Prevent crash in contrib/dblink when too many key columns are specified to a `dblink_build_sql_` function (Rushabh Lathia, Joe Conway)
- Fix assorted crashes in contrib/xml2 caused by sloppy memory management (Tom)
- Make building of contrib/xml2 more robust on Windows (Andrew)
- Fix race condition in Windows signal handling (Radu Ilie)
  One known symptom of this bug is that rows in `pg_listener` could be dropped under heavy load.
- Update time zone data files to tzdata release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

E.33. Release 8.2.15

Release Date: 2009-12-14

This release contains a variety of fixes from 8.2.14. For information about new features in the 8.2 major release, see Section E.48.

E.33.1. Migration to Version 8.2.15

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.14, see the release notes for 8.2.14.

E.33.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)
  This change prevents allegedly-immutable index functions from possibly subverting a superuser’s session (CVE-2009-4136).
- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)
  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).
- Fix possible crash during backend-startup-time cache initialization (Tom)
- Prevent signals from interrupting `VACUUM` at unsafe times (Alvaro)
Appendix E. Release Notes

This fix prevents a PANIC if a VACUUM FULL is canceled after it’s already committed its tuple movements, as well as transient errors if a plain VACUUM is interrupted after having truncated the table.

- Fix possible crash due to integer overflow in hash table size calculation (Tom)
  This could occur with extremely large planner estimates for the size of a hashjoin’s result.
- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
- Ensure that shared tuple-level locks held by prepared transactions are not ignored (Heikki)
- Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction (Heikki)
- Fix incorrect logic for GiST index page splits, when the split depends on a non-first column of the index (Paul Ramsey)
- Don’t error out if recycling or removing an old WAL file fails at the end of checkpoint (Heikki)
  It’s better to treat the problem as non-fatal and allow the checkpoint to complete. Future checkpoints will retry the removal. Such problems are not expected in normal operation, but have been seen to be caused by misdesigned Windows anti-virus and backup software.
- Ensure WAL files aren’t repeatedly archived on Windows (Heikki)
  This is another symptom that could happen if some other process interfered with deletion of a no-longer-needed file.
- Fix PAM password processing to be more robust (Tom)
  The previous code is known to fail with the combination of the Linux pam_krb5 PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.
- Fix processing of ownership dependencies during CREATE OR REPLACE FUNCTION (Tom)
- Fix bug with calling plperl from plperlu or vice versa (Tom)
  An error exit from the inner function could result in crashes due to failure to re-select the correct Perl interpreter for the outer function.
- Fix session-lifespan memory leak when a PL/Perl function is redefined (Tom)
- Ensure that Perl arrays are properly converted to PostgreSQL arrays when returned by a set-returning PL/Perl function (Andrew Dunstan, Abhijit Menon-Sen)
  This worked correctly already for non-set-returning functions.
- Fix rare crash in exception processing in PL/Python (Peter)
- Ensure psql’s flex module is compiled with the correct system header definitions (Tom)
  This fixes build failures on platforms where --enable-largefile causes incompatible changes in the generated code.
- Make the postmaster ignore any application_name parameter in connection request packets, to improve compatibility with future libpq versions (Tom)
- Update the timezone abbreviation files to match current reality (Joachim Wieland)
  This includes adding IDT and SGT to the default timezone abbreviation set.
- Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for Hong Kong.
E.34. Release 8.2.14

Release Date: 2009-09-09

This release contains a variety of fixes from 8.2.13. For information about new features in the 8.2 major release, see Section E.48.

E.34.1. Migration to Version 8.2.14

A dump/restore is not required for those running 8.2.X. However, if you have any hash indexes on interval columns, you must REINDEX them after updating to 8.2.14. Also, if you are upgrading from a version earlier than 8.2.11, see the release notes for 8.2.11.

E.34.2. Changes

- Force WAL segment switch during pg_start_backup() (Heikki)
  This avoids corner cases that could render a base backup unusable.
- Disallow RESET ROLE and RESET SESSION AUTHORIZATION inside security-definer functions (Tom, Heikki)
  This covers a case that was missed in the previous patch that disallowed SET ROLE and SET SESSION AUTHORIZATION inside security-definer functions. (See CVE-2007-6600)
- Make LOAD of an already-loaded loadable module into a no-op (Tom)
  Formerly, LOAD would attempt to unload and re-load the module, but this is unsafe and not all that useful.
- Disallow empty passwords during LDAP authentication (Magnus)
- Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)
- Fix bugs associated with fetching a whole-row value from the output of a Sort or Materialize plan node (Tom)
- Revert planner change that disabled partial-index and constraint exclusion optimizations when there were more than 100 clauses in an AND or OR list (Tom)
- Fix hash calculation for data type interval (Tom)
  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must REINDEX them after updating.
- Treat to_char(..., ‘TH’) as an uppercase ordinal suffix with ‘HH’/’HH12’ (Heikki)
  It was previously handled as ‘th’ (lowercase).
- Fix overflow for INTERVAL ‘x ms’ when x is more than 2 million and integer datetimes are in use (Alex Hunsaker)
- Fix calculation of distance between a point and a line segment (Tom)
  This led to incorrect results from a number of geometric operators.
Appendix E. Release Notes

• Fix money data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)

• Properly round datetime input like 00:12:57.9999999999999999999999999999 (Tom)

• Fix poor choice of page split point in GiST R-tree operator classes (Teodor)

• Avoid performance degradation in bulk inserts into GIN indexes when the input values are (nearly) in sorted order (Tom)

• Correctly enforce NOT NULL domain constraints in some contexts in PL/pgSQL (Tom)

• Fix portability issues in plperl initialization (Andrew Dunstan)

• Fix pg_ctl to not go into an infinite loop if postgresql.conf is empty (Jeff Davis)

• Make contrib/hstore throw an error when a key or value is too long to fit in its data structure, rather than silently truncating it (Andrew Gierth)

• Fix contrib/xml2’s xslt_process() to properly handle the maximum number of parameters (twenty) (Tom)

• Improve robustness of libpq’s code to recover from errors during COPY FROM STDIN (Tom)

• Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)

• Update time zone data files to tzdata release 2009l for DST law changes in Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan (historical correction only), Mauritius, Morocco, Palestine, Syria, Tunisia.

E.35. Release 8.2.13

Release Date: 2009-03-16

This release contains a variety of fixes from 8.2.12. For information about new features in the 8.2 major release, see Section E.48.

E.35.1. Migration to Version 8.2.13

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.11, see the release notes for 8.2.11.

E.35.2. Changes

• Prevent error recursion crashes when encoding conversion fails (Tom)

This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that any error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding
Appendix E. Release Notes

conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)

• Disallow CREATE CONVERSION with the wrong encodings for the specified conversion function (Heikki)
This prevents one possible scenario for encoding conversion failure. The previous change is a back-stop to guard against other kinds of failures in the same area.

• Fix core dump when to_char() is given format codes that are inappropriate for the type of the data argument (Tom)

• Fix possible failure in contrib/tsearch2 when C locale is used with a multi-byte encoding (Teodor)
Crashes were possible on platforms where wchar_t is narrower than int; Windows in particular.

• Fix extreme inefficiency in contrib/tsearch2 parser’s handling of an email-like string containing multiple @ characters (Heikki)

• Fix decompilation of CASE WHEN with an implicit coercion (Tom)
This mistake could lead to Assert failures in an Assert-enabled build, or an “unexpected CASE WHEN clause” error message in other cases, when trying to examine or dump a view.

• Fix possible misassignment of the owner of a TOAST table’s rowtype (Tom)
If CLUSTER or a rewriting variant of ALTER TABLE were executed by someone other than the table owner, the pg_type entry for the table’s TOAST table would end up marked as owned by that someone. This caused no immediate problems, since the permissions on the TOAST rowtype aren’t examined by any ordinary database operation. However, it could lead to unexpected failures if one later tried to drop the role that issued the command (in 8.1 or 8.2), or “owner of data type appears to be invalid” warnings from pg_dump after having done so (in 8.3).

• Fix PL/pgSQL to not treat INTO after INSERT as an INTO-variables clause anywhere in the string, not only at the start; in particular, don’t fail for INSERT INTO within CREATE RULE (Tom)

E.36. Release 8.2.12

Release Date: 2009-02-02

This release contains a variety of fixes from 8.2.11. For information about new features in the 8.2 major release, see Section E.48.
Appendix E. Release Notes

E.36.1. Migration to Version 8.2.12

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.11, see the release notes for 8.2.11.

E.36.2. Changes

- Improve handling of URLs in `headline()` function (Teodor)
- Improve handling of overlength headlines in `headline()` function (Teodor)
- Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)
- Fix possible Assert failure if a statement executed in PL/pgSQL is rewritten into another kind of statement, for example if an `INSERT` is rewritten into an `UPDATE` (Heikki)
- Ensure that a snapshot is available to datatype input functions (Tom)
  This primarily affects domains that are declared with `CHECK` constraints involving user-defined stable or immutable functions. Such functions typically fail if no snapshot has been set.
- Make it safer for SPI-using functions to be used within datatype I/O; in particular, to be used in domain check constraints (Tom)
- Avoid unnecessary locking of small tables in `VACUUM` (Heikki)
- Fix a problem that made `UPDATE RETURNING tableoid` return zero instead of the correct OID (Tom)
- Fix planner misestimation of selectivity when transitive equality is applied to an outer-join clause (Tom)
  This could result in bad plans for queries like ... from a left join b on a.a1 = b.b1 where a.a1 = 42 ...
- Improve optimizer’s handling of long `IN` lists (Tom)
  This change avoids wasting large amounts of time on such lists when constraint exclusion is enabled.
- Ensure that the contents of a holdable cursor don’t depend on the contents of TOAST tables (Tom)
  Previously, large field values in a cursor result might be represented as TOAST pointers, which would fail if the referenced table got dropped before the cursor is read, or if the large value is deleted and then vacuumed away. This cannot happen with an ordinary cursor, but it could with a cursor that is held past its creating transaction.
- Fix memory leak when a set-returning function is terminated without reading its whole result (Tom)
- Fix `contrib/dblink`’s `dblink_get_result(text,bool)` function (Joe)
- Fix possible garbage output from `contrib/sslinfo` functions (Tom)
- Fix configure script to properly report failure when unable to obtain linkage information for PL/Perl (Andrew)
- Make all documentation reference `pgsql-bugs` and/or `pgsql-hackers` as appropriate, instead of the now-decommissioned `pgsql-ports` and `pgsql-patches` mailing lists (Tom)
- Update time zone data files to tzdata release 2009a (for Kathmandu and historical DST corrections in Switzerland, Cuba)
E.37. Release 8.2.11

Release Date: 2008-11-03

This release contains a variety of fixes from 8.2.10. For information about new features in the 8.2 major release, see Section E.48.

E.37.1. Migration to Version 8.2.11

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.7, see the release notes for 8.2.7. Also, if you were running a previous 8.2.X release, it is recommended to REINDEX all GiST indexes after the upgrade.

E.37.2. Changes

- Fix GiST index corruption due to marking the wrong index entry “dead” after a deletion (Teodor)
  This would result in index searches failing to find rows they should have found. Corrupted indexes can be fixed with REINDEX.
- Fix backend crash when the client encoding cannot represent a localized error message (Tom)
  We have addressed similar issues before, but it would still fail if the “character has no equivalent” message itself couldn’t be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.
- Fix possible crash when deeply nested functions are invoked from a trigger (Tom)
- Improve optimization of expression IN (expression-list) queries (Tom, per an idea from Robert Haas)
  Cases in which there are query variables on the right-hand side had been handled less efficiently in 8.2.x and 8.3.x than in prior versions. The fix restores 8.1 behavior for such cases.
- Fix mis-expansion of rule queries when a sub-SELECT appears in a function call in FROM, a multi-row VALUES list, or a RETURNING list (Tom)
  The usual symptom of this problem is an “unrecognized node type” error.
- Fix memory leak during rescan of a hashed aggregation plan (Neil)
- Ensure an error is reported when a newly-defined PL/pgSQL trigger function is invoked as a normal function (Tom)
- Prevent possible collision of reffilenode numbers when moving a table to another tablespace with ALTER SET TABLESPACE (Heikki)
  The command tried to re-use the existing filename, instead of picking one that is known unused in the destination directory.
- Fix incorrect tsearch2 headline generation when single query item matches first word of text (Sushant Sinha)
- Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an --enable-integer-datetimes build (Ron Mayer)
Appendix E. Release Notes

- Ensure SPI_getvalue and SPI_getbinval behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)

  This situation is normal when a table has had columns added or removed, but these two functions didn’t handle it properly. The only likely consequence is an incorrect error indication.

- Fix ecpg’s parsing of CREATE ROLE (Michael)

- Fix recent breakage of pg_ctl restart (Tom)

- Ensure pg_control is opened in binary mode (Itagaki Takahiro)

  pg_controldata and pg_resetxlog did this incorrectly, and so could fail on Windows.

- Update time zone data files to tzdata release 2008i (for DST law changes in Argentina, Brazil, Mauritius, Syria)

**E.38. Release 8.2.10**

**Release Date:** 2008-09-22

This release contains a variety of fixes from 8.2.9. For information about new features in the 8.2 major release, see Section E.48.

**E.38.1. Migration to Version 8.2.10**

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.7, see the release notes for 8.2.7.

**E.38.2. Changes**

- Fix bug in btree WAL recovery code (Heikki)

  Recovery failed if the WAL ended partway through a page split operation.

- Fix potential miscalculation of datfrozenxid (Alvaro)

  This error may explain some recent reports of failure to remove old pg_clog data.

- Widen local lock counters from 32 to 64 bits (Tom)

  This responds to reports that the counters could overflow in sufficiently long transactions, leading to unexpected “lock is already held” errors.

- Fix possible duplicate output of tuples during a GiST index scan (Teodor)

- Fix missed permissions checks when a view contains a simple UNION ALL construct (Heikki)

  Permissions for the referenced tables were checked properly, but not permissions for the view itself.

- Add checks in executor startup to ensure that the tuples produced by an INSERT or UPDATE will match the target table’s current rowtype (Tom)

  ALTER COLUMN TYPE, followed by re-use of a previously cached plan, could produce this type of situation. The check protects against data corruption and/or crashes that could ensue.
Appendix E. Release Notes

• Fix possible repeated drops during DROP OWNED (Tom)

This would typically result in strange errors such as "cache lookup failed for relation NNN".

• Fix AT TIME ZONE to first try to interpret its timezone argument as a timezone abbreviation, and only try it as a full timezone name if that fails, rather than the other way around as formerly (Tom)

The timestamp input functions have always resolved ambiguous zone names in this order. Making AT TIME ZONE do so as well improves consistency, and fixes a compatibility bug introduced in 8.1: in ambiguous cases we now behave the same as 8.0 and before did, since in the older versions AT TIME ZONE accepted only abbreviations.

• Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)

• Prevent integer overflows during units conversion when displaying a configuration parameter that has units (Tom)

• Improve performance of writing very long log messages to syslog (Tom)

• Allow spaces in the suffix part of an LDAP URL in pg_hba.conf (Tom)

• Fix bug in backwards scanning of a cursor on a SELECT DISTINCT ON query (Tom)

• Fix planner bug with nested sub-select expressions (Tom)

If the outer sub-select has no direct dependency on the parent query, but the inner one does, the outer value might not get recalculated for new parent query rows.

• Fix planner to estimate that GROUP BY expressions yielding boolean results always result in two groups, regardless of the expressions’ contents (Tom)

This is very substantially more accurate than the regular GROUP BY estimate for certain boolean tests like col IS NULL.

• Fix PL/pgSQL to not fail when a FOR loop’s target variable is a record containing composite-type fields (Tom)

• Fix PL/Tcl to behave correctly with Tcl 8.5, and to be more careful about the encoding of data sent to or from Tcl (Tom)

• On Windows, work around a Microsoft bug by preventing libpq from trying to send more than 64kB per system call (Magnus)

• Improve pg_dump and pg_restore’s error reporting after failure to send a SQL command (Tom)

• Fix pg_ctl to properly preserve postmaster command-line arguments across a restart (Bruce)

• Update time zone data files to tzdata release 2008f (for DST law changes in Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine, and Paraguay)

E.39. Release 8.2.9

Release Date: 2008-06-12

This release contains one serious and one minor bug fix over 8.2.8. For information about new features in the 8.2 major release, see Section E.48.
E.39.1. Migration to Version 8.2.9

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.7, see the release notes for 8.2.7.

E.39.2. Changes

• Make `pg_get_ruledef()` parenthesize negative constants (Tom)

  Before this fix, a negative constant in a view or rule might be dumped as, say, `-42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL to reject what had been a valid `SELECT DISTINCT` view query. Since this could result in pg_dump output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.

• Make `ALTER AGGREGATE ... OWNER TO` update `pg_shdepend` (Tom)

  This oversight could lead to problems if the aggregate was later involved in a `DROP OWNED` or `REASSIGN OWNED` operation.

E.40. Release 8.2.8

  Release Date: never released

This release contains a variety of fixes from 8.2.7. For information about new features in the 8.2 major release, see Section E.48.

E.40.1. Migration to Version 8.2.8

A dump/restore is not required for those running 8.2.X. However, if you are upgrading from a version earlier than 8.2.7, see the release notes for 8.2.7.

E.40.2. Changes

• Fix `ERRORDATA_STACK_SIZE exceeded` crash that occurred on Windows when using UTF-8 database encoding and a different client encoding (Tom)

• Fix `ALTER TABLE ADD COLUMN ... PRIMARY KEY` so that the new column is correctly checked to see if it’s been initialized to all non-nulls (Brendan Jurd)

  Previous versions neglected to check this requirement at all.

• Fix possible `CREATE TABLE` failure when inheriting the “same” constraint from multiple parent relations that inherited that constraint from a common ancestor (Tom)
Appendix E. Release Notes

- Fix pg_get_ruledef() to show the alias, if any, attached to the target table of an UPDATE or DELETE (Tom)
- Fix GIN bug that could result in a too many LWLocks taken failure (Teodor)
- Avoid possible crash when decompressing corrupted data (Zdenek Kotala)
- Repair two places where SIGTERM exit of a backend could leave corrupted state in shared memory (Tom)

Neither case is very important if SIGTERM is used to shut down the whole database cluster together, but there was a problem if someone tried to SIGTERM individual backends.

- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic “Yo” characters (e and E with two dots) (Sergey Burladyan)
- Fix several datatype input functions, notably array_in(), that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)

This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched ORDER BY and DISTINCT expressions.

- Fix a corner case in regular-expression substring matching (substring(string from pattern)) (Tom)

The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn’t got a match. An example is substring(‘foo’ from ‘foo(bar)?’). This should return NULL, since (bar) isn’t matched, but it was mistakenly returning the whole-pattern match instead (ie, foo).

- Update time zone data files to tzdata release 2008c (for DST law changes in Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, and Argentina/San_Luis)
- Fix incorrect result from ecpg’s PGTYPEStimestamp_sub() function (Michael)
- Fix broken GiST comparison function for contrib/tsearch2’s tsquery type (Teodor)
- Fix core dump in contrib/cube functions (Tom)
- Fix core dump in contrib/xml2’s xpath_table() function when the input query returns a NULL value (Tom)
- Fix contrib/xml2’s makefile to not override CFLAGS (Tom)
- Fix DatumGetBool macro to not fail with gcc 4.3 (Tom)

This problem affects “old style” (V0) C functions that return boolean. The fix is already in 8.3, but the need to back-patch it was not realized at the time.

E.41. Release 8.2.7

Release Date: 2008-03-17

This release contains a variety of fixes from 8.2.6. For information about new features in the 8.2 major release, see Section E.48.
Appendix E. Release Notes

E.41.1. Migration to Version 8.2.7

A dump/restore is not required for those running 8.2.X. However, you might need to `reindex` indexes on textual columns after updating, if you are affected by the Windows locale issue described below.

E.41.2. Changes

- Fix character string comparison for Windows locales that consider different character combinations as equal (Tom)
  
  This fix applies only on Windows and only when using UTF-8 database encoding. The same fix was made for all other cases over two years ago, but Windows with UTF-8 uses a separate code path that was not updated. If you are using a locale that considers some non-identical strings as equal, you may need to `reindex` to fix existing indexes on textual columns.

- Repair potential deadlock between concurrent `vacuum full` operations on different system catalogs (Tom)

- Fix longstanding `listen/notify` race condition (Tom)
  
  In rare cases a session that had just executed a `listen` might not get a notification, even though one would be expected because the concurrent transaction executing `notify` was observed to commit later.

  A side effect of the fix is that a transaction that has executed a not-yet-committed `listen` command will not see any row in `pg_listener` for the `listen`, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

- Disallow `listen` and `unlisten` within a prepared transaction (Tom)

  This was formerly allowed but trying to do it had various unpleasant consequences, notably that the originating backend could not exit as long as an `unlisten` remained uncommitted.

- Disallow dropping a temporary table within a prepared transaction (Heikki)

  This was correctly disallowed by 8.1, but the check was inadvertently broken in 8.2.

- Fix rare crash when an error occurs during a query using a hash index (Heikki)

- Fix memory leaks in certain usages of set-returning functions (Neil)

- Fix input of datetime values for February 29 in years BC (Tom)

  The former coding was mistaken about which years were leap years.

- Fix “unrecognized node type” error in some variants of `alter owner` (Tom)

- Ensure `pg_stat_activity.waiting` flag is cleared when a lock wait is aborted (Tom)

- Fix handling of process permissions on Windows Vista (Dave, Magnus)

  In particular, this fix allows starting the server as the Administrator user.

- Update time zone data files to `tzdata` release 2008a (in particular, recent Chile changes); adjust timezone abbreviation `VET` (Venezuela) to mean UTC-4:30, not UTC-4:00 (Tom)

- Fix `pg_ctl` to correctly extract the postmaster’s port number from command-line options (Itagaki Takahiro, Tom)

  Previously, `pg_ctl start -w` could try to contact the postmaster on the wrong port, leading to bogus reports of startup failure.
Use `-fwrapv` to defend against possible misoptimization in recent gcc versions (Tom)

This is known to be necessary when building PostgreSQL with gcc 4.3 or later.

Correctly enforce `statement_timeout` values longer than `INT_MAX` microseconds (about 35 minutes) (Tom)

This bug affects only builds with `--enable-integer-datetimes`.

Fix "unexpected PARAM_SUBLINK ID" planner error when constant-folding simplifies a sub-select (Tom)

Fix logical errors in constraint-exclusion handling of `IS NULL` and `NOT` expressions (Tom)

The planner would sometimes exclude partitions that should not have been excluded because of the possibility of NULL results.

Fix another cause of “failed to build any N-way joins” planner errors (Tom)

This could happen in cases where a clauseless join needed to be forced before a join clause could be exploited.

Fix incorrect constant propagation in outer-join planning (Tom)

The planner could sometimes incorrectly conclude that a variable could be constrained to be equal to a constant, leading to wrong query results.

Fix display of constant expressions in `ORDER BY` and `GROUP BY` (Tom)

An explicitly casted constant would be shown incorrectly. This could for example lead to corruption of a view definition during dump and reload.

Fix `libpq` to handle NOTICE messages correctly during COPY OUT (Tom)

This failure has only been observed to occur when a user-defined datatype’s output routine issues a NOTICE, but there is no guarantee it couldn’t happen due to other causes.

E.42. Release 8.2.6

Release Date: 2008-01-07

This release contains a variety of fixes from 8.2.5, including fixes for significant security issues. For information about new features in the 8.2 major release, see Section E.48.

E.42.1. Migration to Version 8.2.6

A dump/restore is not required for those running 8.2.X.

E.42.2. Changes

Prevent functions in indexes from executing with the privileges of the user running `VACUUM`, `ANALYZE`, etc (Tom)
Appendix E. Release Notes

Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as VACUUM FULL, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including VACUUM, ANALYZE, REINDEX, and CLUSTER) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for SECURITY DEFINER functions. To prevent bypassing this security measure, execution of SET SESSION AUTHORIZATION and SET ROLE is now forbidden within a SECURITY DEFINER context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)
  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)
  The fix that appeared for this in 8.2.5 was incomplete, as it plugged the hole for only some dblink functions. (CVE-2007-6601, CVE-2007-3278)

- Fix bugs in WAL replay for GIN indexes (Teodor)

- Fix GIN index build to work properly when maintenance_work_mem is 4GB or more (Tom)

- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)

- Improve planner’s handling of LIKE/regex estimation in non-C locales (Tom)

- Fix planning-speed problem for deep outer-join nests, as well as possible poor choice of join order (Tom)

- Fix planner failure in some cases of WHERE false AND var IN (SELECT ...) (Tom)

- Make CREATE TABLE ... SERIAL and ALTER SEQUENCE ... OWNED BY not change the currval() state of the sequence (Tom)

- Preserve the tablespace and storage parameters of indexes that are rebuilt by ALTER TABLE ... ALTER COLUMN TYPE (Tom)

- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop time was used (Simon)
  This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL segment, and seems simpler and cleaner than the original definition.

- Make VACUUM not use all of maintenance_work_mem when the table is too small for it to be useful (Alvaro)

- Fix potential crash in translate() when using a multibyte database encoding (Tom)

- Make corr() return the correct result for negative correlation values (Neil)

- Fix overflow in extract(epoch from interval) for intervals exceeding 68 years (Tom)

- Fix PL/Perl to not fail when a UTF-8 regular expression is used in a trusted function (Andrew)

- Fix PL/Perl to cope when platform’s Perl defines type bool as int rather than char (Tom)
Appendix E. Release Notes

While this could theoretically happen anywhere, no standard build of Perl did things this way ... until Mac OS X 10.5.

- Fix PL/Python to work correctly with Python 2.5 on 64-bit machines (Marko Kreen)
- Fix PL/Python to not crash on long exception messages (Alvaro)
- Fix pg_dump to correctly handle inheritance child tables that have default expressions different from their parent's (Tom)
- Fix libpq crash when PGPASSFILE refers to a file that is not a plain file (Martin Pitt)
- ecpg parser fixes (Michael)
- Make contrib/pgcrypto defend against OpenSSL libraries that fail on keys longer than 128 bits; which is the case at least on some Solaris versions (Marko Kreen)
- Make contrib/tablefunc's crosstab() handle NULL rowid as a category in its own right, rather than crashing (Joe)
- Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)
- Fix crash of to_tsvector() on huge input strings (Teodor)
- Require a specific version of Autoconf to be used when re-generating the configure script (Peter)

This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it’s your responsibility whether the result works or not.

- Update gettimeofday configuration check so that PostgreSQL can be built on newer versions of MinGW (Magnus)

E.43. Release 8.2.5

Release Date: 2007-09-17

This release contains a variety of fixes from 8.2.4. For information about new features in the 8.2 major release, see Section E.48.

E.43.1. Migration to Version 8.2.5

A dump/restore is not required for those running 8.2.X.

E.43.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent VACUUM on the same table (Tom)
- Fix ALTER DOMAIN ADD CONSTRAINT for cases involving domains over domains (Tom)
- Make CREATE DOMAIN ... DEFAULT NULL work properly (Tom)
• Fix some planner problems with outer joins, notably poor size estimation for \( t1 \text{ LEFT JOIN } t2 \) WHERE \( t2.\text{col} \text{ IS NULL} \) (Tom)

• Allow the interval data type to accept input consisting only of milliseconds or microseconds (Neil)

• Allow timezone name to appear before the year in timestamp input (Tom)

• Fixes for GIN indexes used by /contrib/tsearch2 (Teodor)

• Speed up rtree index insertion (Teodor)

• Fix excessive logging of SSL error messages (Tom)

• Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)

• Fix crash when log_min_error_statement logging runs out of memory (Tom)

• Fix incorrect handling of some foreign-key corner cases (Tom)

• Fix stddev_pop(numeric) and var_pop(numeric) (Tom)

• Prevent REINDEX and CLUSTER from failing due to attempting to process temporary tables of other sessions (Alvaro)

• Update the time zone database rules, particularly New Zealand’s upcoming changes (Tom)

• Windows socket and semaphore improvements (Magnus)

• Make pg_ctl -w work properly in Windows service mode (Dave Page)

• Fix memory allocation bug when using MIT Kerberos on Windows (Magnus)

• Suppress timezone name (\%Z) in log timestamps on Windows because of possible encoding mismatches (Tom)

• Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

• Restrict /contrib/pgstattuple functions to superusers, for security reasons (Tom)

• Do not let /contrib/intarray try to make its GIN opclass the default (this caused problems at dump/restore) (Tom)

E.44. Release 8.2.4

Release Date: 2007-04-23

This release contains a variety of fixes from 8.2.3, including a security fix. For information about new features in the 8.2 major release, see Section E.48.

E.44.1. Migration to Version 8.2.4

A dump/restore is not required for those running 8.2.X.
E.44.2. Changes

- Support explicit placement of the temporary-table schema within search_path, and disable searching it for functions and operators (Tom)
  This is needed to allow a security-definer function to set a truly secure value of search_path. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See CREATE FUNCTION for more information.
- Fix shared_preload_libraries for Windows by forcing reload in each backend (Korry Douglas)
- Fix to_char() so it properly upper/lower cases localized day or month names (Pavel Stehule)
- /contrib/tsearch2 crash fixes (Teodor)
- Require COMMIT PREPARED to be executed in the same database as the transaction was prepared in (Heikki)
- Allow pg_dump to do binary backups larger than two gigabytes on Windows (Magnus)
- New traditional (Taiwan) Chinese FAQ (Zhou Daojing)
- Prevent the statistics collector from writing to disk too frequently (Tom)
- Fix potential-data-corruption bug in how VACUUM FULL handles UPDATE chains (Tom, Pavan Deolasee)
- Fix bug in domains that use array types (Tom)
- Fix pg_dump so it can dump a serial column’s sequence using -t when not also dumping the owning table (Tom)
- Planner fixes, including improving outer join and bitmap scan selection logic (Tom)
- Fix possible wrong answers or crash when a PL/pgSQL function tries to RETURN from within an EXCEPTION block (Tom)
- Fix PANIC during enlargement of a hash index (Tom)
- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)

E.45. Release 8.2.3

Release Date: 2007-02-07

This release contains two fixes from 8.2.2. For information about new features in the 8.2 major release, see Section E.48.

E.45.1. Migration to Version 8.2.3

A dump/restore is not required for those running 8.2.X.
E.45.2. Changes

- Remove overly-restrictive check for type length in constraints and functional indexes (Tom)
- Fix optimization so MIN/MAX in subqueries can again use indexes (Tom)

E.46. Release 8.2.2

Release Date: 2007-02-05

This release contains a variety of fixes from 8.2.1, including a security fix. For information about new features in the 8.2 major release, see Section E.48.

E.46.1. Migration to Version 8.2.2

A dump/restore is not required for those running 8.2.X.

E.46.2. Changes

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)
  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it’s declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix not-so-rare-anymore bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Fix Borland C compile scripts (L Bayuk)
- Properly handle `to_char('CC')` for years ending in 00 (Tom)
  Year 2000 is in the twentieth century, not the twenty-first.
- /contrib/tsearch2 localization improvements (Tatsuo, Teodor)
- Fix incorrect permission check in `information_schema.key_column_usage` view (Tom)
  The symptom is “relation with OID nnnnn does not exist” errors. To get this fix without using `initdb`, use `CREATE OR REPLACE VIEW` to install the corrected definition found in `share/information_schema.sql`. Note you will need to do this in each database.
- Improve `VACUUM` performance for databases with many tables (Tom)
- Fix for rare Assert() crash triggered by `UNION` (Tom)
- Fix potentially incorrect results from index searches using `ROW` inequality conditions (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)
Appendix E. Release Notes

- Fix bogus “permission denied” failures occurring on Windows due to attempts to fsync already-deleted files (Magnus, Tom)
- Fix bug that could cause the statistics collector to hang on Windows (Magnus)
  This would in turn lead to autovacuum not working.
- Fix possible crashes when an already-in-use PL/pgSQL function is updated (Tom)
- Improve PL/pgSQL handling of domain types (Sergiy Vyshnevetskiy, Tom)
- Fix possible errors in processing PL/pgSQL exception blocks (Tom)

E.47. Release 8.2.1

Release Date: 2007-01-08

This release contains a variety of fixes from 8.2. For information about new features in the 8.2 major release, see Section E.48.

E.47.1. Migration to Version 8.2.1

A dump/restore is not required for those running 8.2.

E.47.2. Changes

- Fix crash with SELECT ... LIMIT ALL (also LIMIT NULL) (Tom)
- Several /contrib/tsearch2 fixes (Teodor)
- On Windows, make log messages coming from the operating system use ASCII encoding (Hiroshi Saito)
  This fixes a conversion problem when there is a mismatch between the encoding of the operating system and database server.
- Fix Windows linking of pg_dump using win32.mak (Hiroshi Saito)
- Fix planner mistakes for outer join queries (Tom)
- Fix several problems in queries involving sub-SELECTs (Tom)
- Fix potential crash in SPI during subtransaction abort (Tom)
  This affects all PL functions since they all use SPI.
- Improve build speed of PDF documentation (Peter)
- Re-add JST (Japan) timezone abbreviation (Tom)
- Improve optimization decisions related to index scans (Tom)
- Have psql print multi-byte combining characters as before, rather than output as \u (Tom)
- Improve index usage of regular expressions that use parentheses (Tom)
This improves psql \d performance also.

- Make pg_dumpall assume that databases have public CONNECT privilege, when dumping from a pre-8.2 server (Tom)

This preserves the previous behavior that anyone can connect to a database if allowed by pg_hba.conf.

**E.48. Release 8.2**

**Release Date:** 2006-12-05

**E.48.1. Overview**

This release adds many functionality and performance improvements that were requested by users, including:

- Query language enhancements including *INSERT/UPDATE/DELETE RETURNING, multirow VALUES* lists, and optional target-table alias in *UPDATE/DELETE*
- Index creation without blocking concurrent *INSERT/UPDATE/DELETE operations*
- Many query optimization improvements, including support for reordering outer joins
- Improved sorting performance with lower memory usage
- More efficient locking with better concurrency
- More efficient vacuuming
- Easier administration of warm standby servers
- New *FILLFACTOR* support for tables and indexes
- Monitoring, logging, and performance tuning additions
- More control over creating and dropping objects
- Table inheritance relationships can be defined for and removed from pre-existing tables
- *COPY TO* can copy the output of an arbitrary *SELECT* statement
- Array improvements, including nulls in arrays
- Aggregate-function improvements, including multiple-input aggregates and SQL:2003 statistical functions
- Many *contrib/* improvements

**E.48.2. Migration to Version 8.2**

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.
Appendix E. Release Notes

Observe the following incompatibilities:

• Set \texttt{escape\_string\_warning} to \texttt{on} by default (Bruce)
  This issues a warning if backslash escapes are used in non-escape (non-\texttt{-E}) strings.

• Change the row constructor syntax \texttt{(ROW(...))} so that list elements \texttt{foo.*} will be expanded to a
  list of their member fields, rather than creating a nested row type field as formerly (Tom)
  The new behavior is substantially more useful since it allows, for example, triggers to check for
  data changes with \texttt{IF row(new.*) IS DISTINCT FROM row(old.*)}. The old behavior is still
  available by omitting \texttt{.*}.

• Make row comparisons follow SQL standard semantics and allow them to be used in index scans
  (Tom)
  Previously, \texttt{row =} and \texttt{<>} comparisons followed the standard but \texttt{< <= > >=} did not. A row
  comparison can now be used as an index constraint for a multicolonm index matching the row
  value.

• Make \texttt{row IS [NOT] NULL} tests follow SQL standard semantics (Tom)
  The former behavior conformed to the standard for simple cases with \texttt{IS NULL}, but \texttt{IS NOT NULL}
  would return true if any row field was non-null, whereas the standard says it should return true only
  when all fields are non-null.

• Make \texttt{SET CONSTRAINT} affect only one constraint (Kris Jurka)
  In previous releases, \texttt{SET CONSTRAINT} modified all constraints with a matching name. In this
  release, the schema search path is used to modify only the first matching constraint. A schema
  specification is also supported. This more nearly conforms to the SQL standard.

• Remove \texttt{RULE} permission for tables, for security reasons (Tom)
  As of this release, only a table’s owner can create or modify rules for the table. For backwards
  compatibility, \texttt{GRANT/REVOKE RULE} is still accepted, but it does nothing.

• Array comparison improvements (Tom)
  Now array dimensions are also compared.

• Change array concatenation to match documented behavior (Tom)
  This changes the previous behavior where concatenation would modify the array lower bound.

• Make command-line options of postmaster and postgres identical (Peter)
  This allows the postmaster to pass arguments to each backend without using \texttt{-o}. Note that some
  options are now only available as long-form options, because there were conflicting single-letter
  options.

• Deprecate use of postmaster symbolic link (Peter)
  postmaster and postgres commands now act identically, with the behavior determined by command-
  line options. The postmaster symbolic link is kept for compatibility, but is not really needed.

• Change \texttt{log\_duration} to output even if the query is not output (Tom)
  In prior releases, \texttt{log\_duration} only printed if the query appeared earlier in the log.

• Make \texttt{to\_char(time)} and \texttt{to\_char(interval)} treat \texttt{HH} and \texttt{HH12} as 12-hour intervals
  Most applications should use \texttt{HH24} unless they want a 12-hour display.

• Zero unmasked bits in conversion from \texttt{INET} to \texttt{CIDR} (Tom)
  This ensures that the converted value is actually valid for \texttt{CIDR}.
• Remove `australian_timezones` configuration variable (Joachim Wieland)
  This variable has been superseded by a more general facility for configuring timezone abbreviations.
• Improve cost estimation for nested-loop index scans (Tom)
  This might eliminate the need to set unrealistically small values of `random_page_cost`. If you have been using a very small `random_page_cost`, please recheck your test cases.
• Change behavior of `pg_dump -n` and `-t` options. (Greg Sabino Mullane)
  See the `pg_dump` manual page for details.
• Change `libpq PQdsplen()` to return a useful value (Martijn van Oosterhout)
• Declare `libpq PQgetssl()` as returning `void *`, rather than `SSL *` (Martijn van Oosterhout)
  This allows applications to use the function without including the OpenSSL headers.
• C-language loadable modules must now include a `PG_MODULE_MAGIC` macro call for version compatibility checking (Martijn van Oosterhout)
• For security’s sake, modules used by a PL/PerlU function are no longer available to PL/Perl functions (Andrew)

  **Note:** This also implies that data can no longer be shared between a PL/Perl function and a PL/PerlU function. Some Perl installations have not been compiled with the correct flags to allow multiple interpreters to exist within a single process. In this situation PL/Perl and PL/PerlU cannot both be used in a single backend. The solution is to get a Perl installation which supports multiple interpreters.

• In `contrib/xml2/`, rename `xml_valid()` to `xml_is_well_formed()` (Tom)
  `xml_valid()` will remain for backward compatibility, but its behavior will change to do schema checking in a future release.
• Remove `contrib/ora2pg/`, now at http://www.samse.fr/GPL/ora2pg
• Remove contrib modules that have been migrated to PgFoundry: `adddepend`, `dbase`, `dbmirror`, `fulltextindex`, `mac`, `userlock`
• Remove abandoned contrib modules: `mSQL-interface`, `tips`
• Remove QNX and BEOS ports (Bruce)
  These ports no longer had active maintainers.

**E.48.3. Changes**

Below you will find a detailed account of the changes between PostgreSQL 8.2 and the previous major release.

**E.48.3.1. Performance Improvements**

• Allow the planner to reorder outer joins in some circumstances (Tom)
In previous releases, outer joins would always be evaluated in the order written in the query. This change allows the query optimizer to consider reordering outer joins, in cases where it can determine that the join order can be changed without altering the meaning of the query. This can make a considerable performance difference for queries involving multiple outer joins or mixed inner and outer joins.

- Improve efficiency of IN (list-of-expressions) clauses (Tom)
- Improve sorting speed and reduce memory usage (Simon, Tom)
- Improve subtransaction performance (Alvaro, Itagaki Takahiro, Tom)
- Add FILLFACTOR to table and index creation (ITAGAKI Takahiro)
  
  This leaves extra free space in each table or index page, allowing improved performance as the database grows. This is particularly valuable to maintain clustering.
- Increase default values for shared_buffers and max_fsm_pages (Andrew)
- Improve locking performance by breaking the lock manager tables into sections (Tom)
  
  This allows locking to be more fine-grained, reducing contention.
- Reduce locking requirements of sequential scans (Qingqing Zhou)
- Reduce locking required for database creation and destruction (Tom)
- Improve the optimizer’s selectivity estimates for LIKE, ILIKE, and regular expression operations (Tom)
- Improve planning of joins to inherited tables and UNION ALL views (Tom)
- Allow constraint exclusion to be applied to inherited UPDATE and DELETE queries (Tom)
  
  This improves performance because the exact parameter values can be used in the plan.
- Protocol-level unnamed prepared statements are re-planned for each set of BIND values (Tom)
  
  This speeds up vacuuming of B-Tree indexes (Heikki Linnakangas, Tom)
- Avoid extra scan of tables without indexes during VACUUM (Greg Stark)
- Improve multicolumn GiST indexing (Oleg, Teodor)
- Remove dead index entries before B-Tree page split (Junji Teramoto)

**E.48.3.2. Server Changes**

- Allow a forced switch to a new transaction log file (Simon, Tom)
  
  This is valuable for keeping warm standby slave servers in sync with the master. Transaction log file switching now also happens automatically during pg_stop_backup(). This ensures that all transaction log files needed for recovery can be archived immediately.
- Add WAL informational functions (Simon)
  
  Add functions for interrogating the current transaction log insertion point and determining WAL filenames from the hex WAL locations displayed by pg_stop_backup() and related functions.
- Improve recovery from a crash during WAL replay (Simon)
Appendix E. Release Notes

The server now does periodic checkpoints during WAL recovery, so if there is a crash, future WAL recovery is shortened. This also eliminates the need for warm standby servers to replay the entire log since the base backup if they crash.

- Improve reliability of long-term WAL replay (Heikki, Simon, Tom)
  
  Formerly, trying to roll forward through more than 2 billion transactions would not work due to XID wraparound. This meant warm standby servers had to be reloaded from fresh base backups periodically.

- Add archive_timeout to force transaction log file switches at a given interval (Simon)
  
  This enforces a maximum replication delay for warm standby servers.

- Add native LDAP authentication (Magnus Hagander)
  
  This is particularly useful for platforms that do not support PAM, such as Windows.

- Add GRANT CONNECT ON DATABASE (Gevik Babakhani)
  
  This gives SQL-level control over database access. It works as an additional filter on top of the existing pg_hba.conf controls.

- Add support for SSL Certificate Revocation List (CRL) files (Libor Hohoš)
  
  The server and libpq both recognize CRL files now.

- GiST indexes are now clusterable (Teodor)

- Remove routine autovacuum server log entries (Bruce)
  
  pg_stat_activity now shows autovacuum activity.

- Track maximum XID age within individual tables, instead of whole databases (Alvaro)
  
  This reduces the overhead involved in preventing transaction ID wraparound, by avoiding unnecessary VACUUMs.

- Add last vacuum and analyze timestamp columns to the stats collector (Larry Rosenman)
  
  These values now appear in the pg_stat_*_tables system views.

- Improve performance of statistics monitoring, especially stats_command_string (Tom, Bruce)
  
  This release enables stats_command_string by default, now that its overhead is minimal. This means pg_stat_activity will now show all active queries by default.

- Add a waiting column to pg_stat_activity (Tom)
  
  This allows pg_stat_activity to show all the information included in the ps display.

- Add configuration parameter update_process_title to control whether the ps display is updated for every command (Bruce)
  
  On platforms where it is expensive to update the ps display, it might be worthwhile to turn this off and rely solely on pg_stat_activity for status information.

- Allow units to be specified in configuration settings (Peter)
  
  For example, you can now set shared_buffers to 32MB rather than mentally converting sizes.

- Add support for include directives in postgresql.conf (Joachim Wieland)

- Improve logging of protocol-level prepare/bind/execute messages (Bruce, Tom)
  
  Such logging now shows statement names, bind parameter values, and the text of the query being executed. Also, the query text is properly included in logged error messages when enabled by log_min_error_statement.
Appendix E. Release Notes

- Prevent `max_stack_depth` from being set to unsafe values
  
  On platforms where we can determine the actual kernel stack depth limit (which is most), make sure that the initial default value of `max_stack_depth` is safe, and reject attempts to set it to unsafely large values.

- Enable highlighting of error location in query in more cases (Tom)
  
  The server is now able to report a specific error location for some semantic errors (such as unrecognized column name), rather than just for basic syntax errors as before.

- Fix “failed to re-find parent key” errors in VACUUM (Tom)

- Clean out `pg_internal.init` cache files during server restart (Simon)
  
  This avoids a hazard that the cache files might contain stale data after PITR recovery.

- Fix race condition for truncation of a large relation across a gigabyte boundary by VACUUM (Tom)

- Fix bug causing needless deadlock errors on row-level locks (Tom)

- Fix bugs affecting multi-gigabyte hash indexes (Tom)

- Each backend process is now its own process group leader (Tom)
  
  This allows query cancel to abort subprocesses invoked from a backend or archive/recovery process.

E.48.3.3. Query Changes

- Add `INSERT/UPDATE/DELETE RETURNING` (Jonah Harris, Tom)
  
  This allows these commands to return values, such as the computed serial key for a new row. In the `UPDATE` case, values from the updated version of the row are returned.

- Add support for multiple-row `VALUES` clauses, per SQL standard (Joe, Tom)

  This allows `INSERT` to insert multiple rows of constants, or queries to generate result sets using constants. For example, `INSERT ... VALUES (...), (...), ....` and `SELECT * FROM (VALUES (...), (...), ....) AS alias(f1, ...)

- Allow `UPDATE` and `DELETE` to use an alias for the target table (Atsushi Ogawa)

  The SQL standard does not permit an alias in these commands, but many database systems allow one anyway for notational convenience.

- Allow `UPDATE` to set multiple columns with a list of values (Susanne Ebrecht)

  This is basically a short-hand for assigning the columns and values in pairs. The syntax is `UPDATE tab SET (column, ...) = (val, ...).

- Make row comparisons work per standard (Tom)

  The forms `<, <=, >, >=` now compare rows lexicographically, that is, compare the first elements, if equal compare the second elements, and so on. Formerly they expanded to an AND condition across all the elements, which was neither standard nor very useful.

- Add `CASCADE` option to TRUNCATE (Joachim Wieland)

  This causes TRUNCATE to automatically include all tables that reference the specified table(s) via foreign keys. While convenient, this is a dangerous tool — use with caution!

- Support FOR UPDATE and FOR SHARE in the same `SELECT` command (Tom)

- Add `IS NOT DISTINCT FROM` (Pavel Stehule)
This operator is similar to equality (\(=\)), but evaluates to true when both left and right operands are NULL, and to false when just one is, rather than yielding NULL in these cases.

- **Improve the length output used by** \texttt{UNION/INTERSECT/EXCEPT} (Tom)
  When all corresponding columns are of the same defined length, that length is used for the result, rather than a generic length.

- **Allow** \texttt{ILIKE} **to work for multi-byte encodings** (Tom)
  Internally, \texttt{ILIKE} now calls \texttt{lower()} and then uses \texttt{LIKE}. Locale-specific regular expression patterns still do not work in these encodings.

- **Enable** \texttt{standard_conforming_strings} **to be turned on** (Kevin Grittner)
  This allows backslash escaping in strings to be disabled, making PostgreSQL more standards-compliant. The default is off for backwards compatibility, but future releases will default this to on.

- **Do not flatten subqueries that contain** \texttt{volatile} **functions in their target lists** (Jaime Casanova)
  This prevents surprising behavior due to multiple evaluation of a volatile function (such as \texttt{random()} or \texttt{nextval()}). It might cause performance degradation in the presence of functions that are unnecessarily marked as volatile.

- **Add** \texttt{system views pg_prepared_statements} **and pg_cursors** **to show prepared statements and open cursors** (Joachim Wieland, Neil)
  These are very useful in pooled connection setups.

- **Support portal parameters in** \texttt{EXPLAIN} **and** \texttt{EXECUTE} (Tom)
  This allows, for example, JDBC ? parameters to work in these commands.

- **If SQL-level** \texttt{PREPARE} **parameters are unspecified, infer their types from the content of the query** (Neil)
  Protocol-level \texttt{PREPARE} already did this.

- **Allow** \texttt{LIMIT} **and** \texttt{OFFSET} **to exceed two billion** (Dhanaraj M)

### E.48.3.4. Object Manipulation Changes

- **Add** \texttt{TABLESPACE} **clause to** \texttt{CREATE TABLE AS} (Neil)
  This allows a tablespace to be specified for the new table.

- **Add** \texttt{ON COMMIT} **clause to** \texttt{CREATE TABLE AS} (Neil)
  This allows temporary tables to be truncated or dropped on transaction commit. The default behavior is for the table to remain until the session ends.

- **Add** \texttt{INCLUDING CONSTRAINTS} **to** \texttt{CREATE TABLE LIKE} (Greg Stark)
  This allows easy copying of \texttt{CHECK} constraints to a new table.

- **Allow the creation of placeholder (shell) types** (Martijn van Oosterhout)
  A shell type declaration creates a type name, without specifying any of the details of the type. Making a shell type is useful because it allows cleaner declaration of the type’s input/output functions, which must exist before the type can be defined “for real”. The syntax is \texttt{CREATE TYPE} \texttt{typename}.

- **Aggregate functions now support multiple input parameters** (Sergey Koposov, Tom)

- **Add new aggregate creation syntax** (Tom)
The new syntax is `CREATE AGGREGATE agname (input_type) (parameter_list)`. This more naturally supports the new multi-parameter aggregate functionality. The previous syntax is still supported.

- Add `ALTER ROLE PASSWORD NULL` to remove a previously set role password (Peter)
- Add `DROP object IF EXISTS` for many object types (Andrew)
  This allows `DROP` operations on non-existent objects without generating an error.
- Add `DROP OWNED` to drop all objects owned by a role (Alvaro)
- Add `REASSIGN OWNED` to reassign ownership of all objects owned by a role (Alvaro)
  This, and `DROP OWNED` above, facilitate dropping roles.
- Add `GRANT ON SEQUENCE` syntax (Bruce)
  This was added for setting sequence-specific permissions. `GRANT ON TABLE` for sequences is still supported for backward compatibility.
- Add `USAGE` permission for sequences that allows only `currval()` and `nextval()`, not `setval()` (Bruce)
  `USAGE` permission allows more fine-grained control over sequence access. Granting `USAGE` allows users to increment a sequence, but prevents them from setting the sequence to an arbitrary value using `setval()`.
- Add `ALTER TABLE [ NO ] INHERIT` (Greg Stark)
  This allows inheritance to be adjusted dynamically, rather than just at table creation and destruction. This is very valuable when using inheritance to implement table partitioning.
- Allow comments on global objects to be stored globally (Kris Jurka)
  Previously, comments attached to databases were stored in individual databases, making them ineffective, and there was no provision at all for comments on roles or table spaces. This change adds a new shared catalog `pg_shdescription` and stores comments on databases, roles, and table spaces therein.

### E.48.3.5. Utility Command Changes

- Add option to allow indexes to be created without blocking concurrent writes to the table (Greg Stark, Tom)
  The new syntax is `CREATE INDEX CONCURRENTLY`. The default behavior is still to block table modification while a index is being created.
- Provide advisory locking functionality (Abhijit Menon-Sen, Tom)
  This is a new locking API designed to replace what used to be in `/contrib/userlock`. The userlock code is now on pgfoundry.
- Allow `COPY` to dump a `SELECT` query (Zoltan Boszormenyi, Karel Zak)
  This allows `COPY` to dump arbitrary SQL queries. The syntax is `COPY (SELECT ...) TO`.
- Make the `COPY` command return a command tag that includes the number of rows copied (Volkan YAYICI)
- Allow `VACUUM` to expire rows without being affected by other concurrent `VACUUM` operations (Hannu Krossing, Alvaro, Tom)
- Make initdb detect the operating system locale and set the default `DateStyle` accordingly (Peter)
Appendix E. Release Notes

This makes it more likely that the installed `postgresql.conf` DateStyle value will be as desired.

- Reduce number of progress messages displayed by initdb (Tom)

E.48.3.6. Date/Time Changes

- Allow full timezone names in `timestamp` input values (Joachim Wieland)
- Support configurable timezone abbreviations (Joachim Wieland)
  A desired set of timezone abbreviations can be chosen via the configuration parameter `timezone_abbreviations`.
- Add `pg_timezone_abbrs` and `pg_timezone_names` views to show supported timezones (Magnus Hagander)
- Add `clock_timestamp()`, `statement_timestamp()`, and `transaction_timestamp()` (Bruce)
  `clock_timestamp()` is the current wall-clock time, `statement_timestamp()` is the time the current statement arrived at the server, and `transaction_timestamp()` is an alias for `now()`.
- Allow `to_char()` to print localized month and day names (Euler Taveira de Oliveira)
- Allow `to_char(time)` and `to_char(interval)` to output AM/PM specifications (Bruce)
  Intervals and times are treated as 24-hour periods, e.g. 25 hours is considered AM.
- Add new function `justify_interval()` to adjust interval units (Mark Dilger)
- Allow timezone offsets up to 14:59 away from GMT
  Kiribati uses GMT+14, so we’d better accept that.
- Interval computation improvements (Michael Glaeemann, Bruce)

E.48.3.7. Other Data Type and Function Changes

- Allow arrays to contain `NULL` elements (Tom)
- Allow assignment to array elements not contiguous with the existing entries (Tom)
  The intervening array positions will be filled with nulls. This is per SQL standard.
- New built-in operators for array-subset comparisons (`@>`, `@<`, `&&`) (Teodor, Tom)
  These operators can be indexed for many data types using GiST or GIN indexes.
- Add convenient arithmetic operations on INET/CIDR values (Stephen R. van den Berg)
  The new operators are `&` (and), `|` (or), `-` (not), `inet + int8`, `inet - int8`, and `inet - inet`.
- Add new aggregate functions from SQL:2003 (Neil)
  The new functions are `var_pop()`, `var_samp()`, `stddev_pop()`, and `stddev_samp()`. `var_samp()` and `stddev_samp()` are merely renamings of the existing aggregates `variance()` and `stddev()`. The latter names remain available for backward compatibility.
- Add SQL:2003 statistical aggregates (Sergey Koposov)
Appendix E. Release Notes

New functions: `regr_intercept()`, `regr_slope()`, `regr_r2()`, `corr()`, `covar_samp()`, `covar_pop()`, `regr_avgx()`, `regr_avgy()`, `regr_sxy()`, `regr_sxx()`, `regr_syy()`, `regr_count()`.

- Allow domains to be based on other domains (Tom)
- Properly enforce domain CHECK constraints everywhere (Neil, Tom)
  For example, the result of a user-defined function that is declared to return a domain type is now checked against the domain’s constraints. This closes a significant hole in the domain implementation.
- Fix problems with dumping renamed `SERIAL` columns (Tom)
  The fix is to dump a `SERIAL` column by explicitly specifying its `DEFAULT` and sequence elements, and reconstructing the `SERIAL` column on reload using a new `ALTER SEQUENCE` command. This also allows dropping a `SERIAL` column specification.
- Add a server-side sleep function `pg_sleep()` (Joachim Wieland)
- Add all comparison operators for the `tid` (tuple id) data type (Mark Kirkwood, Greg Stark, Tom)

E.48.3.8. PL/pgSQL Server-Side Language Changes

- Add `TG_table_name` and `TG_table_schema` to trigger parameters (Andrew)
  `TG_relname` is now deprecated. Comparable changes have been made in the trigger parameters for the other PLs as well.
- Allow `FOR` statements to return values to scalars as well as records and row types (Pavel Stehule)
- Add a `BY` clause to the `FOR` loop, to control the iteration increment (Jaime Casanova)
- Add `STRICT` to `SELECT INTO` (Matt Miller)
  `STRICT` mode throws an exception if more or less than one row is returned by the `SELECT`, for Oracle PL/SQL compatibility.

E.48.3.9. PL/Perl Server-Side Language Changes

- Add `table_name` and `table_schema` to trigger parameters (Adam Sjøgren)
- Add prepared queries (Dmitry Karasik)
- Make `$TD` trigger data a global variable (Andrew)
  Previously, it was lexical, which caused unexpected sharing violations.
- Run PL/Perl and PL/PerlU in separate interpreters, for security reasons (Andrew)
  In consequence, they can no longer share data nor loaded modules. Also, if Perl has not been compiled with the requisite flags to allow multiple interpreters, only one of these languages can be used in any given backend process.

E.48.3.10. PL/Python Server-Side Language Changes

- Named parameters are passed as ordinary variables, as well as in the `args[]` array (Sven Suursoho)
Appendix E. Release Notes

- Add `table_name` and `table_schema` to trigger parameters (Andrew)
- Allow returning of composite types and result sets (Sven Suursoho)
- Return result-set as `list`, `iterator`, or `generator` (Sven Suursoho)
- Allow functions to return `void` (Neil)
- Python 2.5 is now supported (Tom)

### E.48.3.11. `psql` Changes

- Add new command `\password` for changing role password with client-side password encryption (Peter)
- Allow `\c` to connect to a new host and port number (David, Volkan YAZICI)
- Add tablespace display to `\l+` (Philip Yarra)
- Improve `\df` slash command to include the argument names and modes (`OUT` or `INOUT`) of the function (David Fetter)
- Support binary `COPY` (Andreas Pflug)
- Add option to run the entire session in a single transaction (Simon)
  Use option `-i` or `--single-transaction`.
- Support for automatically retrieving `SELECT` results in batches using a cursor (Chris Mair)
  This is enabled using `\set FETCH_COUNT n`. This feature allows large result sets to be retrieved in `psql` without attempting to buffer the entire result set in memory.
- Make multi-line values align in the proper column (Martijn van Oosterhout)
  Field values containing newlines are now displayed in a more readable fashion.
- Save multi-line statements as a single entry, rather than one line at a time (Sergey E. Koposov)
  This makes up-arrow recall of queries easier. (This is not available on Windows, because that platform uses the native command-line editing present in the operating system.)
- Make the line counter 64-bit so it can handle files with more than two billion lines (David Fetter)
- Report both the returned data and the command status tag for `INSERT/UPDATE/DELETE RETURNING` (Tom)

### E.48.3.12. `pg_dump` Changes

- Allow complex selection of objects to be included or excluded by `pg_dump` (Greg Sabino Mullane)
  `pg_dump` now supports multiple `-n` (schema) and `-t` (table) options, and adds `-N` and `-T` options to exclude objects. Also, the arguments of these switches can now be wild-card expressions rather than single object names, for example `-t 'foo*'`, and a schema can be part of a `-t` or `-T` switch, for example `-t schema1.table1`.
- Add `pg_restore --no-data-for-failed-tables` option to suppress loading data if table creation failed (i.e., the table already exists) (Martin Pitt)
- Add `pg_restore` option to run the entire session in a single transaction (Simon)
  Use option `-i` or `--single-transaction`. 

1571
E.48.3.13. libpq Changes

- Add PQencryptPassword() to encrypt passwords (Tom)
  This allows passwords to be sent pre-encrypted for commands like ALTER ROLE ... PASSWORD.
- Add function PQisthreadsafe() (Bruce)
  This allows applications to query the thread-safety status of the library.
- Add PQdescribePrepared(), PQdescribePortal(), and related functions to return information about previously prepared statements and open cursors (Volkan YAZICI)
- Allow LDAP lookups from pg_service.conf (Laurenz Albe)
- Allow a hostname in ~/.pgpass to match the default socket directory (Bruce)
  A blank hostname continues to match any Unix-socket connection, but this addition allows entries that are specific to one of several postmasters on the machine.

E.48.3.14. ecpg Changes

- Allow SHOW to put its result into a variable (Joachim Wieland)
- Add COPY TO STDOUT (Joachim Wieland)
- Add regression tests (Joachim Wieland, Michael)
- Major source code cleanups (Joachim Wieland, Michael)

E.48.3.15. Windows Port

- Allow MSVC to compile the PostgreSQL server (Magnus, Hiroshi Saito)
- Add MSVC support for utility commands and pg_dump (Hiroshi Saito)
- Add support for Windows code pages 1253, 1254, 1255, and 1257 (Kris Jurka)
- Drop privileges on startup, so that the server can be started from an administrative account (Magnus)
  Stability fixes (Qingqing Zhou, Magnus)
- Add native semaphore implementation (Qingqing Zhou)
  The previous code mimicked SysV semaphores.

E.48.3.16. Source Code Changes

- Add GIN (Generalized Inverted iNdex) index access method (Teodor, Oleg)
- Remove R-tree indexing (Tom)
  Rtree has been re-implemented using GiST. Among other differences, this means that rtree indexes now have support for crash recovery via write-ahead logging (WAL).
- Reduce libraries needlessly linked into the backend (Martijn van Oosterhout, Tom)
- Add a configure flag to allow libedit to be preferred over GNU readline (Bruce)
Appendix E. Release Notes

Use configure --with-libedit-preferred.

- Allow installation into directories containing spaces (Peter)
- Improve ability to relocate installation directories (Tom)
- Add support for Solaris x86_64 using the Solaris compiler (Pierre Girard, Theo Schlossnagle, Bruce)
- Add DTrace support (Robert Lor)
- Add \texttt{PG\_VERSION\_NUM} for use by third-party applications wanting to test the backend version in C using \texttt{> and < comparisons} (Bruce)
- Add \texttt{XLOG\_BLCKSZ} as independent from \texttt{BLCKSZ} (Mark Wong)
- Add \texttt{LNLOCK\_STATS} define to report locking activity (Tom)
- Emit warnings for unknown configure options (Martijn van Oosterhout)
- Add server support for “plugin” libraries that can be used for add-on tasks such as debugging and performance measurement (Korry Douglas)
  This consists of two features: a table of “rendezvous variables” that allows separately-loaded shared libraries to communicate, and a new configuration parameter \texttt{local\_preload\_libraries} that allows libraries to be loaded into specific sessions without explicit cooperation from the client application. This allows external add-ons to implement features such as a PL/pgSQL debugger.
- Rename existing configuration parameter \texttt{preload\_libraries} to \texttt{shared\_preload\_libraries} (Tom)
  This was done for clarity in comparison to \texttt{local\_preload\_libraries}.
- Add new configuration parameter \texttt{server\_version\_num} (Greg Sabino Mullane)
  This is like \texttt{server\_version}, but is an integer, e.g. 80200. This allows applications to make version checks more easily.
- Add a configuration parameter \texttt{seq\_page\_cost} (Tom)
- Re-implement the regression test script as a C program (Magnus, Tom)
- Allow loadable modules to allocate shared memory and lightweight locks (Marc Munro)
- Add automatic initialization and finalization of dynamically loaded libraries (Ralf Engelschall, Tom)
  New functions \texttt{_PG\_init()} and \texttt{_PG\_fini()} are called if the library defines such symbols. Hence we no longer need to specify an initialization function in \texttt{shared\_preload\_libraries}; we can assume that the library used the \texttt{_PG\_init()} convention instead.
- Add \texttt{PG\_MODULE\_MAGIC} header block to all shared object files (Martijn van Oosterhout)
  The magic block prevents version mismatches between loadable object files and servers.
- Add shared library support for AIX (Laurenz Albe)
- New XML documentation section (Bruce)

**E.48.3.17. Contrib Changes**

- Major tsearch2 improvements (Oleg, Teodor)
  - multibyte encoding support, including UTF8
Appendix E. Release Notes

- query rewriting support
- improved ranking functions
- thesaurus dictionary support
- Ispell dictionaries now recognize MySpell format, used by OpenOffice
- GIN support

- Add adminpack module containing Pgadmin administration functions (Dave)
  These functions provide additional file system access routines not present in the default PostgreSQL server.

- Add sslinfo module (Victor Wagner)
  Reports information about the current connection’s SSL certificate.

- Add pgrowlocks module (Tatsuo)
  This shows row locking information for a specified table.

- Add hstore module (Oleg, Teodor)

- Add isn module, replacing isbn_isn (Jeremy Kronuz)
  This new implementation supports EAN13, UPC, ISBN (books), ISMN (music), and ISSN (serials).

- Add index information functions to pgstattuple (ITAGAKI Takahiro, Satoshi Nagayasu)

- Add pg_freespacemap module to display free space map information (Mark Kirkwood)

- pgcrypto now has all planned functionality (Marko Kreen)
  - Include iMath library in pgcrypto to have the public-key encryption functions always available.
  - Add SHA224 algorithm that was missing in OpenBSD code.
  - Activate builtin code for SHA224/256/384/512 hashes on older OpenSSL to have those algorithms always available.
  - New function gen_random_bytes() that returns cryptographically strong randomness. Useful for generating encryption keys.
  - Remove digest_exists(), hmac_exists() and cipherexists() functions.

- Improvements to cube module (Joshua Reich)
  New functions are cube(float[]), cube(float[], float[]), and cube_subset(cube, int4[]).

- Add async query capability to dblink (Kai Londenberg, Joe Conway)

- New operators for array-subset comparisons (\@\rangle, <\#, \&\&\) (Tom)
  Various contrib packages already had these operators for their datatypes, but the naming wasn’t consistent. We have now added consistently named array-subset comparison operators to the core code and all the contrib packages that have such functionality. (The old names remain available, but are deprecated.)

- Add uninstall scripts for all contrib packages that have install scripts (David, Josh Drake)
Appendix E. Release Notes

E.49. Release 8.1.23

Release date: 2010-12-16

This release contains a variety of fixes from 8.1.22. For information about new features in the 8.1 major release, see Section E.72.

This is expected to be the last PostgreSQL release in the 8.1.X series. Users are encouraged to update to a newer release branch soon.

E.49.1. Migration to Version 8.1.23

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see the release notes for 8.1.18.

E.49.2. Changes

- Force the default wal_sync_method to be fdatasync on Linux (Tom Lane, Marti Raudsepp)
  The default on Linux has actually been fdatasync for many years, but recent kernel changes caused PostgreSQL to choose open_dataasync instead. This choice did not result in any performance improvement, and caused outright failures on certain filesystems, notably ext4 with the data=journal mount option.

- Fix recovery from base backup when the starting checkpoint WAL record is not in the same WAL segment as its redo point (Jeff Davis)

- Add support for detecting register-stack overrun on IA64 (Tom Lane)
  The IA64 architecture has two hardware stacks. Full prevention of stack-overrun failures requires checking both.

- Add a check for stack overflow in copyObject() (Tom Lane)
  Certain code paths could crash due to stack overflow given a sufficiently complex query.

- Fix detection of page splits in temporary GiST indexes (Heikki Linnakangas)
  It is possible to have a “concurrent” page split in a temporary index, if for example there is an open cursor scanning the index when an insertion is done. GiST failed to detect this case and hence could deliver wrong results when execution of the cursor continued.

- Avoid memory leakage while ANALYZE'ing complex index expressions (Tom Lane)

- Ensure an index that uses a whole-row Var still depends on its table (Tom Lane)
  An index declared like create index i on t (foo(t.*)) would not automatically get dropped when its table was dropped.

- Do not “inline” a SQL function with multiple OUT parameters (Tom Lane)
  This avoids a possible crash due to loss of information about the expected result rowtype.

- Fix constant-folding of COALESCE() expressions (Tom Lane)
  The planner would sometimes attempt to evaluate sub-expressions that in fact could never be reached, possibly leading to unexpected errors.
• Add print functionality for InhRelation nodes (Tom Lane)
  This avoids a failure when `debug_print_parse` is enabled and certain types of query are executed.
• Fix incorrect calculation of distance from a point to a horizontal line segment (Tom Lane)
  This bug affected several different geometric distance-measurement operators.
• Fix PL/pgSQL's handling of “simple” expressions to not fail in recursion or error-recovery cases (Tom Lane)
• Fix bug in contrib/cube’s GiST picksplit algorithm (Alexander Korotkov)
  This could result in considerable inefficiency, though not actually incorrect answers, in a GiST index on a cube column. If you have such an index, consider `REINDEXing` it after installing this update.
• Don’t emit “identifier will be truncated” notices in contrib/dblink except when creating new connections (Itagaki Takahiro)
• Fix potential coredump on missing public key in contrib/pgcrypto (Marti Raudsepp)
• Fix memory leak in contrib/xml2’s XPath query functions (Tom Lane)
• Update time zone data files to tzdata release 2010o for DST law changes in Fiji and Samoa; also historical corrections for Hong Kong.

E.50. Release 8.1.22

Release date: 2010-10-04

This release contains a variety of fixes from 8.1.21. For information about new features in the 8.1 major release, see Section E.72.

The PostgreSQL community will stop releasing updates for the 8.1.X release series in November 2010. Users are encouraged to update to a newer release branch soon.

E.50.1. Migration to Version 8.1.22

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see the release notes for 8.1.18.

E.50.2. Changes

• Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)
  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a `SECURITY DEFINER` function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this
change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function’s owner.

The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).

• Prevent possible crashes in \texttt{pg\_get\_expr()} by disallowing it from being called with an argument that is not one of the system catalog columns it’s intended to be used with (Heikki Linnakangas, Tom Lane)

• Fix “cannot handle unplanned sub-select” error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.

• Prevent \texttt{show\_session\_authorization()} from crashing within autovacuum processes (Tom Lane)

• Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)

• Fix possible failure when hashing a pass-by-reference function result (Tao Ma, Tom Lane)

• Take care to \texttt{fsync} the contents of lockfiles (both postmaster.pid and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.

• Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)

  The original coding could result in a crash if there was limited stack space.

• Fix \texttt{log\_line\_prefix}’s \texttt{\%i} escape, which could produce junk early in backend startup (Tom Lane)

• Fix possible data corruption in \texttt{ALTER TABLE ... SET TABLESPACE} when archiving is enabled (Jeff Davis)

• Allow \texttt{CREATE DATABASE} and \texttt{ALTER DATABASE ... SET TABLESPACE} to be interrupted by query-cancel (Guillaume Lelarge)

• In PL/Python, defend against null pointer results from \texttt{PyCObject\_AsVoidPtr} and \texttt{PyCObject\_FromVoidPtr} (Peter Eisentraut)

• Improve contrib/dblink’s handling of tables containing dropped columns (Tom Lane)

• Fix connection leak after “duplicate connection name” errors in contrib/dblink (Itagaki Takahiro)

• Fix contrib/dblink to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)

• Update build infrastructure and documentation to reflect the source code repository’s move from CVS to Git (Magnus Hagander and others)
Appendix E. Release Notes

- Update time zone data files to tzdata release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.

This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.

E.51. Release 8.1.21

Release date: 2010-05-17

This release contains a variety of fixes from 8.1.20. For information about new features in the 8.1 major release, see Section E.72.

E.51.1. Migration to Version 8.1.21

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see the release notes for 8.1.18.

E.51.2. Changes

- Enforce restrictions in \texttt{plperl} using an opmask applied to the whole interpreter, instead of using \texttt{Safe.pm} (Tim Bunce, Andrew Dunstan)

  Recent developments have convinced us that \texttt{Safe.pm} is too insecure to rely on for making \texttt{plperl} trustable. This change removes use of \texttt{Safe.pm} altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl’s \texttt{strict} pragma in a natural way in \texttt{plperl}, and that Perl’s \texttt{$a} and \texttt{$b} variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from \texttt{pltcl_modules} (Tom)

  PL/Tcl’s feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless \texttt{pltcl_modules} is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted “normal” Tcl interpreter unless we are really going to execute a \texttt{pltclu} function. (CVE-2010-1170)

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran \texttt{ALTER USER ... RESET ALL} for himself, or \texttt{ALTER DATABASE ... RESET ALL} for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the \texttt{ALTER} will only remove the parameters that the user has permission to change.
• Avoid possible crash during backend shutdown if shutdown occurs when a CONTEXT addition would be made to log entries (Tom)

In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

• Update pl/perl’s ppport.h for modern Perl versions (Andrew)

• Fix assorted memory leaks in pl/python (Andreas Freund, Tom)

• Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)

• Ensure that contrib/pgstattuple functions respond to cancel interrupts promptly (Tatsuhito Kasahara)

• Make server startup deal properly with the case that shmget() returns EINVAL for an existing shared memory segment (Tom)

   This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.

• Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.

E.52. Release 8.1.20

Release date: 2010-03-15

This release contains a variety of fixes from 8.1.19. For information about new features in the 8.1 major release, see Section E.72.

E.52.1. Migration to Version 8.1.20

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see the release notes for 8.1.18.

E.52.2. Changes

• Add new configuration parameter ssl_renegotiation_limit to control how often we do session key renegotiation for an SSL connection (Magnus)

   This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.

• Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)

• Fix server memory leak associated with use of savepoints and a client encoding different from server’s encoding (Tom)
Appendix E. Release Notes

- Make substring() for bit types treat any negative length as meaning “all the rest of the string” (Tom)
  
  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).

- Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)

- Fix some cases of pathologically slow regular expression matching (Tom)

- Fix the STOP WAL LOCATION entry in backup history files to report the next WAL segment’s name when the end location is exactly at a segment boundary (Itagaki Takahiro)

- Fix some more cases of temporary-file leakage (Heikki)

  This corrects a problem introduced in the previous minor release. One case that failed is when a plpgsql function returning set is called within another function’s exception handler.

- When reading pg_hba.conf and related files, do not treat @something as a file inclusion request if the @ appears inside quote marks; also, never treat @ by itself as a file inclusion request (Tom)

  This prevents erratic behavior if a role or database name starts with @. If you need to include a file whose path name contains spaces, you can still do so, but you must write @"/path to/file" rather than putting the quotes around the whole construct.

- Prevent infinite loop on some platforms if a directory is named as an inclusion target in pg_hba.conf and related files (Tom)

- Fix psql’s numericlocale option to not format strings it shouldn’t in latex and troff output formats (Heikki)

- Fix plpgsql failure in one case where a composite column is set to NULL (Tom)

- Add volatile markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)

- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)

  The only known symptom of this oversight is that the Tcl clock command misbehaves if using Tcl 8.5 or later.

- Prevent crash in contrib/dblink when too many key columns are specified to a dblink_build_sql_* function (Rushabh Lathia, Joe Conway)

- Fix assorted crashes in contrib/xml2 caused by sloppy memory management (Tom)

- Update time zone data files to tzdata release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

E.53. Release 8.1.19

Release date: 2009-12-14

This release contains a variety of fixes from 8.1.18. For information about new features in the 8.1 major release, see Section E.72.
Appendix E. Release Notes

E.53.1. Migration to Version 8.1.19

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.18, see the release notes for 8.1.18.

E.53.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)
  
  This change prevents allegedly-immutable index functions from possibly subverting a superuser’s session (CVE-2009-4136).

- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)
  
  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).

- Fix possible crash during backend-startup-time cache initialization (Tom)

- Prevent signals from interrupting VACUUM at unsafe times (Alvaro)
  
  This fix prevents a PANIC if a VACUUM FULL is cancelled after it’s already committed its tuple movements, as well as transient errors if a plain VACUUM is interrupted after having truncated the table.

- Fix possible crash due to integer overflow in hash table size calculation (Tom)
  
  This could occur with extremely large planner estimates for the size of a hashjoin’s result.

- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)

- Ensure that shared tuple-level locks held by prepared transactions are not ignored (Heikki)

- Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction (Heikki)

- Fix PAM password processing to be more robust (Tom)
  
  The previous code is known to fail with the combination of the Linux pam_krb5 PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.

- Fix processing of ownership dependencies during CREATE OR REPLACE FUNCTION (Tom)

- Ensure that Perl arrays are properly converted to PostgreSQL arrays when returned by a set-returning PL/Perl function (Andrew Dunstan, Abhijit Menon-Sen)
  
  This worked correctly already for non-set-returning functions.

- Fix rare crash in exception processing in PL/Python (Peter)

- Ensure psql’s flex module is compiled with the correct system header definitions (Tom)
  
  This fixes build failures on platforms where --enable-largefile causes incompatible changes in the generated code.

- Make the postmaster ignore any application_name parameter in connection request packets, to improve compatibility with future libpq versions (Tom)

- Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina, Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for Hong Kong.
E.54. Release 8.1.18

**Release date:** 2009-09-09

This release contains a variety of fixes from 8.1.17. For information about new features in the 8.1 major release, see Section E.72.

E.54.1. Migration to Version 8.1.18

A dump/restore is not required for those running 8.1.X. However, if you have any hash indexes on `interval` columns, you must `REINDEX` them after updating to 8.1.18. Also, if you are upgrading from a version earlier than 8.1.15, see the release notes for 8.1.15.

E.54.2. Changes

- **Disallow** `RESET ROLE` and `RESET SESSION AUTHORIZATION` inside security-definer functions
  (Tom, Heikki)

  This covers a case that was missed in the previous patch that disallowed `SET ROLE` and `SET SESSION AUTHORIZATION` inside security-definer functions. (See CVE-2007-6600)

- **Fix** handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function
  (Tom)

- **Fix** hash calculation for data type `interval` (Tom)

  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must `REINDEX` them after updating.

- **Treat** `to_char(..., 'TH')` as an uppercase ordinal suffix with ‘HH’/’HH12’ (Heikki)

  It was previously handled as ‘th’ (lowercase).

- **Fix** overflow for `INTERVAL ‘x ms’` when x is more than 2 million and integer datetimes are in use
  (Alex Hunsaker)

- **Fix** calculation of distance between a point and a line segment (Tom)

  This led to incorrect results from a number of geometric operators.

- **Fix** `money` data type to work in locales where currency amounts have no fractional digits, e.g. Japan
  (Itagaki Takahiro)

- **Properly round** datetime input like `00:12:57.9999999999999999999999999999` (Tom)

- **Fix** portability issues in plperl initialization (Andrew Dunstan)

- **Fix** `pg_ctl` to not go into an infinite loop if `postgresql.conf` is empty (Jeff Davis)

- **Fix contrib/xml2's xslt_process()** to properly handle the maximum number of parameters
  (twenty) (Tom)

- **Improve** robustness of libpq’s code to recover from errors during `COPY FROM STDIN` (Tom)

- **Avoid including** conflicting readline and editline header files when both libraries are installed
  (Zdenek Kotala)
Appendix E. Release Notes

- Update time zone data files to tzdata release 2009l for DST law changes in Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan (historical correction only), Mauritius, Morocco, Palestine, Syria, Tunisia.

E.55. Release 8.1.17

Release date: 2009-03-16

This release contains a variety of fixes from 8.1.16. For information about new features in the 8.1 major release, see Section E.72.

E.55.1. Migration to Version 8.1.17

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.15, see the release notes for 8.1.15.

E.55.2. Changes

- Prevent error recursion crashes when encoding conversion fails (Tom)
  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that any error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)
- Disallow CREATE CONVERSION with the wrong encodings for the specified conversion function (Heikki)
  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.
- Fix core dump when to_char() is given format codes that are inappropriate for the type of the data argument (Tom)
- Fix decompilation of CASE WHEN with an implicit coercion (Tom)
  This mistake could lead to Assert failures in an Assert-enabled build, or an “unexpected CASE WHEN clause” error message in other cases, when trying to examine or dump a view.
- Fix possible misassignment of the owner of a TOAST table’s rowtype (Tom)
  If CLUSTER or a rewriting variant of ALTER TABLE were executed by someone other than the table owner, the pg_type entry for the table’s TOAST table would end up marked as owned by that someone. This caused no immediate problems, since the permissions on the TOAST rowtype aren’t examined by any ordinary database operation. However, it could lead to unexpected failures if one later tried to drop the role that issued the command (in 8.1 or 8.2), or “owner of data type appears to be invalid” warnings from pg_dump after having done so (in 8.3).
Appendix E. Release Notes

- Clean up PL/pgSQL error status variables fully at block exit (Ashesh Vashi and Dave Page)
  This is not a problem for PL/pgSQL itself, but the omission could cause the PL/pgSQL Debugger
to crash while examining the state of a function.
- Add MUST (Mauritius Island Summer Time) to the default list of known timezone abbreviations
  (Xavier Bugaud)

E.56. Release 8.1.16

Release date: 2009-02-02

This release contains a variety of fixes from 8.1.15. For information about new features in the 8.1
major release, see Section E.72.

E.56.1. Migration to Version 8.1.16

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version
earlier than 8.1.15, see the release notes for 8.1.15.

E.56.2. Changes

- Fix crash in autovacuum (Alvaro)
  The crash occurs only after vacuuming a whole database for anti-transaction-wraparound purposes,
  which means that it occurs infrequently and is hard to track down.
- Improve handling of URLs in headline() function (Teodor)
- Improve handling of overlength headlines in headline() function (Teodor)
- Prevent possible Assert failure or misconversion if an encoding conversion is created with the
  wrong conversion function for the specified pair of encodings (Tom, Heikki)
- Avoid unnecessary locking of small tables in VACUUM (Heikki)
- Ensure that the contents of a holdable cursor don’t depend on the contents of TOAST tables (Tom)
  Previously, large field values in a cursor result might be represented as TOAST pointers, which
  would fail if the referenced table got dropped before the cursor is read, or if the large value is
deleted and then vacuumed away. This cannot happen with an ordinary cursor, but it could with a
cursor that is held past its creating transaction.
- Fix uninitialized variables in contrib/tsearch2's get_covers() function (Teodor)
- Fix configure script to properly report failure when unable to obtain linkage information for PL/Perl
  (Andrew)
- Make all documentation reference pgsql-bugs and/or pgsql-hackers as appropriate, instead
  of the now-decommissioned pgsq1-ports and pgsq1-patches mailing lists (Tom)
- Update time zone data files to tzdata release 2009a (for Kathmandu and historical DST corrections
  in Switzerland, Cuba)
E.57. Release 8.1.15

Release date: 2008-11-03

This release contains a variety of fixes from 8.1.14. For information about new features in the 8.1 major release, see Section E.72.

E.57.1. Migration to Version 8.1.15

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2. Also, if you were running a previous 8.1.X release, it is recommended to REINDEX all GiST indexes after the upgrade.

E.57.2. Changes

- Fix GiST index corruption due to marking the wrong index entry “dead” after a deletion (Teodor)
  This would result in index searches failing to find rows they should have found. Corrupted indexes can be fixed with REINDEX.
- Fix backend crash when the client encoding cannot represent a localized error message (Tom)
  We have addressed similar issues before, but it would still fail if the “character has no equivalent” message itself couldn’t be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.
- Fix possible crash when deeply nested functions are invoked from a trigger (Tom)
- Fix mis-expansion of rule queries when a sub-SELECT appears in a function call in FROM, a multi-row VALUES list, or a RETURNING list (Tom)
  The usual symptom of this problem is an “unrecognized node type” error.
- Ensure an error is reported when a newly-defined PL/pgSQL trigger function is invoked as a normal function (Tom)
- Prevent possible collision of relfilenode numbers when moving a table to another tablespace with ALTER SET TABLESPACE (Heikki)
  The command tried to re-use the existing filename, instead of picking one that is known unused in the destination directory.
- Fix incorrect tsearch2 headline generation when single query item matches first word of text (Sushant Sinha)
- Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an --enable-integer-datetimes build (Ron Mayer)
- Ensure SPI_getvalue and SPI_getbinval behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)
  This situation is normal when a table has had columns added or removed, but these two functions didn’t handle it properly. The only likely consequence is an incorrect error indication.
- Fix ecpg’s parsing of CREATE ROLE (Michael)
E.58. Release 8.1.14

Release date: 2008-09-22

This release contains a variety of fixes from 8.1.13. For information about new features in the 8.1 major release, see Section E.72.

E.58.1. Migration to Version 8.1.14

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.58.2. Changes

- Widen local lock counters from 32 to 64 bits (Tom)
  This responds to reports that the counters could overflow in sufficiently long transactions, leading to unexpected “lock is already held” errors.
- Fix possible duplicate output of tuples during a GiST index scan (Teodor)
- Add checks in executor startup to ensure that the tuples produced by an INSERT or UPDATE will match the target table’s current rowtype (Tom)
  ALTER COLUMN TYPE, followed by re-use of a previously cached plan, could produce this type of situation. The check protects against data corruption and/or crashes that could ensue.
- Fix AT TIME ZONE to first try to interpret its timezone argument as a timezone abbreviation, and only try it as a full timezone name if that fails, rather than the other way around as formerly (Tom)
  The timestamp input functions have always resolved ambiguous zone names in this order. Making AT TIME ZONE do so as well improves consistency, and fixes a compatibility bug introduced in 8.1: in ambiguous cases we now behave the same as 8.0 and before did, since in the older versions AT TIME ZONE accepted only abbreviations.
- Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)
- Improve performance of writing very long log messages to syslog (Tom)
- Fix bug in backwards scanning of a cursor on a SELECT DISTINCT ON query (Tom)
- Fix planner bug with nested sub-select expressions (Tom)
  If the outer sub-select has no direct dependency on the parent query, but the inner one does, the outer value might not get recalculated for new parent query rows.
• Fix planner to estimate that `GROUP BY` expressions yielding boolean results always result in two groups, regardless of the expressions’ contents (Tom)

This is very substantially more accurate than the regular `GROUP BY` estimate for certain boolean tests like `col IS NULL`.

• Fix PL/pgSQL to not fail when a `FOR` loop’s target variable is a record containing composite-type fields (Tom)

• Fix PL/Tcl to behave correctly with Tcl 8.5, and to be more careful about the encoding of data sent to or from Tcl (Tom)

• Fix PL/Python to work with Python 2.5

This is a back-port of fixes made during the 8.2 development cycle.

• Improve `pg_dump` and `pg_restore`’s error reporting after failure to send a SQL command (Tom)

• Fix `pg_ctl` to properly preserve postmaster command-line arguments across a `restart` (Bruce)

• Update time zone data files to tzdata release 2008f (for DST law changes in Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine, and Paraguay)

### E.59. Release 8.1.13

**Release date:** 2008-06-12

This release contains one serious and one minor bug fix over 8.1.12. For information about new features in the 8.1 major release, see Section E.72.

#### E.59.1. Migration to Version 8.1.13

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

#### E.59.2. Changes

• Make `pg_get_ruledef()` parenthesize negative constants (Tom)

Before this fix, a negative constant in a view or rule might be dumped as, say, `-42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL to reject what had been a valid `SELECT DISTINCT` view query. Since this could result in `pg_dump` output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.

• Make `ALTER AGGREGATE ... OWNER TO update pg_shdepend` (Tom)

This oversight could lead to problems if the aggregate was later involved in a `DROP OWNED` or `REASSIGN OWNED` operation.
E.60. Release 8.1.12

Release date: never released

This release contains a variety of fixes from 8.1.11. For information about new features in the 8.1 major release, see Section E.72.

E.60.1. Migration to Version 8.1.12

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.60.2. Changes

- **Fix ALTER TABLE ADD COLUMN ... PRIMARY KEY** so that the new column is correctly checked to see if it’s been initialized to all non-nulls (Brendan Jurd)
  
  Previous versions neglected to check this requirement at all.

- Fix possible CREATE TABLE failure when inheriting the “same” constraint from multiple parent relations that inherited that constraint from a common ancestor (Tom)

- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic “Yo” characters (e and E with two dots) (Sergey Burladyan)

- Fix a few datatype input functions that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)
  
  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched ORDER BY and DISTINCT expressions.

- Fix a corner case in regular-expression substring matching (substring(string from pattern)) (Tom)

  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn’t got a match. An example is substring(’foo’ from ’foo(bar)?’). This should return NULL, since (bar) isn’t matched, but it was mistakenly returning the whole-pattern match instead (ie, foo).

- Update time zone data files to tzdta release 2008c (for DST law changes in Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, Argentina/San_Luis, and Chile)

- Fix incorrect result from ecpg’s PGTYPEStimestamp_sub() function (Michael)

- Fix core dump in contrib/xml2’s xpath_table() function when the input query returns a NULL value (Tom)

- Fix contrib/xml2’s makefile to not override CFLAGS (Tom)

- Fix DatumGetBool macro to not fail with gcc 4.3 (Tom)
  
  This problem affects “old style” (V0) C functions that return boolean. The fix is already in 8.3, but the need to back-patch it was not realized at the time.

- Fix longstanding LISTEN/NOTIFY race condition (Tom)
Appendix E. Release Notes

In rare cases a session that had just executed a `LISTEN` might not get a notification, even though one would be expected because the concurrent transaction executing `NOTIFY` was observed to commit later.

A side effect of the fix is that a transaction that has executed a not-yet-committed `LISTEN` command will not see any row in `pg_listener` for the `LISTEN`, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

• Disallow `LISTEN` and `UNLISTEN` within a prepared transaction (Tom)

This was formerly allowed but trying to do it had various unpleasant consequences, notably that the originating backend could not exit as long as an `UNLISTEN` remained uncommitted.

• Fix rare crash when an error occurs during a query using a hash index (Heikki)

• Fix input of datetime values for February 29 in years BC (Tom)

The former coding was mistaken about which years were leap years.

• Fix “unrecognized node type” error in some variants of `ALTER OWNER` (Tom)

• Fix `pg_ctl` to correctly extract the postmaster’s port number from command-line options (Itagaki Takahiro, Tom)

Previously, `pg_ctl start -w` could try to contact the postmaster on the wrong port, leading to bogus reports of startup failure.

• Use `-fwrapv` to defend against possible misoptimization in recent gcc versions (Tom)

This is known to be necessary when building PostgreSQL with gcc 4.3 or later.

• Fix display of constant expressions in `ORDER BY` and `GROUP BY` (Tom)

An explicitly casted constant would be shown incorrectly. This could for example lead to corruption of a view definition during dump and reload.

• Fix `libpq` to handle `NOTICE` messages correctly during `COPY OUT` (Tom)

This failure has only been observed to occur when a user-defined datatype’s output routine issues a `NOTICE`, but there is no guarantee it couldn’t happen due to other causes.

E.61. Release 8.1.11

Release date: 2008-01-07

This release contains a variety of fixes from 8.1.10, including fixes for significant security issues. For information about new features in the 8.1 major release, see Section E.72.

This is the last 8.1.X release for which the PostgreSQL community will produce binary packages for Windows. Windows users are encouraged to move to 8.2.X or later, since there are Windows-specific fixes in 8.2.X that are impractical to back-port. 8.1.X will continue to be supported on other platforms.

E.61.1. Migration to Version 8.1.11

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.
E.61.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running VACUUM, ANALYZE, etc (Tom)

Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as VACUUM FULL, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including VACUUM, ANALYZE, REINDEX, and CLUSTER) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for SECURITY DEFINER functions. To prevent bypassing this security measure, execution of SET SESSION AUTHORIZATION and SET ROLE is now forbidden within a SECURITY DEFINER context. (CVE-2007-6600)

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)

Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

The fix that appeared for this in 8.1.10 was incomplete, as it plugged the hole for only some dblink functions. (CVE-2007-6601, CVE-2007-3278)

- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)

- Improve planner’s handling of LIKE/regex estimation in non-C locales (Tom)

- Fix planner failure in some cases of WHERE false AND var IN (SELECT ... ) (Tom)

- Preserve the tablespace of indexes that are rebuilt by ALTER TABLE ... ALTER COLUMN TYPE (Tom)

- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop time was used (Simon)

This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL segment, and seems simpler and cleaner than the original definition.

- Make VACUUM not use all of maintenance_work_mem when the table is too small for it to be useful (Alvaro)

- Fix potential crash in translate() when using a multibyte database encoding (Tom)

- Fix overflow in extract(epoch from interval) for intervals exceeding 68 years (Tom)

- Fix PL/Perl to not fail when a UTF-8 regular expression is used in a trusted function (Andrew)

- Fix PL/Perl to cope when platform’s Perl defines type bool as int rather than char (Tom)

While this could theoretically happen anywhere, no standard build of Perl did things this way ... until Mac OS X 10.5.

- Fix PL/Python to not crash on long exception messages (Alvaro)
• Fix pg_dump to correctly handle inheritance child tables that have default expressions different from their parent’s (Tom)
• Fix libpq crash when PGPASSFILE refers to a file that is not a plain file (Martin Pitt)
• ecpg parser fixes (Michael)
• Make contrib/pgcrypto defend against OpenSSL libraries that fail on keys longer than 128 bits; which is the case at least on some Solaris versions (Marko Kreen)
• Make contrib/tablefunc’s crosstab() handle NULL rowid as a category in its own right, rather than crashing (Joe)
• Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)
• Fix crash of to_tsvector() on huge input strings (Teodor)
• Require a specific version of Autoconf to be used when re-generating the configure script (Peter)

This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it’s your responsibility whether the result works or not.

E.62. Release 8.1.10

Release date: 2007-09-17

This release contains a variety of fixes from 8.1.9. For information about new features in the 8.1 major release, see Section E.72.

E.62.1. Migration to Version 8.1.10

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.62.2. Changes

• Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent VACUUM on the same table (Tom)
• Make CREATE DOMAIN ... DEFAULT NULL work properly (Tom)
• Allow the interval data type to accept input consisting only of milliseconds or microseconds (Neil)
• Speed up rtree index insertion (Teodor)
• Fix excessive logging of SSL error messages (Tom)
• Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)
• Fix crash when log_min_error_statement logging runs out of memory (Tom)
Appendix E. Release Notes

- Fix incorrect handling of some foreign-key corner cases (Tom)
- Prevent REINDEX and CLUSTER from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Update the time zone database rules, particularly New Zealand’s upcoming changes (Tom)
- Windows socket improvements (Magnus)
- Suppress timezone name (%Z) in log timestamps on Windows because of possible encoding mismatches (Tom)
- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

E.63. Release 8.1.9

Release date: 2007-04-23

This release contains a variety of fixes from 8.1.8, including a security fix. For information about new features in the 8.1 major release, see Section E.72.

E.63.1. Migration to Version 8.1.9

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.63.2. Changes

- Support explicit placement of the temporary-table schema within search_path, and disable searching it for functions and operators (Tom)
  This is needed to allow a security-definer function to set a truly secure value of search_path. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See CREATE FUNCTION for more information.
- /contrib/tsearch2 crash fixes (Teodor)
- Require COMMIT PREPARED to be executed in the same database as the transaction was prepared in (Heikki)
- Fix potential-data-corruption bug in how VACUUM FULL handles UPDATE chains (Tom, Pavan Deolasee)
- Planner fixes, including improving outer join and bitmap scan selection logic (Tom)
- Fix PANIC during enlargement of a hash index (bug introduced in 8.1.6) (Tom)
- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)
E.64. Release 8.1.8

**Release date:** 2007-02-07

This release contains one fix from 8.1.7. For information about new features in the 8.1 major release, see Section E.72.

**E.64.1. Migration to Version 8.1.8**

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

**E.64.2. Changes**

- Remove overly-restrictive check for type length in constraints and functional indexes(Tom)

E.65. Release 8.1.7

**Release date:** 2007-02-05

This release contains a variety of fixes from 8.1.6, including a security fix. For information about new features in the 8.1 major release, see Section E.72.

**E.65.1. Migration to Version 8.1.7**

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

**E.65.2. Changes**

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)
  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it’s declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Improve `VACUUM` performance for databases with many tables (Tom)
Appendix E. Release Notes

- Fix autovacuum to avoid leaving non-permanent transaction IDs in non-connectable databases (Alvaro)
  This bug affects the 8.1 branch only.
- Fix for rare Assert() crash triggered by UNION (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)
- Fix bogus “permission denied” failures occurring on Windows due to attempts to fsync already-deleted files (Magnus, Tom)
- Fix possible crashes when an already-in-use PL/pgSQL function is updated (Tom)

E.66. Release 8.1.6

Release date: 2007-01-08

This release contains a variety of fixes from 8.1.5. For information about new features in the 8.1 major release, see Section E.72.

E.66.1. Migration to Version 8.1.6

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.66.2. Changes

- Improve handling of getaddrinfo() on AIX (Tom)
  This fixes a problem with starting the statistics collector, among other things.
- Fix pg_restore to handle a tar-format backup that contains large objects (blobs) with comments (Tom)
- Fix “failed to re-find parent key” errors in VACUUM (Tom)
- Clean out pg_internal.init cache files during server restart (Simon)
  This avoids a hazard that the cache files might contain stale data after PITR recovery.
- Fix race condition for truncation of a large relation across a gigabyte boundary by VACUUM (Tom)
- Fix bug causing needless deadlock errors on row-level locks (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix possible deadlock in Windows signal handling (Teodor)
- Fix error when constructing an ARRAY[] made up of multiple empty elements (Tom)
- Fix ecpg memory leak during connection (Michael)
- Fix for Darwin (OS X) compilation (Tom)
Appendix E. Release Notes

- `to_number()` and `to_char(numeric)` are now STABLE, not IMMUTABLE, for new initdb installs (Tom)
  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)
  This improves psql \d performance also.
- Update timezone database
  This affects Australian and Canadian daylight-savings rules in particular.

E.67. Release 8.1.5

**Release date:** 2006-10-16

This release contains a variety of fixes from 8.1.4. For information about new features in the 8.1 major release, see Section E.72.

E.67.1. Migration to Version 8.1.5

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.67.2. Changes

- Disallow aggregate functions in UPDATE commands, except within sub-SELECTs (Tom)
  The behavior of such an aggregate was unpredictable, and in 8.1.X could cause a crash, so it has been disabled. The SQL standard does not allow this either.
- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix core dump in duration logging for extended query protocol when a COMMIT or ROLLBACK is executed
- Fix mishandling of AFTER triggers when query contains a SQL function returning multiple rows (Tom)
- Fix `ALTER TABLE ... TYPE` to recheck NOT NULL for USING clause (Tom)
- Fix `string_to_array()` to handle overlapping matches for the separator string
  For example, `string_to_array('123xx456xxx789', 'xx')`.
- Fix `to_timestamp()` for AM/PM formats (Bruce)
- Fix autovacuum’s calculation that decides whether ANALYZE is needed (Alvaro)
- Fix corner cases in pattern matching for psql’s \d commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Numerous robustness fixes in ecpg (Joachim Wieland)
Appendix E. Release Notes

• Fix backslash escaping in /contrib/dbmirror
• Minor fixes in /contrib/dblink and /contrib/tsearch2
• Efficiency improvements in hash tables and bitmap index scans (Tom)
• Fix instability of statistics collection on Windows (Tom, Andrew)
• Fix statement_timeout to use the proper units on Win32 (Bruce)
  In previous Win32 8.1.X versions, the delay was off by a factor of 100.
• Fixes for MSVC and Borland C++ compilers (Hiroshi Saito)
• Fixes for AIX and Intel compilers (Tom)
• Fix rare bug in continuous archiving (Tom)

E.68. Release 8.1.4

Release date: 2006-05-23

This release contains a variety of fixes from 8.1.3, including patches for extremely serious security issues. For information about new features in the 8.1 major release, see Section E.72.

E.68.1. Migration to Version 8.1.4

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq’s PQescapeStringConn()) to perform string escaping, rather than relying on ad hoc code to do it.

E.68.2. Changes

• Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)
  While PostgreSQL has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.
• Reject unsafe uses of \’ in string literals
  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts ” and not \’ as a representation of ASCII single quote in SQL string literals. By default, \’ is rejected only when client_encoding is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter backslash_quote is available to adjust this behavior when needed.
Appendix E. Release Notes

Note that full security against CVE-2006-2314 might require client-side changes; the purpose of backslash_quote is in part to make it obvious that insecure clients are insecure.

- Modify libpq’s string-escaping routines to be aware of encoding considerations and standard_conforming_strings

This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple PostgreSQL connections concurrently should migrate to PQescapeStringConn() and PQescapeByteaConn() to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping “by hand” should be modified to rely on library routines instead.

- Fix weak key selection in pgcrypto (Marko Kreen)

Errors in fortuna PRNG reseeding logic could cause a predictable session key to be selected by pgp_sym_encrypt() in some cases. This only affects non-OpenSSL-using builds.

- Fix some incorrect encoding conversion functions

  win1251_to_iso, win866_to_iso, euc_tw_to_big5, euc_tw_to_mic, mic_to_euc_tw were all broken to varying extents.

- Clean up stray remaining uses of \’ in strings (Bruce, Jan)

- Make autovacuum visible in pg_stat_activity (Alvaro)

- Disable full_page_writes (Tom)

  In certain cases, having full_page_writes off would cause crash recovery to fail. A proper fix will appear in 8.2; for now it’s just disabled.

- Various planner fixes, particularly for bitmap index scans and MIN/MAX optimization (Tom)

- Fix incorrect optimization in merge join (Tom)

  Outer joins could sometimes emit multiple copies of unmatched rows.

- Fix crash from using and modifying a plpgsql function in the same transaction

- Fix WAL replay for case where a B-Tree index has been truncated

- Fix SIMILAR TO for patterns involving | (Tom)

- Fix SELECT INTO and CREATE TABLE AS to create tables in the default tablespace, not the base directory (Kris Jurka)

- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)

- Improve qsort performance (Dann Corbit)

  Currently this code is only used on Solaris.

- Fix for OS/X Bonjour on x86 systems (Ashley Clark)

- Fix various minor memory leaks

- Fix problem with password prompting on some Win32 systems (Robert Kinberg)

- Improve pg_dump’s handling of default values for domains

- Fix pg_dumpall to handle identically-named users and groups reasonably (only possible when dumping from a pre-8.1 server) (Tom)

  The user and group will be merged into a single role with LOGIN permission. Formerly the merged role wouldn’t have LOGIN permission, making it unusable as a user.

- Fix pg_restore -n to work as documented (Tom)
Appendix E. Release Notes

E.69. Release 8.1.3

Release date: 2006-02-14

This release contains a variety of fixes from 8.1.2, including one very serious security issue. For information about new features in the 8.1 major release, see Section E.72.

E.69.1. Migration to Version 8.1.3

A dump/restore is not required for those running 8.1.X. However, if you are upgrading from a version earlier than 8.1.2, see the release notes for 8.1.2.

E.69.2. Changes

- Fix bug that allowed any logged-in user to \texttt{SET ROLE} to any other database user id (CVE-2006-0553)
  
  Due to inadequate validity checking, a user could exploit the special case that \texttt{SET ROLE} normally uses to restore the previous role setting after an error. This allowed ordinary users to acquire superuser status, for example. The escalation-of-privilege risk exists only in 8.1.0-8.1.2. However, in all releases back to 7.3 there is a related bug in \texttt{SET SESSION AUTHORIZATION} that allows unprivileged users to crash the server, if it has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.

- Fix bug with row visibility logic in self-inserted rows (Tom)

  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 8.0.4, 7.4.9, and 7.3.11 releases.

- Fix race condition that could lead to “file already exists” errors during pg_clog and pg_subtrans file creation (Tom)

- Fix cases that could lead to crashes if a cache-invalidation message arrives at just the wrong time (Tom)

- Properly check \texttt{DOMAIN} constraints for \texttt{UNKNOWN} parameters in prepared statements (Neil)

- Ensure \texttt{ALTER COLUMN TYPE} will process \texttt{FOREIGN KEY}, \texttt{UNIQUE}, and \texttt{PRIMARY KEY} constraints in the proper order (Nakano Yoshihisa)

- Fixes to allow restoring dumps that have cross-schema references to custom operators or operator classes (Tom)

- Allow pg_restore to continue properly after a \texttt{COPY} failure; formerly it tried to treat the remaining \texttt{COPY} data as SQL commands (Stephen Frost)

- Fix pg_ctl unregister crash when the data directory is not specified (Magnus)

- Fix libpq \texttt{PQprint} HTML tags (Christoph Zwerschke)

- Fix ecpg crash on AMD64 and PPC (Neil)

- Allow \texttt{SETOF} and \texttt{\$TYPE} to be used together in function result type declarations

- Recover properly if error occurs during argument passing in PL/python (Neil)
E.70. Release 8.1.2

Release date: 2006-01-09

This release contains a variety of fixes from 8.1.1. For information about new features in the 8.1 major release, see Section E.72.

E.70.1. Migration to Version 8.1.2

A dump/restore is not required for those running 8.1.X. However, you might need to REINDEX indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.70.2. Changes

- Fix Windows code so that postmaster will continue rather than exit if there is no more room in ShmemBackendArray (Magnus)
  The previous behavior could lead to a denial-of-service situation if too many connection requests arrive close together. This applies only to the Windows port.
- Fix bug introduced in 8.0 that could allow ReadBuffer to return an already-used page as new, potentially causing loss of recently-committed data (Tom)
- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)
- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)
  This might require REINDEX to fix existing indexes on textual columns.
- Set locale environment variables during postmaster startup to ensure that plperl won’t change the locale later
This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need \texttt{REINDEX} to fix existing indexes on textual columns if this has happened to you.

- Allow more flexible relocation of installation directories (Tom)
  
  Previous releases supported relocation only if all installation directory paths were the same except for the last component.

- Prevent crashes caused by the use of \texttt{ISO-8859-5} and \texttt{ISO-8859-9} encodings (Tatsuo)

- Fix longstanding bug in \texttt{strpos()} and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)

- Fix bug where \texttt{COPY CSV} mode considered any `\.` to terminate the copy data
  
  The new code requires `\.` to appear alone on a line, as per documentation.

- Make \texttt{COPY CSV} mode quote a literal data value of `\.` to ensure it cannot be interpreted as the end-of-data marker (Bruce)

- Various fixes for functions returning \texttt{RECORD}s (Tom)

- Fix processing of \texttt{postgresql.conf} so a final line with no newline is processed properly (Tom)

- Fix bug in \texttt{/contrib/pgcrypto gen_salt}, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)
  
  Salts for Blowfish and standard DES are unaffected.

- Fix autovacuum crash when processing expression indexes

- Fix \texttt{/contrib/dblink} to throw an error, rather than crashing, when the number of columns specified is different from what’s actually returned by the query (Joe)

\section*{E.71. Release 8.1.1}

\textbf{Release date:} 2005-12-12

This release contains a variety of fixes from 8.1.0. For information about new features in the 8.1 major release, see Section E.72.

\subsection*{E.71.1. Migration to Version 8.1.1}

A dump/restore is not required for those running 8.1.X.

\subsection*{E.71.2. Changes}

- Fix incorrect optimizations of outer-join conditions (Tom)

- Fix problems with wrong reported column names in cases involving sub-selects flattened by the optimizer (Tom)
• Fix update failures in scenarios involving CHECK constraints, toasted columns, and indexes (Tom)
• Fix bgwriter problems after recovering from errors (Tom)
The background writer was found to leak buffer pins after write errors. While not fatal in itself, this might lead to mysterious blockages of later VACUUM commands.
• Prevent failure if client sends Bind protocol message when current transaction is already aborted
• /contrib/tsearch2 and /contrib/ltree fixes (Teodor)
• Fix problems with translated error messages in languages that require word reordering, such as Turkish; also problems with unexpected truncation of output strings and wrong display of the smallest possible bigint value (Andrew, Tom)
  These problems only appeared on platforms that were using our port/snprintf.c code, which includes BSD variants if --enable-nls was given, and perhaps others. In addition, a different form of the translated-error-message problem could appear on Windows depending on which version of libintl was used.
• Re-allow AM/PM, HH, HH12, and D format specifiers for to_char(time) and to_char(interval). (to_char(interval) should probably use HH24.) (Bruce)
• AIX, HPUX, and MSVC compile fixes (Tom, Hiroshi Saito)
• Optimizer improvements (Tom)
• Retry file reads and writes after Windows NO_SYSTEM_RESOURCES error (Qingqing Zhou)
• Prevent autovacuum from crashing during ANALYZE of expression index (Alvaro)
• Fix problems with ON COMMIT DELETE ROWS temp tables
• Fix problems when a trigger alters the output of a SELECT DISTINCT query
• Add 8.1.0 release note item on how to migrate invalid UTF-8 byte sequences (Paul Lindner)

E.72. Release 8.1

Release date: 2005-11-08

E.72.1. Overview

Major changes in this release:

Improve concurrent access to the shared buffer cache (Tom)

Access to the shared buffer cache was identified as a significant scalability problem, particularly on multi-CPU systems. In this release, the way that locking is done in the buffer manager has been overhauled to reduce lock contention and improve scalability. The buffer manager has also been changed to use a “clock sweep” replacement policy.

Allow index scans to use an intermediate in-memory bitmap (Tom)

In previous releases, only a single index could be used to do lookups on a table. With this feature, if a query has WHERE tab.col1 = 4 and tab.col2 = 9, and there is no multicolunm index on col1 and col2, but there is an index on col1 and another on col2, it is possible to
search both indexes and combine the results in memory, then do heap fetches for only the rows matching both the col1 and col2 restrictions. This is very useful in environments that have a lot of unstructured queries where it is impossible to create indexes that match all possible access conditions. Bitmap scans are useful even with a single index, as they reduce the amount of random access needed; a bitmap index scan is efficient for retrieving fairly large fractions of the complete table, whereas plain index scans are not.

Add two-phase commit (Heikki Linnakangas, Alvaro, Tom)

Two-phase commit allows transactions to be "prepared" on several computers, and once all computers have successfully prepared their transactions (none failed), all transactions can be committed. Even if a machine crashes after a prepare, the prepared transaction can be committed after the machine is restarted. New syntax includes `PREPARE TRANSACTION` and `COMMIT/ROLLBACK PREPARED`. A new system view `pg_prepared_xacts` has also been added.

Create a new role system that replaces users and groups (Stephen Frost)

Roles are a combination of users and groups. Like users, they can have login capability, and like groups, a role can have other roles as members. Roles basically remove the distinction between users and groups. For example, a role can:

- Have login capability (optionally)
- Own objects
- Hold access permissions for database objects
- Inherit permissions from other roles it is a member of

Once a user logs into a role, she obtains capabilities of the login role plus any inherited roles, and can use `SET ROLE` to switch to other roles she is a member of. This feature is a generalization of the SQL standard’s concept of roles. This change also replaces `pg_shadow` and `pg_group` by new role-capable catalogs `pg_authid` and `pg_auth_members`. The old tables are redefined as read-only views on the new role tables.

Automatically use indexes for MIN() and MAX() (Tom)

In previous releases, the only way to use an index for `MIN()` or `MAX()` was to rewrite the query as `SELECT col FROM tab ORDER BY col LIMIT 1`. Index usage now happens automatically.

Move `/contrib/pg_autovacuum` into the main server (Alvaro)

Integrating autovacuum into the server allows it to be automatically started and stopped in sync with the database server, and allows autovacuum to be configured from `postgresql.conf`.

Add shared row level locks using `SELECT ... FOR SHARE` (Alvaro)

While PostgreSQL’s MVCC locking allows `SELECT` to never be blocked by writers and therefore does not need shared row locks for typical operations, shared locks are useful for applications that require shared row locking. In particular this reduces the locking requirements imposed by referential integrity checks.

Add dependencies on shared objects, specifically roles (Alvaro)

This extension of the dependency mechanism prevents roles from being dropped while there are still database objects they own. Formerly it was possible to accidentally “orphan” objects by deleting their owner. While this could be recovered from, it was messy and unpleasant.

Improve performance for partitioned tables (Simon)

The new `constraint_exclusion` configuration parameter avoids lookups on child tables where constraints indicate that no matching rows exist in the child table.
This allows for a basic type of table partitioning. If child tables store separate key ranges and this is enforced using appropriate `CHECK` constraints, the optimizer will skip child table accesses when the constraint guarantees no matching rows exist in the child table.

### E.72.2. Migration to Version 8.1

A dump/restore using `pg_dump` is required for those wishing to migrate data from any previous release.

The 8.0 release announced that the `to_char()` function for intervals would be removed in 8.1. However, since no better API has been suggested, `to_char(interval)` has been enhanced in 8.1 and will remain in the server.

Observe the following incompatibilities:

- **add_missing_from is now false by default (Neil)**
  
  By default, we now generate an error if a table is used in a query without a `FROM` reference. The old behavior is still available, but the parameter must be set to `true` to obtain it.

  It might be necessary to set `add_missing_from` to `true` in order to load an existing dump file, if the dump contains any views or rules created using the implicit-`FROM` syntax. This should be a one-time annoyance, because PostgreSQL 8.1 will convert such views and rules to standard explicit-`FROM` syntax. Subsequent dumps will therefore not have the problem.

- **Cause input of a zero-length string ("") for float4/float8/oid to throw an error, rather than treating it as a zero (Neil)**

  This change is consistent with the current handling of zero-length strings for integers. The schedule for this change was announced in 8.0.

- **default_with_oids is now false by default (Neil)**

  With this option set to false, user-created tables no longer have an OID column unless `WITH OIDS` is specified in `CREATE TABLE`. Though OIDs have existed in all releases of PostgreSQL, their use is limited because they are only four bytes long and the counter is shared across all installed databases. The preferred way of uniquely identifying rows is via sequences and the `SERIAL` type, which have been supported since PostgreSQL 6.4.

- **Add E" syntax so eventually ordinary strings can treat backslashes literally (Bruce)**

  Currently PostgreSQL processes a backslash in a string literal as introducing a special escape sequence, e.g. \n or \010. While this allows easy entry of special values, it is nonstandard and makes porting of applications from other databases more difficult. For this reason, the PostgreSQL project is planning to remove the special meaning of backslashes in strings. For backward compatibility and for users who want special backslash processing, a new string syntax has been created. This new string syntax is formed by writing an E immediately preceding the single quote that starts the string, e.g. E’hi
’. While this release does not change the handling of backslashes in strings, it does add new configuration parameters to help users migrate applications for future releases:

  - **standard_conforming_strings** — does this release treat backslashes literally in ordinary strings?
  
  - **escape_string_warning** — warn about backslashes in ordinary (non-E) strings

  The `standard_conforming_strings` value is read-only. Applications can retrieve the value to know how backslashes are processed. (Presence of the parameter can also be taken as an indication that E" string syntax is supported.) In a future release, `standard_conforming_strings` will be
Appendix E. Release Notes

true, meaning backslashes will be treated literally in non-E strings. To prepare for this change, use E\" strings in places that need special backslash processing, and turn on escape_string_warning to find additional strings that need to be converted to use E\". Also, use two single-quotes (\") to embed a literal single-quote in a string, rather than the PostgreSQL-supported syntax of backslash single-quote (\'). The former is standards-conforming and does not require the use of the E\" string syntax. You can also use the $$ string syntax, which does not treat backslashes specially.

- Make REINDEX DATABASE reindex all indexes in the database (Tom)

Formerly, REINDEX DATABASE reindexed only system tables. This new behavior seems more intuitive. A new command REINDEX SYSTEM provides the old functionality of reindexing just the system tables.

- Read-only large object descriptors now obey MVCC snapshot semantics

When a large object is opened with INV_READ (and not INV_WRITE), the data read from the descriptor will now reflect a “snapshot” of the large object’s state at the time of the transaction snapshot in use by the query that called lo_open(). To obtain the old behavior of always returning the latest committed data, include INV_WRITE in the mode flags for lo_open().

- Add proper dependencies for arguments of sequence functions (Tom)

In previous releases, sequence names passed to nextval(), currval(), and setval() were stored as simple text strings, meaning that renaming or dropping a sequence used in a DEFAULT clause made the clause invalid. This release stores all newly-created sequence function arguments as internal OIDs, allowing them to track sequence renaming, and adding dependency information that prevents improper sequence removal. It also makes such DEFAULT clauses immune to schema renaming and search path changes.

Some applications might rely on the old behavior of run-time lookup for sequence names. This can still be done by explicitly casting the argument to text, for example nextval('myseq'::text).

Pre-8.1 database dumps loaded into 8.1 will use the old text-based representation and therefore will not have the features of OID-stored arguments. However, it is possible to update a database containing text-based DEFAULT clauses. First, save this query into a file, such as fixseq.sql:

```sql
SELECT 'ALTER TABLE ' ||
    pg_catalog.quote_ident(n.nspname) || '.' ||
    pg_catalog.quote_ident(c.relname) ||
' ALTER COLUMN ' || pg_catalog.quote_ident(a.attname) ||
'SET DEFAULT ' ||
    regexp_replace(d.adsrc,
        $$val\((['^']*)::text\)::regclass$$,
        $$val\1$$,
        'g') ||
';'
FROM pg_namespace n, pg_class c, pg_attribute a, pg_attrdef d
WHERE n.oid = c.relnamespace AND
c.oid = a.attrelid AND
a.attrelid = d.adrelid AND
a.attnum = d.adnum AND
d.adsrc ~ $$val\((['^']*)::text\)::regclass$$;
```

Next, run the query against a database to find what adjustments are required, like this for database db1:

```bash
psql -t -f fixseq.sql db1
```

This will show the ALTER TABLE commands needed to convert the database to the newer OID-based representation. If the commands look reasonable, run this to update the database:

```bash
psql -t -f fixseq.sql db1 | psql -e db1
```
This process must be repeated in each database to be updated.

- In psql, treat unquoted `{digit}+` sequences as octal (Bruce)
  In previous releases, `{digit}+` sequences were treated as decimal, and only `\0{digit}+` were treated as octal. This change was made for consistency.
- Remove grammar productions for prefix and postfix `%` and `^` operators (Tom)
  These have never been documented and complicated the use of the modulus operator (\%) with negative numbers.
- Make `≤` and `≥` for polygons consistent with the box "over" operators (Tom)
- `CREATE LANGUAGE` can ignore the provided arguments in favor of information from `pg_pltemplate` (Tom)
  A new system catalog `pg_pltemplate` has been defined to carry information about the preferred definitions of procedural languages (such as whether they have validator functions). When an entry exists in this catalog for the language being created, `CREATE LANGUAGE` will ignore all its parameters except the language name and instead use the catalog information. This measure was taken because of increasing problems with obsolete language definitions being loaded by old dump files. As of 8.1, `pg_dump` will dump procedural language definitions as just `CREATE LANGUAGE name`, relying on a template entry to exist at load time. We expect this will be a more future-proof representation.
- Make `pg_cancel_backend(int)` return a boolean rather than an integer (Neil)
- Some users are having problems loading UTF-8 data into 8.1.X. This is because previous versions allowed invalid UTF-8 byte sequences to be entered into the database, and this release properly accepts only valid UTF-8 sequences. One way to correct a dumpfile is to run the command `iconv -c -f UTF-8 -t UTF-8 -o cleanfile.sql dumpfile.sql`. The `-c` option removes invalid character sequences. A diff of the two files will show the sequences that are invalid. `iconv` reads the entire input file into memory so it might be necessary to use split to break up the dump into multiple smaller files for processing.

E.72.3. Additional Changes

Below you will find a detailed account of the additional changes between PostgreSQL 8.1 and the previous major release.

E.72.3.1. Performance Improvements

- Improve GiST and R-tree index performance (Neil)
- Improve the optimizer, including auto-resizing of hash joins (Tom)
- Overhaul internal API in several areas
- Change WAL record CRCs from 64-bit to 32-bit (Tom)
  We determined that the extra cost of computing 64-bit CRCs was significant, and the gain in reliability too marginal to justify it.
- Prevent writing large empty gaps in WAL pages (Tom)
- Improve spinlock behavior on SMP machines, particularly Opterons (Tom)
- Allow nonconsecutive index columns to be used in a multicoloumn index (Tom)
For example, this allows an index on columns a, b, c to be used in a query with
\[\text{WHERE } a = 4 \text{ and } c = 10\].

- Skip WAL logging for `CREATE TABLE AS / SELECT INTO` (Simon)

Since a crash during `CREATE TABLE AS` would cause the table to be dropped during recovery,
there is no reason to WAL log as the table is loaded. (Logging still happens if WAL archiving is
enabled, however.)

- Allow concurrent GiST index access (Teodor, Oleg)

- Add configuration parameter `full_page_writes` to control writing full pages to WAL (Bruce)

  To prevent partial disk writes from corrupting the database, PostgreSQL writes a complete copy of
each database disk page to WAL the first time it is modified after a checkpoint. This option turns off
that functionality for more speed. This is safe to use with battery-backed disk caches where partial
page writes cannot happen.

- Use `O_DIRECT` if available when using `O_SYNC` for `wal_sync_method` (Itagaki Takahiro)

  `O_DIRECT` causes disk writes to bypass the kernel cache, and for WAL writes, this improves
  performance.

- Improve `COPY` FROM performance (Alon Goldshuv)

  This was accomplished by reading `COPY` input in larger chunks, rather than character by character.

- Improve the performance of `COUNT()`, `SUM`, `AVG()`, `STDDEV()`, and `VARIANCE()` (Neil, Tom)

### E.72.3.2. Server Changes

- Prevent problems due to transaction ID (XID) wraparound (Tom)

  The server will now warn when the transaction counter approaches the wraparound point. If the
counter becomes too close to wraparound, the server will stop accepting queries. This ensures that
data is not lost before needed vacuuming is performed.

- Fix problems with object IDs (OIDs) conflicting with existing system objects after the OID counter
  has wrapped around (Tom)

- Add warning about the need to increase `max_fsm_relations` and `max_fsm_pages` during
  `VACUUM` (Ron Mayer)

- Add `temp_buffers` configuration parameter to allow users to determine the size of the local buffer
  area for temporary table access (Tom)

- Add session start time and client IP address to `pg_stat_activity` (Magnus)

- Adjust `pg_stat` views for bitmap scans (Tom)

  The meanings of some of the fields have changed slightly.

- Enhance `pg_locks` view (Tom)

- Log queries for client-side `PREPARE` and `EXECUTE` (Simon)

- Allow Kerberos name and user name case sensitivity to be specified in `postgresql.conf` (Magnus)

- Add configuration parameter `krb_server_hostname` so that the server host name can be specified
  as part of service principal (Todd Kover)

  If not set, any service principal matching an entry in the keytab can be used. This is new Kerberos
  matching behavior in this release.
Appendix E. Release Notes

- Add log_line_prefix options for millisecond timestamps (%m) and remote host (%h) (Ed L.)
- Add WAL logging for GiST indexes (Teodor, Oleg)
  GiST indexes are now safe for crash and point-in-time recovery.
- Remove old *.backup files when we do pg_stop_backup() (Bruce)
  This prevents a large number of *.backup files from existing in pg_xlog/.
- Add configuration parameters to control TCP/IP keep-alive times for idle, interval, and count (Oliver Jowett)
  These values can be changed to allow more rapid detection of lost client connections.
- Add per-user and per-database connection limits (Petr Jelinek)
  Using ALTER USER and ALTER DATABASE, limits can now be enforced on the maximum number of sessions that can concurrently connect as a specific user or to a specific database. Setting the limit to zero disables user or database connections.
- Allow more than two gigabytes of shared memory and per-backend work memory on 64-bit machines (Koichi Suzuki)
- New system catalog pg_pltemplate allows overriding obsolete procedural-language definitions in dump files (Tom)

E.72.3.3. Query Changes

- Add temporary views (Koju Iijima, Neil)
- Fix HAVING without any aggregate functions or GROUP BY so that the query returns a single group (Tom)
  Previously, such a case would treat the HAVING clause the same as a WHERE clause. This was not per spec.
- Add USING clause to allow additional tables to be specified to DELETE (Euler Taveira de Oliveira, Neil)
  In prior releases, there was no clear method for specifying additional tables to be used for joins in a DELETE statement. UPDATE already has a FROM clause for this purpose.
- Add support for \x hex escapes in backend and ecpg strings (Bruce)
  This is just like the standard C \x escape syntax. Octal escapes were already supported.
- Add BETWEEN SYMMETRIC query syntax (Pavel Stehule)
  This feature allows BETWEEN comparisons without requiring the first value to be less than the second. For example, 2 BETWEEN [ASYMMETRIC] 3 AND 1 returns false, while 2 BETWEEN SYMMETRIC 3 AND 1 returns true. BETWEEN ASYMMETRIC was already supported.
- Add NOWAIT option to SELECT ... FOR UPDATE/SHARE (Hans-Juergen Schoenig)
  While the statement_timeout configuration parameter allows a query taking more than a certain amount of time to be cancelled, the NOWAIT option allows a query to be canceled as soon as a SELECT ... FOR UPDATE/SHARE command cannot immediately acquire a row lock.
E.72.3.4. Object Manipulation Changes

- Track dependencies of shared objects (Alvaro)
  PostgreSQL allows global tables (users, databases, tablespaces) to reference information in multiple databases. This addition adds dependency information for global tables, so, for example, user ownership can be tracked across databases, so a user who owns something in any database can no longer be removed. Dependency tracking already existed for database-local objects.

- Allow limited ALTER OWNER commands to be performed by the object owner (Stephen Frost)
  Prior releases allowed only superusers to change object owners. Now, ownership can be transferred if the user executing the command owns the object and would be able to create it as the new owner (that is, the user is a member of the new owning role and that role has the CREATE permission that would be needed to create the object afresh).

- Add ALTER object SET SCHEMA capability for some object types (tables, functions, types) (Bernd Helmle)
  This allows objects to be moved to different schemas.

- Add ALTER TABLE ENABLE/DISABLE TRIGGER to disable triggers (Satoshi Nagayasu)

E.72.3.5. Utility Command Changes

- Allow TRUNCATE to truncate multiple tables in a single command (Alvaro)
  Because of referential integrity checks, it is not allowed to truncate a table that is part of a referential integrity constraint. Using this new functionality, TRUNCATE can be used to truncate such tables, if both tables involved in a referential integrity constraint are truncated in a single TRUNCATE command.

- Properly process carriage returns and line feeds in COPY CSV mode (Andrew)
  In release 8.0, carriage returns and line feeds in CSV COPY TO were processed in an inconsistent manner. (This was documented on the TODO list.)

- Add COPY WITH CSV HEADER to allow a header line as the first line in COPY (Andrew)
  This allows handling of the common CSV usage of placing the column names on the first line of the data file. For COPY TO, the first line contains the column names, and for COPY FROM, the first line is ignored.

- On Windows, display better sub-second precision in EXPLAIN ANALYZE (Magnus)

- Add trigger duration display to EXPLAIN ANALYZE (Tom)
  Prior releases included trigger execution time as part of the total execution time, but did not show it separately. It is now possible to see how much time is spent in each trigger.

- Add support for \x hex escapes in COPY (Sergey Ten)
  Previous releases only supported octal escapes.

- Make SHOW ALL include variable descriptions (Matthias Schmidt)
  SHOW varname still only displays the variable’s value and does not include the description.

- Make initdb create a new standard database called postgres, and convert utilities to use postgres rather than template1 for standard lookups (Dave)
In prior releases, `template1` was used both as a default connection for utilities like `createuser`, and as a template for new databases. This caused `CREATE DATABASE` to sometimes fail, because a new database cannot be created if anyone else is in the template database. With this change, the default connection database is now `postgres`, meaning it is much less likely someone will be using `template1` during `CREATE DATABASE`.

- Create new `reindexdb` command-line utility by moving `/contrib/reindexdb` into the server (Euler Taveira de Oliveira)

### E.72.3.6. Data Type and Function Changes

- Add `MAX()` and `MIN()` aggregates for array types (Koju Iijima)
- Fix `to_date()` and `to_timestamp()` to behave reasonably when `CC` and `YY` fields are both used (Karel Zak)
  
  If the format specification contains `CC` and a year specification is `YYY` or longer, ignore the `CC`. If the year specification is `YY` or shorter, interpret `CC` as the previous century.

- Add `md5(bytea)` (Abhijit Menon-Sen)
  
  `md5(text)` already existed.

- Add support for `numeric ^ numeric` based on `power(numeric, numeric)`
  
  The function already existed, but there was no operator assigned to it.

- Fix `NUMERIC` modulus by properly truncating the quotient during computation (Bruce)
  
  In previous releases, modulus for large values sometimes returned negative results due to rounding of the quotient.

- Add a function `lastval()` (Dennis Björklund)
  
  `lastval()` is a simplified version of `currval()`. It automatically determines the proper sequence name based on the most recent `nextval()` or `setval()` call performed by the current session.

- Add `to_timestamp(DOUBLE PRECISION)` (Michael Glaesemann)

- Add the full use of time zone names in `AT TIME ZONE`, not just the short list previously available (Magnus)
  
  Previously, only a predefined list of time zone names were supported by `AT TIME ZONE`. Now any supported time zone name can be used, e.g.:
  
  ```sql
  SELECT CURRENT_TIMESTAMP AT TIME ZONE 'Europe/London';
  ```
  
  In the above query, the time zone used is adjusted based on the daylight saving time rules that were in effect on the supplied date.

- Add `GREATEST()` and `LEAST()` variadic functions (Pavel Stehule)
  
  These functions take a variable number of arguments and return the greatest or least value among the arguments.

- Add `pg_column_size()` (Mark Kirkwood)
  
  This returns storage size of a column, which might be compressed.

- Add `regexp_replace()` (Atsushi Ogawa)
This allows regular expression replacement, like sed. An optional flag argument allows selection of global (replace all) and case-insensitive modes.

- Fix interval division and multiplication (Bruce)
  
  Previous versions sometimes returned unjustified results, like ‘4 months’::interval / 5 returning ‘1 mon -6 days’.

- Fix roundoff behavior in timestamp, time, and interval output (Tom)
  
  This fixes some cases in which the seconds field would be shown as 60 instead of incrementing the higher-order fields.

- Add a separate day field to type interval so a one day interval can be distinguished from a 24 hour interval (Michael Glaesemann)
  
  Days that contain a daylight saving time adjustment are not 24 hours long, but typically 23 or 25 hours. This change creates a conceptual distinction between intervals of “so many days” and intervals of “so many hours”. Adding 1 day to a timestamp now gives the same local time on the next day even if a daylight saving time adjustment occurs between, whereas adding 24 hours will give a different local time when this happens. For example, under US DST rules:

  ‘2005-04-03 00:00:00-05’ + ‘1 day’ = ‘2005-04-04 00:00:00-04’
  ‘2005-04-03 00:00:00-05’ + ‘24 hours’ = ‘2005-04-04 01:00:00-04’

- Add justify_days() and justify_hours() (Michael Glaesemann)
  
  These functions, respectively, adjust days to an appropriate number of full months and days, and adjust hours to an appropriate number of full days and hours.

- Move /contrib/dbsize into the backend, and rename some of the functions (Dave Page, Andreas Pflug)

  - pg_tablespace_size()
  - pg_database_size()
  - pg_relation_size()
  - pg_total_relation_size()
  - pg_size_pretty()

  pg_total_relation_size() includes indexes and TOAST tables.

- Add functions for read-only file access to the cluster directory (Dave Page, Andreas Pflug)

  - pg_stat_file()
  - pg_read_file()
  - pg_ls_dir()

- Add pg_reload_conf() to force reloading of the configuration files (Dave Page, Andreas Pflug)

- Add pg_rotate_logfile() to force rotation of the server log file (Dave Page, Andreas Pflug)

- Change pg_stat_* views to include TOAST tables (Tom)
E.72.3.7. Encoding and Locale Changes

- Rename some encodings to be more consistent and to follow international standards (Bruce)
  - UNICODE is now UTF8
  - ALT is now WIN866
  - WIN is now WIN1251
  - TCVN is now WIN1258

The original names still work.
- Add support for WIN1252 encoding (Roland Volkmann)
- Add support for four-byte UTF8 characters (John Hansen)
  Previously only one, two, and three-byte UTF8 characters were supported. This is particularly important for support for some Chinese character sets.
- Allow direct conversion between EUC_JP and SJIS to improve performance (Atsushi Ogawa)
- Allow the UTF8 encoding to work on Windows (Magnus)
  This is done by mapping UTF8 to the Windows-native UTF16 implementation.

E.72.3.8. General Server-Side Language Changes

- Fix ALTER LANGUAGE RENAME (Sergey Yatskevich)
- Allow function characteristics, like strictness and volatility, to be modified via ALTER FUNCTION (Neil)
- Increase the maximum number of function arguments to 100 (Tom)
- Allow SQL and PL/pgSQL functions to use OUT and INOUT parameters (Tom)
  OUT is an alternate way for a function to return values. Instead of using RETURN, values can be returned by assigning to parameters declared as OUT or INOUT. This is notationally simpler in some cases, particularly so when multiple values need to be returned. While returning multiple values from a function was possible in previous releases, this greatly simplifies the process. (The feature will be extended to other server-side languages in future releases.)
- Move language handler functions into the pg_catalog schema
  This makes it easier to drop the public schema if desired.
- Add SPI_getnspname() to SPI (Neil)

E.72.3.9. PL/pgSQL Server-Side Language Changes

- Overhaul the memory management of PL/pgSQL functions (Neil)
  The parsetree of each function is now stored in a separate memory context. This allows this memory to be easily reclaimed when it is no longer needed.
- Check function syntax at CREATE FUNCTION time, rather than at runtime (Neil)
Previously, most syntax errors were reported only when the function was executed.

- **Allow** `OPEN` to open non-`SELECT` queries like `EXPLAIN` and `SHOW` (Tom)
- No longer require functions to issue a `RETURN` statement (Tom)
  This is a byproduct of the newly added `OUT` and `INOUT` functionality. `RETURN` can be omitted when it is not needed to provide the function’s return value.
- Add support for an optional `INTO` clause to PL/pgSQL’s `EXECUTE` statement (Pavel Stehule, Neil)
- Make `CREATE TABLE AS` set `ROW_COUNT` (Tom)
- Define `SQLSTATE` and `SQLERRM` to return the `SQLSTATE` and error message of the current exception (Pavel Stehule, Neil)
  These variables are only defined inside exception blocks.
- Allow the parameters to the `RAISE` statement to be expressions (Pavel Stehule, Neil)
- Add a loop `CONTINUE` statement (Pavel Stehule, Neil)
- Allow block and loop labels (Pavel Stehule)

**E.72.3.10. PL/Perl Server-Side Language Changes**

- Allow large result sets to be returned efficiently (Abhijit Menon-Sen)
  This allows functions to use `return_next()` to avoid building the entire result set in memory.
- Allow one-row-at-a-time retrieval of query results (Abhijit Menon-Sen)
  This allows functions to use `spi_query()` and `spi_fetchrow()` to avoid accumulating the entire result set in memory.
- Force PL/Perl to handle strings as `UTF8` if the server encoding is `UTF8` (David Kamholz)
- Add a validator function for PL/Perl (Andrew)
  This allows syntax errors to be reported at definition time, rather than execution time.
- Allow PL/Perl to return a Perl array when the function returns an array type (Andrew)
  This basically maps PostgreSQL arrays to Perl arrays.
- Allow Perl nonfatal warnings to generate `NOTICE` messages (Andrew)
- Allow Perl’s `strict` mode to be enabled (Andrew)

**E.72.3.11. psql Changes**

- Add `\set ON_ERROR_ROLLBACK` to allow statements in a transaction to error without affecting the rest of the transaction (Greg Sabino Mullane)
  This is basically implemented by wrapping every statement in a sub-transaction.
- Add support for `\x` hex strings in `psql` variables (Bruce)
  Octal escapes were already supported.
- Add support for `troff -ms` output format (Roger Leigh)
- Allow the history file location to be controlled by `HISTFILE` (Andreas Seltenreich)
  This allows configuration of per-database history storage.
Appendix E. Release Notes

- Prevent \x (expanded mode) from affecting the output of \d tablename (Neil)
- Add \-L option to psql to log sessions (Lorne Sunley)
  This option was added because some operating systems do not have simple command-line activity logging functionality.
- Make \d show the tablespaces of indexes (Qingqing Zhou)
- Allow psql help (\h) to make a best guess on the proper help information (Greg Sabino Mullane)
  This allows the user to just add \h to the front of the syntax error query and get help on the supported syntax. Previously any additional query text beyond the command name had to be removed to use \h.
- Add \pset numericlocale to allow numbers to be output in a locale-aware format (Eugen Nedelcu)
  For example, using C locale 100000 would be output as 100,000.0 while a European locale might output this value as 100.000,0.
- Make startup banner show both server version number and psql’s version number, when they are different (Bruce)
  Also, a warning will be shown if the server and psql are from different major releases.

E.72.3.12. pg_dump Changes

- Add --schema switch to pg_restore (Richard van den Berg)
  This allows just the objects in a specified schema to be restored.
- Allow pg_dump to dump large objects even in text mode (Tom)
  With this change, large objects are now always dumped; the former -b switch is a no-op.
- Allow pg_dump to dump a consistent snapshot of large objects (Tom)
- Dump comments for large objects (Tom)
- Add --encoding to pg_dump (Magnus Hagander)
  This allows a database to be dumped in an encoding that is different from the server’s encoding. This is valuable when transferring the dump to a machine with a different encoding.
- Rely on pg_pltemplate for procedural languages (Tom)
  If the call handler for a procedural language is in the pg_catalog schema, pg_dump does not dump the handler. Instead, it dumps the language using just CREATE LANGUAGE name, relying on the pg_pltemplate catalog to provide the language’s creation parameters at load time.

E.72.3.13. libpq Changes

- Add a PGPASSFILE environment variable to specify the password file’s filename (Andrew)
- Add lo_create(), that is similar to lo_creat() but allows the OID of the large object to be specified (Tom)
- Make libpq consistently return an error to the client application on malloc() failure (Neil)
E.72.3.14. Source Code Changes

- Fix pgxs to support building against a relocated installation
- Add spinlock support for the Itanium processor using Intel compiler (Vikram Kalsi)
- Add Kerberos 5 support for Windows (Magnus)
- Add Chinese FAQ (laser@pgsql.com)
- Rename Rendezvous to Bonjour to match OS/X feature renaming (Bruce)
- Add support for fsync_writethrough on Darwin (Chris Campbell)
- Streamline the passing of information within the server, the optimizer, and the lock system (Tom)
- Allow pg_config to be compiled using MSVC (Andrew)
  This is required to build DBD::Pg using MSVC.
- Remove support for Kerberos V4 (Magnus)
  Kerberos 4 had security vulnerabilities and is no longer maintained.
- Code cleanups (Coverity static analysis performed by EnterpriseDB)
- Modify postgresql.conf to use documentation defaults on/off rather than true/false (Bruce)
- Enhance pg_config to be able to report more build-time values (Tom)
- Allow libpq to be built thread-safe on Windows (Dave Page)
- Allow IPv6 connections to be used on Windows (Andrew)
- Add Server Administration documentation about I/O subsystem reliability (Bruce)
- Move private declarations from gist.h to gist_private.h (Neil)
  In previous releases, gist.h contained both the public GiST API (intended for use by authors of GiST index implementations) as well as some private declarations used by the implementation of GiST itself. The latter have been moved to a separate file, gist_private.h. Most GiST index implementations should be unaffected.
- Overhaul GiST memory management (Neil)
  GiST methods are now always invoked in a short-lived memory context. Therefore, memory allocated via palloc() will be reclaimed automatically, so GiST index implementations do not need to manually release allocated memory via pfree().

E.72.3.15. Contrib Changes

- Add /contrib/pg_buffercache contrib module (Mark Kirkwood)
  This displays the contents of the buffer cache, for debugging and performance tuning purposes.
- Remove /contrib/array because it is obsolete (Tom)
- Clean up the /contrib/lo module (Tom)
- Move /contrib/findoidjoins to /src/tools (Tom)
- Remove the <<, >>, &<, and &> operators from /contrib/cube
  These operators were not useful.
• Improve /contrib/btree_gist (Janko Richter)
• Improve /contrib/pgbench (Tomoaki Sato, Tatsuo)

There is now a facility for testing with SQL command scripts given by the user, instead of only a
hard-wired command sequence.

• Improve /contrib/pgcrypto (Marko Kreen)

   - Implementation of OpenPGP symmetric-key and public-key encryption
     Both RSA and Elgamal public-key algorithms are supported.
   - Stand alone build: include SHA256/384/512 hashes, Fortuna PRNG
   - OpenSSL build: support 3DES, use internal AES with OpenSSL < 0.9.7
   - Take build parameters (OpenSSL, zlib) from configure result
     There is no need to edit the Makefile anymore.
   - Remove support for libmhash and libmcrypt

E.73. Release 8.0.26

Release date: 2010-10-04

This release contains a variety of fixes from 8.0.25. For information about new features in the 8.0
major release, see Section E.99.

This is expected to be the last PostgreSQL release in the 8.0.X series. Users are encouraged to update
to a newer release branch soon.

E.73.1. Migration to Version 8.0.26

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version
earlier than 8.0.22, see the release notes for 8.0.22.

E.73.2. Changes

• Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

This change prevents security problems that can be caused by subverting Perl or Tcl code that
will be executed later in the same session under another SQL user identity (for example, within a
SECURITY DEFINER function). Most scripting languages offer numerous ways that that might be
done, such as redefining standard functions or operators called by the target function. Without this
change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the
SQL privileges of the target function’s owner.

The cost of this change is that intentional communication among Perl and Tcl functions becomes
more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only
Appendix E. Release Notes

one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).

• Prevent possible crashes in `pg_get_expr()` by disallowing it from being called with an argument that is not one of the system catalog columns it’s intended to be used with (Heikki Linnakangas, Tom Lane)

• Fix “cannot handle unplanned sub-select” error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.

• Defend against functions returning setof record where not all the returned rows are actually of the same rowtype (Tom Lane)

• Take care to fsync the contents of lockfiles (both `postmaster.pid` and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.

• Avoid recursion while assigning XIDs to heavily-nested subtransactions (Andres Freund, Robert Haas)

  The original coding could result in a crash if there was limited stack space.

• Fix `log_line_prefix`'s `%i` escape, which could produce junk early in backend startup (Tom Lane)

• Fix possible data corruption in `ALTER TABLE ... SET TABLESPACE` when archiving is enabled (Jeff Davis)

• Allow `CREATE DATABASE` and `ALTER DATABASE ... SET TABLESPACE` to be interrupted by query-cancel (Guillaume Lelarge)

• In PL/Python, defend against null pointer results from `PyCObject_AsVoidPtr` and `PyCObject_FromVoidPtr` (Peter Eisentraut)

• Improve contrib/dblink’s handling of tables containing dropped columns (Tom Lane)

• Fix connection leak after “duplicate connection name” errors in contrib/dblink (Itagaki Takahiro)

• Fix contrib/dblink to handle connection names longer than 62 bytes correctly (Itagaki Takahiro)

• Update build infrastructure and documentation to reflect the source code repository’s move from CVS to Git (Magnus Hagander and others)

• Update time zone data files to `tzdata` release 2010l for DST law changes in Egypt and Palestine; also historical corrections for Finland.

  This change also adds new names for two Micronesian timezones: Pacific/Chuuk is now preferred over Pacific/Truk (and the preferred abbreviation is CHUT not TRUT) and Pacific/Pohnpei is preferred over Pacific/Ponape.
E.74. Release 8.0.25

Release date: 2010-05-17

This release contains a variety of fixes from 8.0.24. For information about new features in the 8.0 major release, see Section E.99.

The PostgreSQL community will stop releasing updates for the 8.0.X release series in July 2010. Users are encouraged to update to a newer release branch soon.

E.74.1. Migration to Version 8.0.25

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.22, see the release notes for 8.0.22.

E.74.2. Changes

• Enforce restrictions in plperl using an opmask applied to the whole interpreter, instead of using Safe.pm (Tim Bunce, Andrew Dunstan)

Recent developments have convinced us that Safe.pm is too insecure to rely on for making plperl trustworthy. This change removes use of Safe.pm altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl’s strict pragma in a natural way in plperl, and that Perl’s $a and $b variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

• Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

PL/Tcl’s feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted “normal” Tcl interpreter unless we are really going to execute a pltclu function. (CVE-2010-1170)

• Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

Previously, if an unprivileged user ran ALTER USER ... RESET ALL for himself, or ALTER DATABASE ... RESET ALL for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the ALTER will only remove the parameters that the user has permission to change.

• Avoid possible crash during backend shutdown if shutdown occurs when a CONTEXT addition would be made to log entries (Tom)

In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

• Update pl/perl’s ppport.h for modern Perl versions (Andreas)

• Fix assorted memory leaks in pl/python (Andreas Freund, Tom)
• Prevent infinite recursion in psql when expanding a variable that refers to itself (Tom)

• Ensure that contrib/pgstattuple functions respond to cancel interrupts promptly (Tatsuhito Kasahara)

• Make server startup deal properly with the case that shmget() returns EINVAL for an existing shared memory segment (Tom)

  This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.

• Update time zone data files to tzdata release 2010j for DST law changes in Argentina, Australian Antarctic, Bangladesh, Mexico, Morocco, Pakistan, Palestine, Russia, Syria, Tunisia; also historical corrections for Taiwan.

E.75. Release 8.0.24

Release date: 2010-03-15

This release contains a variety of fixes from 8.0.23. For information about new features in the 8.0 major release, see Section E.99.

The PostgreSQL community will stop releasing updates for the 8.0.X release series in July 2010. Users are encouraged to update to a newer release branch soon.

E.75.1. Migration to Version 8.0.24

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.22, see the release notes for 8.0.22.

E.75.2. Changes

• Add new configuration parameter ssl_renegotiation_limit to control how often we do session key renegotiation for an SSL connection (Magnus)

  This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.

• Fix possible crashes when trying to recover from a failure in subtransaction start (Tom)

• Fix server memory leak associated with use of savepoints and a client encoding different from server’s encoding (Tom)

• Make substring() for bit types treat any negative length as meaning “all the rest of the string” (Tom)

  The previous coding treated only -1 that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).

• Fix integer-to-bit-string conversions to handle the first fractional byte correctly when the output bit width is wider than the given integer by something other than a multiple of 8 bits (Tom)
• Fix some cases of pathologically slow regular expression matching (Tom)
• Fix the STOP WAL LOCATION entry in backup history files to report the next WAL segment’s name when the end location is exactly at a segment boundary (Itagaki Takahiro)
• When reading pg_hba.conf and related files, do not treat @something as a file inclusion request if the @ appears inside quote marks; also, never treat @ by itself as a file inclusion request (Tom)
  This prevents erratic behavior if a role or database name starts with @. If you need to include a file whose path name contains spaces, you can still do so, but you must write @"/path to/file" rather than putting the quotes around the whole construct.
• Prevent infinite loop on some platforms if a directory is named as an inclusion target in pg_hba.conf and related files (Tom)
• Fix pgsql failure in one case where a composite column is set to NULL (Tom)
• Add volatile markings in PL/Python to avoid possible compiler-specific misbehavior (Zdenek Kotala)
• Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)
  The only known symptom of this oversight is that the Tcl clock command misbehaves if using Tcl 8.5 or later.
• Prevent crash in contrib/dblink when too many key columns are specified to a dblink_build_sql_* function (Rushabh Lathia, Joe Conway)
• Fix assorted crashes in contrib/xml2 caused by sloppy memory management (Tom)
• Update time zone data files to tzdata release 2010e for DST law changes in Bangladesh, Chile, Fiji, Mexico, Paraguay, Samoa.

E.76. Release 8.0.23

Release date: 2009-12-14

This release contains a variety of fixes from 8.0.22. For information about new features in the 8.0 major release, see Section E.99.

E.76.1. Migration to Version 8.0.23

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.22, see the release notes for 8.0.22.

E.76.2. Changes

• Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)
  This change prevents allegedly-immutable index functions from possibly subverting a superuser’s session (CVE-2009-4136).
• Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)
  This prevents unintended matching of a certificate to a server or client name during SSL validation
  (CVE-2009-4034).
• Fix possible crash during backend-startup-time cache initialization (Tom)
• Prevent signals from interrupting VACUUM at unsafe times (Alvaro)
  This fix prevents a PANIC if a VACUUM FULL is cancelled after it’s already committed its tuple
  movements, as well as transient errors if a plain VACUUM is interrupted after having truncated the
  table.
• Fix possible crash due to integer overflow in hash table size calculation (Tom)
  This could occur with extremely large planner estimates for the size of a hashjoin’s result.
• Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
• Fix premature drop of temporary files used for a cursor that is accessed within a subtransaction
  (Heikki)
• Fix PAM password processing to be more robust (Tom)
  The previous code is known to fail with the combination of the Linux pam_krb5 PAM module with
  Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since
  it was making unjustified assumptions about what arguments the PAM stack would pass to it.
• Fix rare crash in exception processing in PL/Python (Peter)
• Ensure psql’s flex module is compiled with the correct system header definitions (Tom)
  This fixes build failures on platforms where --enable-largefile causes incompatible changes
  in the generated code.
• Make the postmaster ignore any application_name parameter in connection request packets, to
  improve compatibility with future libpq versions (Tom)
• Update time zone data files to tzdata release 2009s for DST law changes in Antarctica, Argentina,
  Bangladesh, Fiji, Novokuznetsk, Pakistan, Palestine, Samoa, Syria; also historical corrections for
  Hong Kong.

**E.77. Release 8.0.22**

**Release date:** 2009-09-09

This release contains a variety of fixes from 8.0.21. For information about new features in the 8.0
major release, see Section E.99.

**E.77.1. Migration to Version 8.0.22**

A dump/restore is not required for those running 8.0.X. However, if you have any hash indexes on
interval columns, you must REINDEX them after updating to 8.0.22. Also, if you are upgrading
from a version earlier than 8.0.6, see the release notes for 8.0.6.
Appendix E. Release Notes

E.77.2. Changes

- Disallow **RESET ROLE** and **RESET SESSION AUTHORIZATION** inside security-definer functions (Tom, Heikki)
  
  This covers a case that was missed in the previous patch that disallowed **SET ROLE** and **SET SESSION AUTHORIZATION** inside security-definer functions. (See CVE-2007-6600)

- Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)

- Fix hash calculation for data type `interval` (Tom)

  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must **REINDEX** them after updating.

- Treat `to_char(..., 'TH')` as an uppercase ordinal suffix with ‘HH’/’HH12’ (Heikki)

  It was previously handled as ‘th’ (lowercase).

- Fix overflow for `INTERVAL ‘x ms’` when `x` is more than 2 million and integer datetimes are in use (Alex Hunsaker)

- Fix calculation of distance between a point and a line segment (Tom)

  This led to incorrect results from a number of geometric operators.

- Fix `money` data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)

- Properly round datetime input like 00:12:57.9999999999999999999999999999999999 (Tom)

- Fix poor choice of page split point in GiST R-tree operator classes (Teodor)

- Fix portability issues in plperl initialization (Andrew Dunstan)

- Fix `pg_ctl` to not go into an infinite loop if `postgresql.conf` is empty (Jeff Davis)

- Fix contrib/xml2’s `xslt_process()` to properly handle the maximum number of parameters (twenty) (Tom)

- Improve robustness of libpq’s code to recover from errors during **COPY FROM STDIN** (Tom)

- Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)

- Update time zone data files to tzdata release 2009j for DST law changes in Bangladesh, Egypt, Jordan, Pakistan, Argentina/San_Luis, Cuba, Jordan (historical correction only), Mauritius, Morocco, Palestine, Syria, Tunisia.

E.78. Release 8.0.21

**Release date:** 2009-03-16

This release contains a variety of fixes from 8.0.20. For information about new features in the 8.0 major release, see Section E.99.
E.78.1. Migration to Version 8.0.21
A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.78.2. Changes

- Prevent error recursion crashes when encoding conversion fails (Tom)
  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that any error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)
- Disallow CREATE CONVERSION with the wrong encodings for the specified conversion function (Heikki)
  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.
- Fix core dump when to_char() is given format codes that are inappropriate for the type of the data argument (Tom)
- Add MUST (Mauritius Island Summer Time) to the default list of known timezone abbreviations (Xavier Bugaud)

E.79. Release 8.0.20

Release date: 2009-02-02

This release contains a variety of fixes from 8.0.19. For information about new features in the 8.0 major release, see Section E.99.

E.79.1. Migration to Version 8.0.20
A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.79.2. Changes

- Improve handling of URLs in headline() function (Teodor)
- Improve handling of overlength headlines in headline() function (Teodor)
Appendix E. Release Notes

• Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)
• Avoid unnecessary locking of small tables in VACUUM (Heikki)
• Fix uninitialized variables in contrib/tsearch2’s get_covers() function (Teodor)
• Make all documentation reference pgsql-bugs and/or pgsql-hackers as appropriate, instead of the now-decommissioned pgsql-ports and pgsql-patches mailing lists (Tom)
• Update time zone data files to tzdata release 2009a (for Kathmandu and historical DST corrections in Switzerland, Cuba)

E.80. Release 8.0.19

Release date: 2008-11-03

This release contains a variety of fixes from 8.0.18. For information about new features in the 8.0 major release, see Section E.99.

E.80.1. Migration to Version 8.0.19

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.80.2. Changes

• Fix backend crash when the client encoding cannot represent a localized error message (Tom)
  We have addressed similar issues before, but it would still fail if the “character has no equivalent” message itself couldn’t be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.
• Fix possible crash when deeply nested functions are invoked from a trigger (Tom)
• Ensure an error is reported when a newly-defined PL/pgSQL trigger function is invoked as a normal function (Tom)
• Fix incorrect tsearch2 headline generation when single query item matches first word of text (Sushant Sinha)
• Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an --enable-integer-datetimes build (Ron Mayer)
• Ensure SPI_getvalue and SPI_getbinval behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)
  This situation is normal when a table has had columns added or removed, but these two functions didn’t handle it properly. The only likely consequence is an incorrect error indication.
• Fix ecpg’s parsing of CREATE USER (Michael)
• Fix recent breakage of pg_ctl restart (Tom)
Appendix E. Release Notes

• Update time zone data files to tzdata release 2008i (for DST law changes in Argentina, Brazil, Mauritius, Syria)

E.81. Release 8.0.18

Release date: 2008-09-22

This release contains a variety of fixes from 8.0.17. For information about new features in the 8.0 major release, see Section E.99.

E.81.1. Migration to Version 8.0.18

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.81.2. Changes

• Widen local lock counters from 32 to 64 bits (Tom)
  This responds to reports that the counters could overflow in sufficiently long transactions, leading to unexpected “lock is already held” errors.

• Add checks in executor startup to ensure that the tuples produced by an INSERT or UPDATE will match the target table’s current rowtype (Tom)

  ALTER COLUMN TYPE, followed by re-use of a previously cached plan, could produce this type of situation. The check protects against data corruption and/or crashes that could ensue.

• Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)

• Improve performance of writing very long log messages to syslog (Tom)

• Fix bug in backwards scanning of a cursor on a SELECT DISTINCT ON query (Tom)

• Fix planner to estimate that GROUP BY expressions yielding boolean results always result in two groups, regardless of the expressions’ contents (Tom)

  This is very substantially more accurate than the regular GROUP BY estimate for certain boolean tests like col IS NULL.

• Fix PL/Tcl to behave correctly with Tcl 8.5, and to be more careful about the encoding of data sent to or from Tcl (Tom)

• Fix PL/Python to work with Python 2.5

  This is a back-port of fixes made during the 8.2 development cycle.

• Improve pg_dump and pg_restore’s error reporting after failure to send a SQL command (Tom)

• Fix pg_ctl to properly preserve postmaster command-line arguments across a restart (Bruce)
• Update time zone data files to tzdata release 2008f (for DST law changes in Argentina, Bahamas, Brazil, Mauritius, Morocco, Pakistan, Palestine, and Paraguay)

E.82. Release 8.0.17

Release date: 2008-06-12

This release contains one serious bug fix over 8.0.16. For information about new features in the 8.0 major release, see Section E.99.

E.82.1. Migration to Version 8.0.17

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.82.2. Changes

• Make \texttt{pg\_get\_ruledef()} parenthesize negative constants (Tom)

Before this fix, a negative constant in a view or rule might be dumped as, say, \texttt{-42::integer}, which is subtly incorrect: it should be \texttt{(-42)::integer} due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL to reject what had been a valid \texttt{SELECT DISTINCT} view query. Since this could result in \texttt{pg\_dump} output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.

E.83. Release 8.0.16

Release date: never released

This release contains a variety of fixes from 8.0.15. For information about new features in the 8.0 major release, see Section E.99.

E.83.1. Migration to Version 8.0.16

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.
Appendix E. Release Notes

E.83.2. Changes

- Fix `ALTER TABLE ADD COLUMN ... PRIMARY KEY` so that the new column is correctly checked to see if it’s been initialized to all non-nulls (Brendan Jurd)
  Previous versions neglected to check this requirement at all.

- Fix possible `CREATE TABLE` failure when inheriting the “same” constraint from multiple parent relations that inherited that constraint from a common ancestor (Tom)

- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic “Yo” characters (є and Е with two dots) (Sergey Burladyan)

- Fix a few datatype input functions that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)
  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched `ORDER BY` and `DISTINCT` expressions.

- Fix a corner case in regular-expression substring matching `(substring(string from pattern))` (Tom)
  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn’t got a match. An example is `substring('foo' from 'foo(bar)?')`. This should return NULL, since (bar) isn’t matched, but it was mistakenly returning the whole-pattern match instead (ie, `foo`).

- Update time zone data files to tzdata release 2008c (for DST law changes in Morocco, Iraq, Choibalsan, Pakistan, Syria, Cuba, Argentina/San Luis, and Chile)

- Fix incorrect result from ecpg’s `PGTYPEStimestamp_sub()` function (Michael)

- Fix core dump in `contrib/xml2`’s `xpath_table()` function when the input query returns a NULL value (Tom)

- Fix `DatumGetBool` macro to not fail with gcc 4.3 (Tom)
  This problem affects “old style” (V0) C functions that return boolean. The fix is already in 8.3, but the need to back-patch it was not realized at the time.

- Fix longstanding `LISTEN/NOTIFY` race condition (Tom)
  In rare cases a session that had just executed a `LISTEN` might not get a notification, even though one would be expected because the concurrent transaction executing `NOTIFY` was observed to commit later.
  A side effect of the fix is that a transaction that has executed a not-yet-committed `LISTEN` command will not see any row in `pg_listener` for the `LISTEN`, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

- Fix rare crash when an error occurs during a query using a hash index (Heikki)

- Fix input of datetime values for February 29 in years BC (Tom)
  The former coding was mistaken about which years were leap years.

- Fix “unrecognized node type” error in some variants of `ALTER OWNER` (Tom)

- Fix `pg_ctl` to correctly extract the postmaster’s port number from command-line options (Itagaki Takahiro, Tom)
Previously, `pg_ctl start -w` could try to contact the postmaster on the wrong port, leading to bogus reports of startup failure.

- Use `-fwrapv` to defend against possible misoptimization in recent gcc versions (Tom)
  This is known to be necessary when building PostgreSQL with gcc 4.3 or later.

- Fix display of constant expressions in `ORDER BY` and `GROUP BY` (Tom)
  An explicitly casted constant would be shown incorrectly. This could for example lead to corruption of a view definition during dump and reload.

- Fix libpq to handle `NOTICE` messages correctly during `COPY OUT` (Tom)
  This failure has only been observed to occur when a user-defined datatype’s output routine issues a `NOTICE`, but there is no guarantee it couldn’t happen due to other causes.

## E.84. Release 8.0.15

### Release date: 2008-01-07

This release contains a variety of fixes from 8.0.14, including fixes for significant security issues. For information about new features in the 8.0 major release, see Section E.99.

This is the last 8.0.X release for which the PostgreSQL community will produce binary packages for Windows. Windows users are encouraged to move to 8.2.X or later, since there are Windows-specific fixes in 8.2.X that are impractical to back-port. 8.0.X will continue to be supported on other platforms.

### E.84.1. Migration to Version 8.0.15

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

### E.84.2. Changes

- Prevent functions in indexes from executing with the privileges of the user running `VACUUM`, `ANALYZE`, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as `VACUUM FULL`, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including `VACUUM`, `ANALYZE`, `REINDEX`, and `CLUSTER`) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for `SECURITY DEFINER` functions. To prevent bypassing this security measure, execution of `SET SESSION AUTHORIZATION` and `SET ROLE` is now forbidden within a `SECURITY DEFINER` context. (CVE-2007-6600)
Appendix E. Release Notes

- Repair assorted bugs in the regular-expression package (Tom, Will Drewry)
  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping,
  and/or massive memory consumption, all of which pose denial-of-service hazards for applications
  that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772,
  CVE-2007-6067)

- Require non-superusers who use /contrib/dblink to use only password authentication, as a
  security measure (Joe)
  The fix that appeared for this in 8.0.14 was incomplete, as it plugged the hole for only some dblink

- Update time zone data files to tzdata release 2007k (in particular, recent Argentina changes) (Tom)

- Fix planner failure in some cases of WHERE false AND var IN (SELECT ...) (Tom)

- Preserve the tablespace of indexes that are rebuilt by ALTER TABLE ... ALTER COLUMN TYPE
  (Tom)

- Make archive recovery always start a new WAL timeline, rather than only when a recovery stop
  time was used (Simon)
  This avoids a corner-case risk of trying to overwrite an existing archived copy of the last WAL
  segment, and seems simpler and cleaner than the original definition.

- Make VACUUM not use all of maintenance_work_mem when the table is too small for it to be
  useful (Alvaro)

- Fix potential crash in translate() when using a multibyte database encoding (Tom)

- Fix PL/Perl to cope when platform’s Perl defines type bool as int rather than char (Tom)
  While this could theoretically happen anywhere, no standard build of Perl did things this way ...
  until Mac OS X 10.5.

- Fix PL/Python to not crash on long exception messages (Alvaro)

- Fix pg_dump to correctly handle inheritance child tables that have default expressions different
  from their parent’s (Tom)

- ecpg parser fixes (Michael)

- Make contrib/tablefunc’s crosstab() handle NULL rowid as a category in its own right,
  rather than crashing (Joe)

- Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)

- Fix crash of to_tsvector() on huge input strings (Teodor)

- Require a specific version of Autoconf to be used when re-generating the configure script (Peter)
  This affects developers and packagers only. The change was made to prevent accidental use of
  untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if
  you really want to use a different Autoconf version, but it’s your responsibility whether the result
  works or not.

E.85. Release 8.0.14

Release date: 2007-09-17
Appendix E. Release Notes

This release contains a variety of fixes from 8.0.13. For information about new features in the 8.0 major release, see Section E.99.

E.85.1. Migration to Version 8.0.14

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.85.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent VACUUM on the same table (Tom)
- Make CREATE DOMAIN ... DEFAULT NULL work properly (Tom)
- Fix excessive logging of SSL error messages (Tom)
- Fix logging so that log messages are never interleaved when using the syslogger process (Andrew)
- Fix crash when log_min_error_statement logging runs out of memory (Tom)
- Fix incorrect handling of some foreign-key corner cases (Tom)
- Prevent CLUSTER from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Update the time zone database rules, particularly New Zealand’s upcoming changes (Tom)
- Windows socket improvements (Magnus)
- Suppress timezone name (%Z) in log timestamps on Windows because of possible encoding mismatches (Tom)
- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

E.86. Release 8.0.13

Release date: 2007-04-23

This release contains a variety of fixes from 8.0.12, including a security fix. For information about new features in the 8.0 major release, see Section E.99.

E.86.1. Migration to Version 8.0.13

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.
E.86.2. Changes

- Support explicit placement of the temporary-table schema within search_path, and disable searching it for functions and operators (Tom)
  
  This is needed to allow a security-definer function to set a truly secure value of search_path. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See CREATE FUNCTION for more information.

- /contrib/tsearch2 crash fixes (Teodor)

- Fix potential-data-corruption bug in how VACUUM FULL handles UPDATE chains (Tom, Pavan Deolasee)

- Fix PANIC during enlargement of a hash index (bug introduced in 8.0.10) (Tom)

- Fix POSIX-style timezone specs to follow new USA DST rules (Tom)

E.87. Release 8.0.12

Release date: 2007-02-07

This release contains one fix from 8.0.11. For information about new features in the 8.0 major release, see Section E.99.

E.87.1. Migration to Version 8.0.12

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.87.2. Changes

- Remove overly-restrictive check for type length in constraints and functional indexes (Tom)

E.88. Release 8.0.11

Release date: 2007-02-05

This release contains a variety of fixes from 8.0.10, including a security fix. For information about new features in the 8.0 major release, see Section E.99.
E.88.1. Migration to Version 8.0.11

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.88.2. Changes

- Remove security vulnerabilities that allowed connected users to read backend memory (Tom)
  The vulnerabilities involve suppressing the normal check that a SQL function returns the data type it’s declared to, and changing the data type of a table column (CVE-2007-0555, CVE-2007-0556). These errors can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Fix for rare Assert() crash triggered by UNION (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.89. Release 8.0.10

Release date: 2007-01-08

This release contains a variety of fixes from 8.0.9. For information about new features in the 8.0 major release, see Section E.99.

E.89.1. Migration to Version 8.0.10

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.89.2. Changes

- Improve handling of getaddrinfo() on AIX (Tom)
  This fixes a problem with starting the statistics collector, among other things.
- Fix “failed to re-find parent key” errors in VACUUM (Tom)
- Fix race condition for truncation of a large relation across a gigabyte boundary by VACUUM (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix possible deadlock in Windows signal handling (Teodor)
- Fix error when constructing an ARRAY[] made up of multiple empty elements (Tom)
Appendix E. Release Notes

- Fix ecpg memory leak during connection (Michael)
- `to_number()` and `to_char(numeric)` are now **STABLE, not IMMUTABLE**, for new initdb installs (Tom)
  This is because `lc_numeric` can potentially change the output of these functions.
- Improve index usage of regular expressions that use parentheses (Tom)
  This improves `psql` \d performance also.
- Update timezone database
  This affects Australian and Canadian daylight-savings rules in particular.

E.90. Release 8.0.9

**Release date:** 2006-10-16

This release contains a variety of fixes from 8.0.8. For information about new features in the 8.0 major release, see Section E.99.

E.90.1. Migration to Version 8.0.9

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.90.2. Changes

- Fix crash when referencing **NEW** row values in rule WHERE expressions (Tom)
- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix mishandling of AFTER triggers when query contains a SQL function returning multiple rows (Tom)
- Fix `ALTER TABLE ... TYPE` to recheck **NOT NULL** for **USING** clause (Tom)
- Fix `string_to_array()` to handle overlapping matches for the separator string
  For example, `string_to_array(‘123xx456xxx789’, ‘xx’).
- Fix corner cases in pattern matching for `psql`’s \d commands
- Fix index-corrupting bugs in /contrib/ltrie (Teodor)
- Numerous robustness fixes in ecpg (Joachim Wieland)
- Fix backslash escaping in /contrib/dbmirror
- Fix instability of statistics collection on Win32 (Tom, Andrew)
- Fixes for AIX and Intel compilers (Tom)
E.91. Release 8.0.8

Release date: 2006-05-23

This release contains a variety of fixes from 8.0.7, including patches for extremely serious security issues. For information about new features in the 8.0 major release, see Section E.99.

E.91.1. Migration to Version 8.0.8

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq’s PQescapeStringConn()) to perform string escaping, rather than relying on ad hoc code to do it.

E.91.2. Changes

- Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)
  While PostgreSQL has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.

- Reject unsafe uses of \’ in string literals
  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts ” and not \’ as a representation of ASCII single quote in SQL string literals. By default, \’ is rejected only when client_encoding is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter backslash_quote is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of backslash_quote is in part to make it obvious that insecure clients are insecure.

- Modify libpq’s string-escaping routines to be aware of encoding considerations and standard_conforming_strings
  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple PostgreSQL connections concurrently should migrate to PQescapeStringConn() and PQescapeByteaConn() to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping “by hand” should be modified to rely on library routines instead.

- Fix some incorrect encoding conversion functions
  win1251_to_iso, alt_to_iso, euc_tw_to_big5, euc_tw_to_mic, mic_to_euc_tw were all broken to varying extents.

- Clean up stray remaining uses of \’ in strings (Bruce, Jan)
Appendix E. Release Notes

- Fix bug that sometimes caused OR’d index scans to miss rows they should have returned
- Fix WAL replay for case where a btree index has been truncated
- Fix \texttt{SIMILAR TO} for patterns involving | (Tom)
- Fix \texttt{SELECT INTO} and \texttt{CREATE TABLE AS} to create tables in the default tablespace, not the base directory (Kris Jurka)
- Fix server to use custom DH SSL parameters correctly (Michael Fuhr)
- Fix for Bonjour on Intel Macs (Ashley Clark)
- Fix various minor memory leaks
- Fix problem with password prompting on some Win32 systems (Robert Kinberg)

E.92. Release 8.0.7

\textbf{Release date:} 2006-02-14

This release contains a variety of fixes from 8.0.6. For information about new features in the 8.0 major release, see Section E.99.

E.92.1. Migration to Version 8.0.7

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.6, see the release notes for 8.0.6.

E.92.2. Changes

- Fix potential crash in \texttt{SET SESSION AUTHORIZATION} (CVE-2006-0553)
  An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.
- Fix bug with row visibility logic in self-inserted rows (Tom)
  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 8.0.4, 7.4.9, and 7.3.11 releases.
- Fix race condition that could lead to “file already exists” errors during \texttt{pg_clog} and \texttt{pg_subtrans} file creation (Tom)
- Fix cases that could lead to crashes if a cache-invalidation message arrives at just the wrong time (Tom)
- Properly check \texttt{DOMAIN} constraints for \texttt{UNKNOWN} parameters in prepared statements (Neil)
- Ensure \texttt{ALTER COLUMN TYPE} will process \texttt{FOREIGN KEY}, \texttt{UNIQUE}, and \texttt{PRIMARY KEY} constraints in the proper order (Nakano Yoshihisa)
Appendix E. Release Notes

- Fixes to allow restoring dumps that have cross-schema references to custom operators or operator classes (Tom)
- Allow pg_restore to continue properly after a COPY failure; formerly it tried to treat the remaining COPY data as SQL commands (Stephen Frost)
- Fix pg_ctl unregister crash when the data directory is not specified (Magnus)
- Fix ecpg crash on AMD64 and PPC (Neil)
- Recover properly if error occurs during argument passing in PL/python (Neil)
- Fix PL/perl’s handling of locales on Win32 to match the backend (Andrew)
- Fix crash when log_min_messages is set to DEBUG3 or above in postgresql.conf on Win32 (Bruce)
- Fix pgxs -L library path specification for Win32, Cygwin, OS X, AIX (Bruce)
- Check that SID is enabled while checking for Win32 admin privileges (Magnus)
- Properly reject out-of-range date inputs (Kris Jurka)
- Portability fix for testing presence of finite and isinf during configure (Tom)

E.93. Release 8.0.6

Release date: 2006-01-09

This release contains a variety of fixes from 8.0.5. For information about new features in the 8.0 major release, see Section E.99.

E.93.1. Migration to Version 8.0.6

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see the release notes for 8.0.3. Also, you might need to REINDEX indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.93.2. Changes

- Fix Windows code so that postmaster will continue rather than exit if there is no more room in ShmemBackendArray (Magnus)
  The previous behavior could lead to a denial-of-service situation if too many connection requests arrive close together. This applies only to the Windows port.
- Fix bug introduced in 8.0 that could allow ReadBuffer to return an already-used page as new, potentially causing loss of recently-committed data (Tom)
- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)
• Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)

This might require `REINDEX` to fix existing indexes on textual columns.

• Set locale environment variables during postmaster startup to ensure that plperl won’t change the locale later

This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need `REINDEX` to fix existing indexes on textual columns if this has happened to you.

• Allow more flexible relocation of installation directories (Tom)

Previous releases supported relocation only if all installation directory paths were the same except for the last component.

• Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)

• Various fixes for functions returning `RECORDs` (Tom)

• Fix bug in `/contrib/pgcrypto gen_salt`, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)

Salts for Blowfish and standard DES are unaffected.

• Fix `/contrib/dblink` to throw an error, rather than crashing, when the number of columns specified is different from what’s actually returned by the query (Joe)

---

**E.94. Release 8.0.5**

*Release date: 2005-12-12*

This release contains a variety of fixes from 8.0.4. For information about new features in the 8.0 major release, see Section E.99.

**E.94.1. Migration to Version 8.0.5**

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see the release notes for 8.0.3.

**E.94.2. Changes**

• Fix race condition in transaction log management

There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.

• Fix bgwriter problems after recovering from errors (Tom)
The background writer was found to leak buffer pins after write errors. While not fatal in itself, this might lead to mysterious blockages of later VACUUM commands.

- Prevent failure if client sends Bind protocol message when current transaction is already aborted
- /contrib/ltree fixes (Teodor)
- AIX and HPUX compile fixes (Tom)
- Retry file reads and writes after Windows NO_SYSTEM_RESOURCES error (Qingqing Zhou)
- Fix intermittent failure when log_line_prefix includes %i
- Fix psql performance issue with long scripts on Windows (Merlin Moncure)
- Fix missing updates of pg_group flat file
- Fix longstanding planning error for outer joins
  This bug sometimes caused a bogus error “RIGHT JOIN is only supported with merge-joinable join conditions”.
- Postpone timezone initialization until after postmaster.pid is created
  This avoids confusing startup scripts that expect the pid file to appear quickly.
- Prevent core dump in pg_autovacuum when a table has been dropped
- Fix problems with whole-row references (foo.*) to subquery results

E.95. Release 8.0.4

Release date: 2005-10-04

This release contains a variety of fixes from 8.0.3. For information about new features in the 8.0 major release, see Section E.99.

E.95.1. Migration to Version 8.0.4

A dump/restore is not required for those running 8.0.X. However, if you are upgrading from a version earlier than 8.0.3, see the release notes for 8.0.3.

E.95.2. Changes

- Fix error that allowed VACUUM to remove ctid chains too soon, and add more checking in code that follows ctid links
  This fixes a long-standing problem that could cause crashes in very rare circumstances.
- Fix CHAR() to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)
  In prior releases, the padding of CHAR() was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.
Appendix E. Release Notes

- Force a checkpoint before committing CREATE DATABASE
  This should fix recent reports of “index is not a btree” failures when a crash occurs shortly after CREATE DATABASE.
- Fix the sense of the test for read-only transaction in COPY
  The code formerly prohibited COPY TO, where it should prohibit COPY FROM.
- Handle consecutive embedded newlines in COPY CSV-mode input
- Fix date_trunc(week) for dates near year end
- Fix planning problem with outer-join ON clauses that reference only the inner-side relation
- Further fixes for x FULL JOIN y ON true corner cases
- Fix overenthusiastic optimization of x IN (SELECT DISTINCT ...) and related cases
- Fix mis-planning of queries with small LIMIT values due to poorly thought out “fuzzy” cost comparison
- Make array_in and array_recv more paranoid about validating their OID parameter
- Fix missing rows in queries like UPDATE a=... WHERE a... with GiST index on column a
- Improve robustness of datetime parsing
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Improve MIPS and M68K spinlock code
- Don’t try to open more than max_files_per_process files during postmaster startup
- Various memory leakage fixes
- Various portability improvements
- Update timezone data files
- Improve handling of DLL load failures on Windows
- Improve random-number generation on Windows
- Make psql -f filename return a nonzero exit code when opening the file fails
- Change pg_dump to handle inherited check constraints more reliably
- Fix password prompting in pg_restore on Windows
- Fix PL/pgSQL to handle var := var correctly when the variable is of pass-by-reference type
- Fix PL/Perl %_SHARED so it’s actually shared
- Fix contrib/pg_autovacuum to allow sleep intervals over 2000 sec
- Update contrib/tsearch2 to use current Snowball code

E.96. Release 8.0.3

Release date: 2005-05-09
This release contains a variety of fixes from 8.0.2, including several security-related issues. For information about new features in the 8.0 major release, see Section E.99.

### E.96.1. Migration to Version 8.0.3

A dump/restore is not required for those running 8.0.X. However, it is one possible way of handling two significant security problems that have been found in the initial contents of 8.0.X system catalogs. A dump/initdb/reload sequence using 8.0.3's initdb will automatically correct these problems.

The larger security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.)

The lesser problem is that the `contrib/tsearch2` module creates several functions that are improperly declared to return `internal` when they do not accept `internal` arguments. This breaks type safety for all functions using `internal` arguments.

It is strongly recommended that all installations repair these errors, either by initdb or by following the manual repair procedure given below. The errors at least allow unprivileged database users to crash their server process, and might allow unprivileged users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the same manual repair procedures shown in the 7.4.8 release notes.

### E.96.2. Changes

- Change encoding function signature to prevent misuse
- Change `contrib/tsearch2` to avoid unsafe use of `INTERNAL` function results
- Guard against incorrect second parameter to `record_out`
- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (e.g., `SELECT FOR UPDATE`) slightly sooner than for other purposes
  
  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM
  
  This could theoretically have caused loss of a page’s worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix comparisons of `TIME WITH TIME ZONE` values
  
  The comparison code was wrong in the case where the `--enable-integer-datetimes` configuration switch had been used. **NOTE:** if you have an index on a `TIME WITH TIME ZONE` column, it will need to be reindexed after installing this update, because the fix corrects the sort order of column values.
- Fix `EXTRACT(EPOCH)` for `TIME WITH TIME ZONE` values
- Fix mis-display of negative fractional seconds in `INTERVAL` values
This error only occurred when the `--enable-integer-datetimes` configuration switch had been used.

- Fix `pg_dump` to dump trigger names containing `%` correctly (Neil)
- Still more 64-bit fixes for `contrib/intagg`
- Prevent incorrect optimization of functions returning `RECORD`
- Prevent crash on `COALESCE(NULL,NULL)`
- Fix Borland makefile for `libpq`
- Fix `contrib/btree_gist` for `timetz` type (Teodor)
- Make `pg_ctl` check the PID found in `postmaster.pid` to see if it is still a live process
- Fix `pg_dump/pg_restore` problems caused by addition of dump timestamps
- Fix interaction between materializing holdable cursors and firing deferred triggers during transaction commit
- Fix memory leak in SQL functions returning pass-by-reference data types

**E.97. Release 8.0.2**

*Release date:* 2005-04-07

This release contains a variety of fixes from 8.0.1. For information about new features in the 8.0 major release, see Section E.99.

**E.97.1. Migration to Version 8.0.2**

A dump/restore is not required for those running 8.0.*. This release updates the major version number of the PostgreSQL libraries, so it might be necessary to re-link some user applications if they cannot find the properly-numbered shared library.

**E.97.2. Changes**

- Increment the major version number of all interface libraries (Bruce)
  This should have been done in 8.0.0. It is required so 7.4.X versions of PostgreSQL client applications, like `psql`, can be used on the same machine as 8.0.X applications. This might require re-linking user applications that use these libraries.
- Add Windows-only `wal_sync_method` setting of `fsync_writethrough` (Magnus, Bruce)
  This setting causes PostgreSQL to write through any disk-drive write cache when writing to WAL. This behavior was formerly called `fsync`, but was renamed because it acts quite differently from `fsync` on other platforms.
- Enable the `wal_sync_method` setting of `open_datasync` on Windows, and make it the default for that platform (Magnus, Bruce)
Because the default is no longer `fsync_writethrough`, data loss is possible during a power failure if the disk drive has write caching enabled. To turn off the write cache on Windows, from the Device Manager, choose the drive properties, then Policies.

- New cache management algorithm 2Q replaces ARC (Tom)
  This was done to avoid a pending US patent on ARC. The 2Q code might be a few percentage points slower than ARC for some workloads. A better cache management algorithm will appear in 8.1.
- Planner adjustments to improve behavior on freshly-created tables (Tom)
- Allow `plpgsql` to assign to an element of an array that is initially `NULL` (Tom)
  Formerly the array would remain `NULL`, but now it becomes a single-element array. The main SQL engine was changed to handle `UPDATE` of a null array value this way in 8.0, but the similar case in `plpgsql` was overlooked.
- Convert `\r\n` and `\r` to `\n` in plpython function bodies (Michael Fuhr)
  This prevents syntax errors when plpython code is written on a Windows or Mac client.
- Allow SPI cursors to handle utility commands that return rows, such as `EXPLAIN` (Tom)
- Fix `CLUSTER` failure after `ALTER TABLE SET WITHOUT OIDS` (Tom)
- Reduce memory usage of `ALTER TABLE ADD COLUMN` (Neil)
- Fix `ALTER LANGUAGE RENAME` (Tom)
- Document the Windows-only `register` and `unregister` options of `pg_ctl` (Magnus)
- Ensure operations done during backend shutdown are counted by statistics collector
  This is expected to resolve reports of `pg_autovacuum` not vacuuming the system catalogs often enough — it was not being told about catalog deletions caused by temporary table removal during backend exit.
- Change the Windows default for configuration parameter `log_destination` to `eventlog` (Magnus)
  By default, a server running on Windows will now send log output to the Windows event logger rather than standard error.
- Make Kerberos authentication work on Windows (Magnus)
- Allow `ALTER DATABASE RENAME` by superusers who aren’t flagged as having `CREATEDB` privilege (Tom)
- Modify WAL log entries for `CREATE` and `DROP DATABASE` to not specify absolute paths (Tom)
  This allows point-in-time recovery on a different machine with possibly different database location. Note that `CREATE TABLESPACE` still poses a hazard in such situations.
- Fix crash from a backend exiting with an open transaction that created a table and opened a cursor on it (Tom)
- Fix `array_map()` so it can call PL functions (Tom)
- Several `contrib/tsearch2` and `contrib/btree_gist` fixes (Teodor)
- Fix crash of some `contrib/pgcrypto` functions on some platforms (Marko Kreen)
- Fix `contrib/intagg` for 64-bit platforms (Tom)
- Fix `ecpg` bugs in parsing of `CREATE` statement (Michael)
- Work around gcc bug on powerpc and amd64 causing problems in `ecpg` (Christof Petig)
• Do not use locale-aware versions of upper(), lower(), and initcap() when the locale is C (Bruce)
  This allows these functions to work on platforms that generate errors for non-7-bit data when the
  locale is C.
• Fix quote_ident() to quote names that match keywords (Tom)
• Fix to_date() to behave reasonably when CC and YY fields are both used (Karel)
• Prevent to_char(interval) from failing when given a zero-month interval (Tom)
• Fix wrong week returned by date_trunc('week') (Bruce)
  date_trunc('week') returned the wrong year for the first few days of January in some years.
• Use the correct default mask length for class D addresses in INET data types (Tom)

E.98. Release 8.0.1

Release date: 2005-01-31

This release contains a variety of fixes from 8.0.0, including several security-related issues. For infor-
mation about new features in the 8.0 major release, see Section E.99.

E.98.1. Migration to Version 8.0.1

A dump/restore is not required for those running 8.0.0.

E.98.2. Changes

• Disallow LOAD to non-superusers
  On platforms that will automatically execute initialization functions of a shared library (this in-
  cludes at least Windows and ELF-based Unixen), LOAD can be used to make the server execute
  arbitrary code. Thanks to NGS Software for reporting this.
• Check that creator of an aggregate function has the right to execute the specified transition functions
  This oversight made it possible to bypass denial of EXECUTE permission on a function.
• Fix security and 64-bit issues in contrib/intagg
• Add needed STRICT marking to some contrib functions (Kris Jurka)
• Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
• Make ALTER TABLE ADD COLUMN enforce domain constraints in all cases
• Fix planning error for FULL and RIGHT outer joins
  The result of the join was mistakenly supposed to be sorted the same as the left input. This could
  not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright
  wrong answers.
Appendix E. Release Notes

- Improve planning of grouped aggregate queries
- ROLLBACK TO savepoint closes cursors created since the savepoint
- Fix inadequate backend stack size on Windows
- Avoid SHGetSpecialFolderPath() on Windows (Magnus)
- Fix some problems in running pg_autovacuum as a Windows service (Dave Page)
- Multiple minor bug fixes in pg_dump/pg_restore
- Fix ecpg segfault with named structs used in typedefs (Michael)

E.99. Release 8.0

Release date: 2005-01-19

E.99.1. Overview

Major changes in this release:

Microsoft Windows Native Server

This is the first PostgreSQL release to run natively on Microsoft Windows® as a server. It can run as a Windows service. This release supports NT-based Windows releases like Windows 2000 SP4, Windows XP, and Windows 2003. Older releases like Windows 95, Windows 98, and Windows ME are not supported because these operating systems do not have the infrastructure to support PostgreSQL. A separate installer project has been created to ease installation on Windows — see http://www.postgresql.org/ftp/win32/.

Although tested throughout our release cycle, the Windows port does not have the benefit of years of use in production environments that PostgreSQL has on Unix platforms. Therefore it should be treated with the same level of caution as you would a new product.

Previous releases required the Unix emulation toolkit Cygwin in order to run the server on Windows operating systems. PostgreSQL has supported native clients on Windows for many years.

Savepoints

Savepoints allow specific parts of a transaction to be aborted without affecting the remainder of the transaction. Prior releases had no such capability; there was no way to recover from a statement failure within a transaction except by aborting the whole transaction. This feature is valuable for application writers who require error recovery within a complex transaction.

Point-In-Time Recovery

In previous releases there was no way to recover from disk drive failure except to restore from a previous backup or use a standby replication server. Point-in-time recovery allows continuous backup of the server. You can recover either to the point of failure or to some transaction in the past.
Tablespaces

Tablespaces allow administrators to select different file systems for storage of individual tables, indexes, and databases. This improves performance and control over disk space usage. Prior releases used initlocation and manual symlink management for such tasks.

Improved Buffer Management, CHECKPOINT, VACUUM

This release has a more intelligent buffer replacement strategy, which will make better use of available shared buffers and improve performance. The performance impact of vacuum and checkpoints is also lessened.

Change Column Types

A column’s data type can now be changed with ALTER TABLE.

New Perl Server-Side Language

A new version of the plperl server-side language now supports a persistent shared storage area, triggers, returning records and arrays of records, and SPI calls to access the database.

Comma-separated-value (CSV) support in COPY

COPY can now read and write comma-separated-value files. It has the flexibility to interpret non-standard quoting and separation characters too.

E.99.2. Migration to Version 8.0

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- In READ COMMITTED serialization mode, volatile functions now see the results of concurrent transactions committed up to the beginning of each statement within the function, rather than up to the beginning of the interactive command that called the function.
- Functions declared STABLE or IMMUTABLE always use the snapshot of the calling query, and therefore do not see the effects of actions taken after the calling query starts, whether in their own transaction or other transactions. Such a function must be read-only, too, meaning that it cannot use any SQL commands other than SELECT.
- Nondeferred AFTER triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a difference when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation.
- Server configuration parameters virtual_host and tcpip_socket have been replaced with a more general parameter listen_addresses. Also, the server now listens on localhost by default, which eliminates the need for the -i postmaster switch in many scenarios.
- Server configuration parameters SortMem and VacuumMem have been renamed to work_mem and maintenance_work_mem to better reflect their use. The original names are still supported in SET and SHOW.
- Server configuration parameters log_pid, log_timestamp, and log_source_port have been replaced with a more general parameter log_line_prefix.
- Server configuration parameter syslog has been replaced with a more logical log_destination variable to control the log output destination.
- Server configuration parameter `log_statement` has been changed so it can selectively log just database modification or data definition statements. Server configuration parameter `log_duration` now prints only when `log_statement` prints the query.

- Server configuration parameter `max_expr_depth` parameter has been replaced with `max_stack_depth` which measures the physical stack size rather than the expression nesting depth. This helps prevent session termination due to stack overflow caused by recursive functions.

- The `length()` function no longer counts trailing spaces in `CHAR(n)` values.

- Casting an integer to `BIT(N)` selects the rightmost N bits of the integer, not the leftmost N bits as before.

- Updating an element or slice of a NULL array value now produces a nonnull array result, namely an array containing just the assigned-to positions.

- Syntax checking of array input values has been tightened up considerably. Junk that was previously allowed in odd places with odd results now causes an error. Empty-string element values must now be written as "", rather than writing nothing. Also changed behavior with respect to whitespace surrounding array elements: trailing whitespace is now ignored, for symmetry with leading whitespace (which has always been ignored).

- Overflow in integer arithmetic operations is now detected and reported as an error.

- The arithmetic operators associated with the single-byte "char" data type have been removed.

- The `extract()` function (also called `date_part`) now returns the proper year for BC dates. It previously returned one less than the correct year. The function now also returns the proper values for millennium and century.

- `CIDR` values now must have their nonmasked bits be zero. For example, we no longer allow `204.248.199.1/31` as a `CIDR` value. Such values should never have been accepted by PostgreSQL and will now be rejected.

- `EXECUTE` now returns a completion tag that matches the executed statement.

- `psql`'s `\copy` command now reads or writes to the query's `stdin/stdout`, rather than `psql`'s `stdin/stdout`. The previous behavior can be accessed via new `pstdin/pstdout` parameters.

- The JDBC client interface has been removed from the core distribution, and is now hosted at `http://jdbc.postgresql.org`.

- The Tcl client interface has also been removed. There are several Tcl interfaces now hosted at `http://gborg.postgresql.org`.

- The server now uses its own time zone database, rather than the one supplied by the operating system. This will provide consistent behavior across all platforms. In most cases, there should be little noticeable difference in time zone behavior, except that the time zone names used by `SET SHOW TimeZone` might be different from what your platform provides.

- Configure's threading option no longer requires users to run tests or edit configuration files; threading options are now detected automatically.

- Now that tablespaces have been implemented, initlocation has been removed.

- The API for user-defined GiST indexes has been changed. The Union and PickSplit methods are now passed a pointer to a special `GistEntryVector` structure, rather than a `bytea`.
E.99.3. Deprecated Features

Some aspects of PostgreSQL’s behavior have been determined to be suboptimal. For the sake of backward compatibility these have not been removed in 8.0, but they are considered deprecated and will be removed in the next major release.

- The 8.1 release will remove the to_char() function for intervals.
- The server now warns of empty strings passed to oid/float4/float8 data types, but continues to interpret them as zeroes as before. In the next major release, empty strings will be considered invalid input for these data types.
- By default, tables in PostgreSQL 8.0 and earlier are created with OIDs. In the next release, this will not be the case: to create a table that contains OIDs, the WITH OIDS clause must be specified or the default_with_oids configuration parameter must be set. Users are encouraged to explicitly specify WITH OIDS if their tables require OIDs for compatibility with future releases of PostgreSQL.

E.99.4. Changes

Below you will find a detailed account of the changes between release 8.0 and the previous major release.

E.99.4.1. Performance Improvements

- Support cross-data-type index usage (Tom)
  Before this change, many queries would not use an index if the data types did not match exactly. This improvement makes index usage more intuitive and consistent.
- New buffer replacement strategy that improves caching (Jan)
  Prior releases used a least-recently-used (LRU) cache to keep recently referenced pages in memory. The LRU algorithm did not consider the number of times a specific cache entry was accessed, so large table scans could force out useful cache pages. The new cache algorithm uses four separate lists to track most recently used and most frequently used cache pages and dynamically optimize their replacement based on the work load. This should lead to much more efficient use of the shared buffer cache. Administrators who have tested shared buffer sizes in the past should retest with this new cache replacement policy.
- Add subprocess to write dirty buffers periodically to reduce checkpoint writes (Jan)
  In previous releases, the checkpoint process, which runs every few minutes, would write all dirty buffers to the operating system’s buffer cache then flush all dirty operating system buffers to disk. This resulted in a periodic spike in disk usage that often hurt performance. The new code uses a background writer to trickle disk writes at a steady pace so checkpoints have far fewer dirty pages to write to disk. Also, the new code does not issue a global sync() call, but instead fsync()s just the files written since the last checkpoint. This should improve performance and minimize degradation during checkpoints.
- Add ability to prolong vacuum to reduce performance impact (Jan)
  On busy systems, VACUUM performs many I/O requests which can hurt performance for other users. This release allows you to slow down VACUUM to reduce its impact on other users, though this increases the total duration of VACUUM.
• Improve B-tree index performance for duplicate keys (Dmitry Tkach, Tom)
  This improves the way indexes are scanned when many duplicate values exist in the index.
• Use dynamically-generated table size estimates while planning (Tom)
  Formerly the planner estimated table sizes using the values seen by the last VACUUM or ANALYZE,
  both as to physical table size (number of pages) and number of rows. Now, the current physical
  table size is obtained from the kernel, and the number of rows is estimated by multiplying the
  table size by the row density (rows per page) seen by the last VACUUM or ANALYZE. This should
  produce more reliable estimates in cases where the table size has changed significantly since the
  last housekeeping command.
• Improved index usage with OR clauses (Tom)
  This allows the optimizer to use indexes in statements with many OR clauses that would not have
  been indexed in the past. It can also use multi-column indexes where the first column is specified
  and the second column is part of an OR clause.
• Improve matching of partial index clauses (Tom)
  The server is now smarter about using partial indexes in queries involving complex WHERE clauses.
• Improve performance of the GEQO optimizer (Tom)
  The GEQO optimizer is used to plan queries involving many tables (by default, twelve or more).
  This release speeds up the way queries are analyzed to decrease time spent in optimization.
• Miscellaneous optimizer improvements
  There is not room here to list all the minor improvements made, but numerous special cases work
  better than in prior releases.
• Improve lookup speed for C functions (Tom)
  This release uses a hash table to lookup information for dynamically loaded C functions. This
  improves their speed so they perform nearly as quickly as functions that are built into the server
  executable.
• Add type-specific ANALYZE statistics capability (Mark Cave-Ayland)
  This feature allows more flexibility in generating statistics for nonstandard data types.
• ANALYZE now collects statistics for expression indexes (Tom)
  Expression indexes (also called functional indexes) allow users to index not just columns but the
  results of expressions and function calls. With this release, the optimizer can gather and use statistics
  about the contents of expression indexes. This will greatly improve the quality of planning for
  queries in which an expression index is relevant.
• New two-stage sampling method for ANALYZE (Manfred Koizar)
  This gives better statistics when the density of valid rows is very different in different regions of a
  table.
• Speed up TRUNCATE (Tom)
  This buys back some of the performance loss observed in 7.4, while still keeping TRUNCATE
  transaction-safe.

**E.99.4.2. Server Changes**

• Add WAL file archiving and point-in-time recovery (Simon Riggs)
Appendix E. Release Notes

• Add tablespaces so admins can control disk layout (Gavin)
• Add a built-in log rotation program (Andreas Pfug)
  It is now possible to log server messages conveniently without relying on either syslog or an external log rotation program.
• Add new read-only server configuration parameters to show server compile-time settings: block_size, integer_datetimes, max_function_args, max_identifier_length, max_index_keys (Joe)
• Make quoting of sameuser, samegroup, and all remove special meaning of these terms in pg_hba.conf (Andrew)
• Use clearer IPv6 name ::1/128 for localhost in default pg_hba.conf (Andrew)
• Use CIDR format in pg_hba.conf examples (Andrew)
• Rename server configuration parameters SortMem and VacuumMem to work_mem and maintenance_work_mem (Old names still supported) (Tom)
  This change was made to clarify that bulk operations such as index and foreign key creation use maintenance_work_mem, while work_mem is for workspaces used during query execution.
• Allow logging of session disconnections using server configuration log_disconnections (Andrew)
• Add new server configuration parameter log_line_prefix to allow control of information emitted in each log line (Andrew)
  Available information includes user name, database name, remote IP address, and session start time.
• Remove server configuration parameters log_pid, log_timestamp, log_source_port; functionality superseded by log_line_prefix (Andrew)
• Replace the virtual_host and tcpip_socket parameters with a unified listen_addresses parameter (Andrew, Tom)
  virtual_host could only specify a single IP address to listen on. listen_addresses allows multiple addresses to be specified.
• Listen on localhost by default, which eliminates the need for the -i postmaster switch in many scenarios (Andrew)
  Listening on localhost (127.0.0.1) opens no new security holes but allows configurations like Windows and JDBC, which do not support local sockets, to work without special adjustments.
• Remove syslog server configuration parameter, and add more logical log_destination variable to control log output location (Magnus)
• Change server configuration parameter log_statement to take values all, mod, ddl, or none to select which queries are logged (Bruce)
  This allows administrators to log only data definition changes or only data modification statements.
• Some logging-related configuration parameters could formerly be adjusted by ordinary users, but only in the “more verbose” direction. They are now treated more strictly: only superusers can set them. However, a superuser can use ALTER USER to provide per-user settings of these values for non-superusers. Also, it is now possible for superusers to set values of superuser-only configuration parameters via PGOPTIONS.
• Allow configuration files to be placed outside the data directory (mlw)
Appendix E. Release Notes

By default, configuration files are kept in the cluster’s top directory. With this addition, configuration files can be placed outside the data directory, easing administration.

- Plan prepared queries only when first executed so constants can be used for statistics (Oliver Jowett)

Prepared statements plan queries once and execute them many times. While prepared queries avoid the overhead of re-planning on each use, the quality of the plan suffers from not knowing the exact parameters to be used in the query. In this release, planning of unnamed prepared statements is delayed until the first execution, and the actual parameter values of that execution are used as optimization hints. This allows use of out-of-line parameter passing without incurring a performance penalty.

- Allow DECLARE CURSOR to take parameters (Oliver Jowett)

It is now useful to issue DECLARE CURSOR in a Parse message with parameters. The parameter values sent at Bind time will be substituted into the execution of the cursor’s query.

- Fix hash joins and aggregates of inet and cidr data types (Tom)

Release 7.4 handled hashing of mixed inet and cidr values incorrectly. (This bug did not exist in prior releases because they wouldn’t try to hash either data type.)

- Make log_duration print only when log_statement prints the query (Ed L.)

E.99.4.3. Query Changes

- Add savepoints (nested transactions) (Alvaro)

- Unsupported isolation levels are now accepted and promoted to the nearest supported level (Peter)

The SQL specification states that if a database doesn’t support a specific isolation level, it should use the next more restrictive level. This change complies with that recommendation.

- Allow BEGIN WORK to specify transaction isolation levels like START TRANSACTION does (Bruce)

- Fix table permission checking for cases in which rules generate a query type different from the originally submitted query (Tom)

- Implement dollar quoting to simplify single-quote usage (Andrew, Tom, David Fetter)

In previous releases, because single quotes had to be used to quote a function’s body, the use of single quotes inside the function text required use of two single quotes or other error-prone notations. With this release we add the ability to use “dollar quoting” to quote a block of text. The ability to use different quoting delimiters at different nesting levels greatly simplifies the task of quoting correctly, especially in complex functions. Dollar quoting can be used anywhere quoted text is needed.

- Make CASE val WHEN compval1 THEN ... evaluate val only once (Tom)

CASE no longer evaluates the tested expression multiple times. This has benefits when the expression is complex or is volatile.

- Test HAVING before computing target list of an aggregate query (Tom)

Fixes improper failure of cases such as SELECT SUM(win)/SUM(lose) ... GROUP BY ... HAVING SUM(lose) > 0. This should work but formerly could fail with divide-by-zero.

- Replace maxExprDepth parameter with max_stack_depth parameter, measured in kilobytes of stack size (Tom)
Appendix E. Release Notes

This gives us a fairly bulletproof defense against crashing due to runaway recursive functions. Instead of measuring the depth of expression nesting, we now directly measure the size of the execution stack.

• Allow arbitrary row expressions (Tom)

This release allows SQL expressions to contain arbitrary composite types, that is, row values. It also allows functions to more easily take rows as arguments and return row values.

• Allow LIKE/ILIKE to be used as the operator in row and subselect comparisons (Fabien Coelho)

• Avoid locale-specific case conversion of basic ASCII letters in identifiers and keywords (Tom)

This solves the “Turkish problem” with mangling of words containing İ and ğ. Folding of characters outside the 7-bit-ASCII set is still locale-aware.

• Improve syntax error reporting (Fabien, Tom)

Syntax error reports are more useful than before.

• Change EXECUTE to return a completion tag matching the executed statement (Kris Jurka)

Previous releases return an EXECUTE tag for any EXECUTE call. In this release, the tag returned will reflect the command executed.

• Avoid emitting NATURAL CROSS JOIN in rule listings (Tom)

Such a clause makes no logical sense, but in some cases the rule decompiler formerly produced this syntax.

E.99.4.4. Object Manipulation Changes

• Add COMMENT ON for casts, conversions, languages, operator classes, and large objects (Christopher)

• Add new server configuration parameter default_with_ooids to control whether tables are created with OIDS by default (Neil)

This allows administrators to control whether CREATE TABLE commands create tables with or without OID columns by default. (Note: the current factory default setting for default_with_ooids is TRUE, but the default will become FALSE in future releases.)

• Add WITH/WITHOUT OIDS clause to CREATE TABLE AS (Neil)

• Allow ALTER TABLE DROP COLUMN to drop an OID column (ALTER TABLE SET WITHOUT OIDS still works) (Tom)

• Allow composite types as table columns (Tom)

• Allow ALTER ... ADD COLUMN with defaults and NOT NULL constraints; works per SQL spec (Rod)

It is now possible for ADD COLUMN to create a column that is not initially filled with NULLs, but with a specified default value.

• Add ALTER COLUMN TYPE to change column’s type (Rod)

It is now possible to alter a column’s data type without dropping and re-adding the column.

• Allow multiple ALTER actions in a single ALTER TABLE command (Rod)

This is particularly useful for ALTER commands that rewrite the table (which include ALTER COLUMN TYPE and ADD COLUMN with a default). By grouping ALTER commands together, the table need be rewritten only once.
Appendix E. Release Notes

- Allow `ALTER TABLE` to add `SERIAL` columns (Tom)
  This falls out from the new capability of specifying defaults for new columns.
- Allow changing the owners of aggregates, conversions, databases, functions, operators, operator classes, schemas, types, and tablespaces (Christopher, Euler Taveira de Oliveira)
  Previously this required modifying the system tables directly.
- Allow temporary object creation to be limited to `SECURITY DEFINER` functions (Sean Chittenden)
- Add `ALTER TABLE ... SET WITHOUT CLUSTER` (Christopher)
  Prior to this release, there was no way to clear an auto-cluster specification except to modify the system tables.
- Constraint/Index/`SERIAL` names are now `table_column_type` with numbers appended to guarantee uniqueness within the schema (Tom)
  The SQL specification states that such names should be unique within a schema.
- Add `pg_get_serial_sequence()` to return a `SERIAL` column’s sequence name (Christopher)
  This allows automated scripts to reliably find the `SERIAL` sequence name.
- Warn when primary/foreign key data type mismatch requires costly lookup
- New `ALTER INDEX` command to allow moving of indexes between tablespaces (Gavin)
- Make `ALTER TABLE Owner` change dependent sequence ownership too (Alvaro)

E.99.4.5. Utility Command Changes

- Allow `CREATE SCHEMA` to create triggers, indexes, and sequences (Neil)
- Add `ALSO` keyword to `CREATE RULE` (Fabien Coelho)
  This allows `ALSO` to be added to rule creation to contrast it with `INSTEAD` rules.
- Add `NOWAIT` option to `LOCK` (Tatsuo)
  This allows the `LOCK` command to fail if it would have to wait for the requested lock.
- Allow `COPY` to read and write comma-separated-value (CSV) files (Andrew, Bruce)
- Generate error if the `COPY` delimiter and NULL string conflict (Bruce)
- `GRANT/REVOKE` behavior follows the SQL spec more closely
- Avoid locking conflict between `CREATE INDEX` and `CHECKPOINT` (Tom)
  In 7.3 and 7.4, a long-running B-tree index build could block concurrent `CHECKPOINTS` from completing, thereby causing WAL bloat because the WAL log could not be recycled.
- Database-wide `ANALYZE` does not hold locks across tables (Tom)
  This reduces the potential for deadlocks against other backends that want exclusive locks on tables.
  To get the benefit of this change, do not execute database-wide `ANALYZE` inside a transaction block (`BEGIN` block); it must be able to commit and start a new transaction for each table.
- `REINDEX` does not exclusively lock the index’s parent table anymore
  The index itself is still exclusively locked, but readers of the table can continue if they are not using the particular index being rebuilt.
- Erase MD5 user passwords when a user is renamed (Bruce)
PostgreSQL uses the user name as salt when encrypting passwords via MD5. When a user’s name is changed, the salt will no longer match the stored MD5 password, so the stored password becomes useless. In this release a notice is generated and the password is cleared. A new password must then be assigned if the user is to be able to log in with a password.

- New `pg_ctl kill` option for Windows (Andrew)
  Windows does not have a `kill` command to send signals to backends so this capability was added to `pg_ctl`.
- Information schema improvements
- Add `--pwfile` option to `initdb` so the initial password can be set by GUI tools (Magnus)
- Detect locale/encoding mismatch in `initdb` (Peter)
- Add `register` command to `pg_ctl` to register Windows operating system service (Dave Page)

### E.99.4.6. Data Type and Function Changes

- More complete support for composite types (row types) (Tom)
  Composite values can be used in many places where only scalar values worked before.
- Reject nonrectangular array values as erroneous (Joe)
  Formerly, `array_in` would silently build a surprising result.
- Overflow in integer arithmetic operations is now detected (Tom)
- The arithmetic operators associated with the single-byte "char" data type have been removed.
  Formerly, the parser would select these operators in many situations where an “unable to select an operator” error would be more appropriate, such as `null * null`. If you actually want to do arithmetic on a "char" column, you can cast it to integer explicitly.
- Syntax checking of array input values considerably tightened up (Joe)
  Junk that was previously allowed in odd places with odd results now causes an `ERROR`, for example, non-whitespace after the closing right brace.
- Empty-string array element values must now be written as " ", rather than writing nothing (Joe)
  Formerly, both ways of writing an empty-string element value were allowed, but now a quoted empty string is required. The case where nothing at all appears will probably be considered to be a NULL element value in some future release.
- Array element trailing whitespace is now ignored (Joe)
  Formerly leading whitespace was ignored, but trailing whitespace between an element value and the delimiter or right brace was significant. Now trailing whitespace is also ignored.
- Emit array values with explicit array bounds when lower bound is not one (Joe)
- Accept `YYYY-monthname-DD` as a date string (Tom)
- Make `netmask` and `hostmask` functions return maximum-length mask length (Tom)
- Change factorial function to return `numeric` (Gavin)
  Returning `numeric` allows the factorial function to work for a wider range of input values.
- `to_char/to_date()` date conversion improvements (Kurt Roeckx, Fabien Coelho)
- Make `length()` disregard trailing spaces in `CHAR(n)` (Gavin)
Appendix E. Release Notes

This change was made to improve consistency: trailing spaces are semantically insignificant in CHAR(n) data, so they should not be counted by length().

- Warn about empty string being passed to OID/float4/float8 data types (Neil)
  8.1 will throw an error instead.

- Allow leading or trailing whitespace in int2/int4/int8/float4/float8 input routines (Neil)

- Better support for IEEE Infinity and NaN values in float4/float8 (Neil)
  These should now work on all platforms that support IEEE-compliant floating point arithmetic.

- Add week option to date_trunc() (Robert Creager)

- Fix to_char for 1 BC (previously it returned 1 AD) (Bruce)

- Fix date_part (year) for BC dates (previously it returned one less than the correct year) (Bruce)

- Fix date_part() to return the proper millennium and century (Fabien Coelho)
  In previous versions, the century and millennium results had a wrong number and started in the
  wrong year, as compared to standard reckoning of such things.

- Add ceiling() as an alias for ceil(), and power() as an alias for pow() for standards compliance (Neil)

- Change ln(), log(), power(), and sqrt() to emit the correct SQLSTATE error codes for certain
  error conditions, as specified by SQL:2003 (Neil)

- Add width_bucket() function as defined by SQL:2003 (Neil)

- Add generate_series() functions to simplify working with numeric sets (Joe)

- Fix upper/lower/initcap() functions to work with multibyte encodings (Tom)

- Add boolean and bitwise integer AND/OR aggregates (Fabien Coelho)

- New session information functions to return network addresses for client and server (Sean Chittenden)

- Add function to determine the area of a closed path (Sean Chittenden)

- Add function to send cancel request to other backends (Magnus)

- Add interval plus datetime operators (Tom)
  The reverse ordering, datetime plus interval, was already supported, but both are required by
  the SQL standard.

- Casting an integer to BIT(N) selects the rightmost N bits of the integer (Tom)
  In prior releases, the leftmost N bits were selected, but this was deemed unhelpful, not to mention
  inconsistent with casting from bit to int.

- Require CIDR values to have all nonmasked bits be zero (Kevin Brintnall)

E.99.4.7. Server-Side Language Changes

- In READ COMMITTED serialization mode, volatile functions now see the results of concurrent trans-
  actions committed up to the beginning of each statement within the function, rather than up to the
  beginning of the interactive command that called the function.

- Functions declared STABLE or IMMUTABLE always use the snapshot of the calling query, and there-
  fore do not see the effects of actions taken after the calling query starts, whether in their own
transaction or other transactions. Such a function must be read-only, too, meaning that it cannot use any SQL commands other than SELECT. There is a considerable performance gain from declaring a function STABLE or IMMUTABLE rather than VOLATILE.

- Nondeferred AFTER triggers are now fired immediately after completion of the triggering query, rather than upon finishing the current interactive command. This makes a difference when the triggering query occurred within a function: the trigger is invoked before the function proceeds to its next operation. For example, if a function inserts a new row into a table, any nondeferred foreign key checks occur before proceeding with the function.

- Allow function parameters to be declared with names (Dennis Björklund)
  This allows better documentation of functions. Whether the names actually do anything depends on the specific function language being used.

- Allow PL/pgSQL parameter names to be referenced in the function (Dennis Björklund)
  This basically creates an automatic alias for each named parameter.

- Do minimal syntax checking of PL/pgSQL functions at creation time (Tom)
  This allows us to catch simple syntax errors sooner.

- More support for composite types (row and record variables) in PL/pgSQL
  For example, it now works to pass a rowtype variable to another function as a single variable.

- Default values for PL/pgSQL variables can now reference previously declared variables

- Improve parsing of PL/pgSQL FOR loops (Tom)
  Parsing is now driven by presence of "..." rather than data type of FOR variable. This makes no difference for correct functions, but should result in more understandable error messages when a mistake is made.

- Major overhaul of PL/Perl server-side language (Command Prompt, Andrew Dunstan)

- In PL/Tcl, SPI commands are now run in subtransactions. If an error occurs, the subtransaction is cleaned up and the error is reported as an ordinary Tcl error, which can be trapped with catch. Formerly, it was not possible to catch such errors.

- Accept ELSEIF in PL/pgSQL (Neil)
  Previously PL/pgSQL only allowed ELSIF, but many people are accustomed to spelling this keyword ELSEIF.

E.99.4.8. psql Changes

- Improve psql information display about database objects (Christopher)
- Allow psql to display group membership in \du and \dg (Markus Bertheau)
- Prevent psql \dn from showing temporary schemas (Bruce)
- Allow psql to handle tilde user expansion for file names (Zach Irmen)
- Allow psql to display fancy prompts, including color, via readline (Reece Hart, Chet Ramey)
- Make psql \copy match COPY command syntax fully (Tom)
- Show the location of syntax errors (Fabien Coelho, Tom)
- Add CLUSTER information to psql \d display (Bruce)
- Change psql \copy stdin/stdout to read from command input/output (Bruce)
Appendix E. Release Notes

- Add `pstdin/pstdout` to read from psql’s stdin/stdout (Mark Feit)
- Add global psql configuration file, `psqlrc.sample` (Bruce)
  This allows a central file where global psql startup commands can be stored.
- Have `psql \d+` indicate if the table has an OID column (Neil)
- On Windows, use binary mode in psql when reading files so control-Z is not seen as end-of-file
- Have `\dn+` show permissions and description for schemas (Dennis Björklund)
- Improve tab completion support (Stefan Kaltenbrunn, Greg Sabino Mullane)
- Allow boolean settings to be set using upper or lower case (Michael Paesold)

E.99.4.9. pg_dump Changes

- Use dependency information to improve the reliability of pg_dump (Tom)
  This should solve the longstanding problems with related objects sometimes being dumped in the wrong order.
- Have pg_dump output objects in alphabetical order if possible (Tom)
  This should make it easier to identify changes between dump files.
- Allow pg_restore to ignore some SQL errors (Fabien Coelho)
  This makes pg_restore’s behavior similar to the results of feeding a pg_dump output script to psql. In most cases, ignoring errors and plowing ahead is the most useful thing to do. Also added was a pg_restore option to give the old behavior of exiting on an error.
- pg_restore -l display now includes objects’ schema names
- New begin/end markers in pg_dump text output (Bruce)
- Add start/stop times for pg_dump/pg_dumpall in verbose mode (Bruce)
- Allow most pg_dump options in pg_dumpall (Christopher)
- Have pg_dump use `ALTER OWNER` rather than `SET SESSION AUTHORIZATION` by default (Christopher)

E.99.4.10. libpq Changes

- Make libpq’s SIGPIPE handling thread-safe (Bruce)
- Add `PQmbdsplen()` which returns the display length of a character (Tatsuo)
- Add thread locking to SSL and Kerberos connections (Manfred Spraul)
- Allow `PQoidValue()`, `PQcmdTuples()`, and `PQoidStatus()` to work on `EXECUTE` commands (Neil)
- Add `PQserverVersion()` to provide more convenient access to the server version number (Greg Sabino Mullane)
- Add `PQprepare/PQsendPrepared()` functions to support preparing statements without necessarily specifying the data types of their parameters (Abhijit Menon-Sen)
- Many ECPG improvements, including `SET DESCRIPTOR` (Michael)
Appendix E. Release Notes

E.99.4.11. Source Code Changes

• Allow the database server to run natively on Windows (Claudio, Magnus, Andrew)
• Shell script commands converted to C versions for Windows support (Andrew)
• Create an extension makefile framework (Fabien Coelho, Peter)
  This simplifies the task of building extensions outside the original source tree.
• Support relocatable installations (Bruce)
  Directory paths for installed files (such as the /share directory) are now computed relative to the
  actual location of the executables, so that an installation tree can be moved to another place without
  reconfiguring and rebuilding.
• Use --with-docdir to choose installation location of documentation; also allow --infodir
  (Peter)
• Add --without-docdir to prevent installation of documentation (Peter)
• Upgrade to DocBook V4.2 SGML (Peter)
• New PostgreSQL CVS tag (Marc)
  This was done to make it easier for organizations to manage their own copies of the PostgreSQL
  CVS repository. File version stamps from the master repository will not get munged by checking
  into or out of a copied repository.
• Clarify locking code (Manfred Koizar)
• Buffer manager cleanup (Neil)
• Decouple platform tests from CPU spinlock code (Bruce, Tom)
• Add inlined test-and-set code on PA-RISC for gcc (ViSolve, Tom)
• Improve i386 spinlock code (Manfred Spraul)
• Clean up spinlock assembly code to avoid warnings from newer gcc releases (Tom)
• Remove JDBC from source tree; now a separate project
• Remove the libpgtcl client interface; now a separate project
• More accurately estimate memory and file descriptor usage (Tom)
• Improvements to the Mac OS X startup scripts (Ray A.)
• New fsync() test program (Bruce)
• Major documentation improvements (Neil, Peter)
• Remove pg_encoding; not needed anymore
• Remove pg_id; not needed anymore
• Remove initlocation; not needed anymore
• Auto-detect thread flags (no more manual testing) (Bruce)
• Use Olson’s public domain timezone library (Magnus)
• With threading enabled, use thread flags on Unixware for backend executables too (Bruce)
  Unixware cannot mix threaded and nonthreaded object files in the same executable, so everything
  must be compiled as threaded.
• psql now uses a flex-generated lexical analyzer to process command strings
• Reimplement the linked list data structure used throughout the backend (Neil)
  This improves performance by allowing list append and length operations to be more efficient.
• Allow dynamically loaded modules to create their own server configuration parameters (Thomas Hallgren)
• New Brazilian version of FAQ (Euler Taveira de Oliveira)
• Add French FAQ (Guillaume Lelarge)
• New pgevent for Windows logging
• Make libpq and ECPG build as proper shared libraries on OS X (Tom)

E.99.4.12. Contrib Changes

• Overhaul of contrib/dblink (Joe)
• contrib/dbmirror improvements (Steven Singer)
• New contrib/xml2 (John Gray, Torchbox)
• Updated contrib/mysql
• New version of contrib/btree_gist (Teodor)
• New contrib/trgm, trigram matching for PostgreSQL (Teodor)
• Many contrib/tsearch2 improvements (Teodor)
• Add double metaphone to contrib/fuzzystrmatch (Andrew)
• Allow contrib/pg_autovacuum to run as a Windows service (Dave Page)
• Add functions to contrib/dbsize (Andreas Pflug)
• Removed contrib/pg_logger: obsoleted by integrated logging subprocess
• Removed contrib/rserv: obsoleted by various separate projects

E.100. Release 7.4.30

Release date: 2010-10-04

This release contains a variety of fixes from 7.4.29. For information about new features in the 7.4 major release, see Section E.130.
This is expected to be the last PostgreSQL release in the 7.4.X series. Users are encouraged to update to a newer release branch soon.

E.100.1. Migration to Version 7.4.30

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.26, see the release notes for 7.4.26.
E.100.2. Changes

- Use a separate interpreter for each calling SQL userid in PL/Perl and PL/Tcl (Tom Lane)

  This change prevents security problems that can be caused by subverting Perl or Tcl code that will be executed later in the same session under another SQL user identity (for example, within a \texttt{SECURITY DEFINER} function). Most scripting languages offer numerous ways that that might be done, such as redefining standard functions or operators called by the target function. Without this change, any SQL user with Perl or Tcl language usage rights can do essentially anything with the SQL privileges of the target function’s owner.

  The cost of this change is that intentional communication among Perl and Tcl functions becomes more difficult. To provide an escape hatch, PL/PerlU and PL/TclU functions continue to use only one interpreter per session. This is not considered a security issue since all such functions execute at the trust level of a database superuser already.

  It is likely that third-party procedural languages that claim to offer trusted execution have similar security issues. We advise contacting the authors of any PL you are depending on for security-critical purposes.

  Our thanks to Tim Bunce for pointing out this issue (CVE-2010-3433).

- Prevent possible crashes in \texttt{pg\_get\_expr()} by disallowing it from being called with an argument that is not one of the system catalog columns it’s intended to be used with (Heikki Linnakangas, Tom Lane)

- Fix “cannot handle unplanned sub-select” error (Tom Lane)

  This occurred when a sub-select contains a join alias reference that expands into an expression containing another sub-select.

- Take care to \texttt{fsync} the contents of lockfiles (both \texttt{postmaster.pid} and the socket lockfile) while writing them (Tom Lane)

  This omission could result in corrupted lockfile contents if the machine crashes shortly after postmaster start. That could in turn prevent subsequent attempts to start the postmaster from succeeding, until the lockfile is manually removed.

- Improve \texttt{contrib/dblink}’s handling of tables containing dropped columns (Tom Lane)

- Fix connection leak after “duplicate connection name” errors in \texttt{contrib/dblink} (Itagaki Takahiro)

- Update build infrastructure and documentation to reflect the source code repository’s move from CVS to Git (Magnus Hagander and others)

E.101. Release 7.4.29

\textbf{Release date}: 2010-05-17

This release contains a variety of fixes from 7.4.28. For information about new features in the 7.4 major release, see Section E.130.

The PostgreSQL community will stop releasing updates for the 7.4.X release series in July 2010. Users are encouraged to update to a newer release branch soon.
Appendix E. Release Notes

E.101.1. Migration to Version 7.4.29

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.26, see the release notes for 7.4.26.

E.101.2. Changes

- Enforce restrictions in plperl using an opmask applied to the whole interpreter, instead of using Safe.pm (Tim Bunce, Andrew Dunstan)

  Recent developments have convinced us that Safe.pm is too insecure to rely on for making plperl trustworthy. This change removes use of Safe.pm altogether, in favor of using a separate interpreter with an opcode mask that is always applied. Pleasant side effects of the change include that it is now possible to use Perl’s strict pragma in a natural way in plperl, and that Perl’s $a and $b variables work as expected in sort routines, and that function compilation is significantly faster. (CVE-2010-1169)

- Prevent PL/Tcl from executing untrustworthy code from pltcl_modules (Tom)

  PL/Tcl’s feature for autoloading Tcl code from a database table could be exploited for trojan-horse attacks, because there was no restriction on who could create or insert into that table. This change disables the feature unless pltcl_modules is owned by a superuser. (However, the permissions on the table are not checked, so installations that really need a less-than-secure modules table can still grant suitable privileges to trusted non-superusers.) Also, prevent loading code into the unrestricted “normal” Tcl interpreter unless we are really going to execute a pltclu function. (CVE-2010-1170)

- Do not allow an unprivileged user to reset superuser-only parameter settings (Alvaro)

  Previously, if an unprivileged user ran ALTER USER ... RESET ALL for himself, or ALTER DATABASE ... RESET ALL for a database he owns, this would remove all special parameter settings for the user or database, even ones that are only supposed to be changeable by a superuser. Now, the ALTER will only remove the parameters that the user has permission to change.

- Avoid possible crash during backend shutdown if shutdown occurs when a CONTEXT addition would be made to log entries (Tom)

  In some cases the context-printing function would fail because the current transaction had already been rolled back when it came time to print a log message.

- Update pl/perl’s ppport.h for modern Perl versions (Andrew)

- Fix assorted memory leaks in pl/python (Andreas Freund, Tom)

- Ensure that contrib/pgstattuple functions respond to cancel interrupts promptly (Tatsuhito Kasahara)

- Make server startup deal properly with the case that shmget() returns EINVAL for an existing shared memory segment (Tom)

  This behavior has been observed on BSD-derived kernels including OS X. It resulted in an entirely-misleading startup failure complaining that the shared memory request size was too large.
Appendix E. Release Notes

E.102. Release 7.4.28

Release date: 2010-03-15

This release contains a variety of fixes from 7.4.27. For information about new features in the 7.4 major release, see Section E.130.

The PostgreSQL community will stop releasing updates for the 7.4.X release series in July 2010. Users are encouraged to update to a newer release branch soon.

E.102.1. Migration to Version 7.4.28

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.26, see the release notes for 7.4.26.

E.102.2. Changes

- Add new configuration parameter `ssl_renegotiation_limit` to control how often we do session key renegotiation for an SSL connection (Magnus)
  
  This can be set to zero to disable renegotiation completely, which may be required if a broken SSL library is used. In particular, some vendors are shipping stopgap patches for CVE-2009-3555 that cause renegotiation attempts to fail.

- Make `substring()` for bit types treat any negative length as meaning “all the rest of the string” (Tom)
  
  The previous coding treated only `-1` that way, and would produce an invalid result value for other negative values, possibly leading to a crash (CVE-2010-0442).

- Fix some cases of pathologically slow regular expression matching (Tom)

- When reading `pg_hba.conf` and related files, do not treat `@something` as a file inclusion request if the `@` appears inside quote marks; also, never treat `@` by itself as a file inclusion request (Tom)
  
  This prevents erratic behavior if a role or database name starts with `@`. If you need to include a file whose path name contains spaces, you can still do so, but you must write `@"/path to/file"` rather than putting the quotes around the whole construct.

- Prevent infinite loop on some platforms if a directory is named as an inclusion target in `pg_hba.conf` and related files (Tom)

- Ensure PL/Tcl initializes the Tcl interpreter fully (Tom)
  
  The only known symptom of this oversight is that the Tcl `clock` command misbehaves if using Tcl 8.5 or later.

- Prevent crash in `contrib/dlink` when too many key columns are specified to a `dblink_build_sql_*` function (Rushabh Lathia, Joe Conway)
E.103. Release 7.4.27

Release date: 2009-12-14

This release contains a variety of fixes from 7.4.26. For information about new features in the 7.4 major release, see Section E.130.

E.103.1. Migration to Version 7.4.27

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.26, see the release notes for 7.4.26.

E.103.2. Changes

- Protect against indirect security threats caused by index functions changing session-local state (Gurjeet Singh, Tom)
  This change prevents allegedly-immutable index functions from possibly subverting a superuser’s session (CVE-2009-4136).
- Reject SSL certificates containing an embedded null byte in the common name (CN) field (Magnus)
  This prevents unintended matching of a certificate to a server or client name during SSL validation (CVE-2009-4034).
- Fix possible crash during backend-startup-time cache initialization (Tom)
- Prevent signals from interrupting VACUUM at unsafe times (Alvaro)
  This fix prevents a PANIC if a VACUUM FULL is cancelled after it’s already committed its tuple movements, as well as transient errors if a plain VACUUM is interrupted after having truncated the table.
- Fix possible crash due to integer overflow in hash table size calculation (Tom)
  This could occur with extremely large planner estimates for the size of a hashjoin’s result.
- Fix very rare crash in inet/cidr comparisons (Chris Mikkelson)
- Fix PAM password processing to be more robust (Tom)
  The previous code is known to fail with the combination of the Linux pam_krb5 PAM module with Microsoft Active Directory as the domain controller. It might have problems elsewhere too, since it was making unjustified assumptions about what arguments the PAM stack would pass to it.
- Make the postmaster ignore any application_name parameter in connection request packets, to improve compatibility with future libpq versions (Tom)

E.104. Release 7.4.26

Release date: 2009-09-09
This release contains a variety of fixes from 7.4.25. For information about new features in the 7.4 major release, see Section E.130.

E.104.1. Migration to Version 7.4.26

A dump/restore is not required for those running 7.4.X. However, if you have any hash indexes on interval columns, you must \texttt{REINDEX} them after updating to 7.4.26. Also, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.104.2. Changes

- Disallow \texttt{RESET ROLE} and \texttt{RESET SESSION AUTHORIZATION} inside security-definer functions (Tom, Heikki)
  
  This covers a case that was missed in the previous patch that disallowed \texttt{SET ROLE} and \texttt{SET SESSION AUTHORIZATION} inside security-definer functions. (See CVE-2007-6600)

- Fix handling of sub-SELECTs appearing in the arguments of an outer-level aggregate function (Tom)

- Fix hash calculation for data type \texttt{interval} (Tom)

  This corrects wrong results for hash joins on interval values. It also changes the contents of hash indexes on interval columns. If you have any such indexes, you must \texttt{REINDEX} them after updating.

- Fix overflow for \texttt{INTERVAL 'x ms'} when $x$ is more than 2 million and integer datetimes are in use (Alex Hunsaker)

- Fix calculation of distance between a point and a line segment (Tom)

  This led to incorrect results from a number of geometric operators.

- Fix \texttt{money} data type to work in locales where currency amounts have no fractional digits, e.g. Japan (Itagaki Takahiro)

- Properly round datetime input like \texttt{00:12:57.9999999999999999999999999} (Tom)

- Fix poor choice of page split point in GiST R-tree operator classes (Teodor)

- Fix portability issues in plperl initialization (Andrew Dunstan)

- Improve robustness of \texttt{libpq}’s code to recover from errors during \texttt{COPY FROM STDIN} (Tom)

- Avoid including conflicting readline and editline header files when both libraries are installed (Zdenek Kotala)

E.105. Release 7.4.25

\textbf{Release date}: 2009-03-16
This release contains a variety of fixes from 7.4.24. For information about new features in the 7.4 major release, see Section E.130.

**E.105.1. Migration to Version 7.4.25**

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

**E.105.2. Changes**

- Prevent error recursion crashes when encoding conversion fails (Tom)
  
  This change extends fixes made in the last two minor releases for related failure scenarios. The previous fixes were narrowly tailored for the original problem reports, but we have now recognized that any error thrown by an encoding conversion function could potentially lead to infinite recursion while trying to report the error. The solution therefore is to disable translation and encoding conversion and report the plain-ASCII form of any error message, if we find we have gotten into a recursive error reporting situation. (CVE-2009-0922)

- Disallow `CREATE CONVERSION` with the wrong encodings for the specified conversion function (Heikki)
  
  This prevents one possible scenario for encoding conversion failure. The previous change is a backstop to guard against other kinds of failures in the same area.

- Fix core dump when `to_char()` is given format codes that are inappropriate for the type of the data argument (Tom)

- Add **MUST** (Mauritius Island Summer Time) to the default list of known timezone abbreviations (Xavier Bugaud)

**E.106. Release 7.4.24**

**Release date:** 2009-02-02

This release contains a variety of fixes from 7.4.23. For information about new features in the 7.4 major release, see Section E.130.

**E.106.1. Migration to Version 7.4.24**

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.
Appendix E. Release Notes

E.106.2. Changes

- Improve handling of URLs in \texttt{headline()} function (Teodor)
- Improve handling of overlength headlines in \texttt{headline()} function (Teodor)
- Prevent possible Assert failure or misconversion if an encoding conversion is created with the wrong conversion function for the specified pair of encodings (Tom, Heikki)
- Avoid unnecessary locking of small tables in \texttt{VACUUM} (Heikki)
- Fix uninitialized variables in \texttt{contrib/tsearch2’s get_covers()} function (Teodor)
- Fix bug in \texttt{to_char()’s handling of TH format codes} (Andreas Scherbaum)
- Make all documentation reference \texttt{pgsql-bugs} and/or \texttt{pgsql-hackers} as appropriate, instead of the now-decommissioned \texttt{pgsql-ports} and \texttt{pgsql-patches} mailing lists (Tom)

E.107. Release 7.4.23

\textbf{Release date:} 2008-11-03

This release contains a variety of fixes from 7.4.22. For information about new features in the 7.4 major release, see Section E.130.

E.107.1. Migration to Version 7.4.23

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.107.2. Changes

- Fix backend crash when the client encoding cannot represent a localized error message (Tom)
  We have addressed similar issues before, but it would still fail if the “character has no equivalent” message itself couldn’t be converted. The fix is to disable localization and send the plain ASCII error message when we detect such a situation.
- Fix incorrect \texttt{tsearch2} headline generation when single query item matches first word of text (Sushant Sinha)
- Fix improper display of fractional seconds in interval values when using a non-ISO datestyle in an \texttt{--enable-integer-datetimes build} (Ron Mayer)
- Ensure \texttt{SPI_getvalue} and \texttt{SPI_getbinval} behave correctly when the passed tuple and tuple descriptor have different numbers of columns (Tom)
  This situation is normal when a table has had columns added or removed, but these two functions didn’t handle it properly. The only likely consequence is an incorrect error indication.
- Fix ecpg’s parsing of \texttt{CREATE USER} (Michael)
Appendix E. Release Notes

E.108. Release 7.4.22

Release date: 2008-09-22

This release contains a variety of fixes from 7.4.21. For information about new features in the 7.4 major release, see Section E.130.

E.108.1. Migration to Version 7.4.22

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.108.2. Changes

- Fix datetime input functions to correctly detect integer overflow when running on a 64-bit platform (Tom)
- Improve performance of writing very long log messages to syslog (Tom)
- Fix bug in backwards scanning of a cursor on a SELECT DISTINCT ON query (Tom)
- Fix planner to estimate that GROUP BY expressions yielding boolean results always result in two groups, regardless of the expressions’ contents (Tom)
  
  This is very substantially more accurate than the regular GROUP BY estimate for certain boolean tests like \texttt{col IS NULL}.
- Improve pg_dump and pg_restore’s error reporting after failure to send a SQL command (Tom)

E.109. Release 7.4.21

Release date: 2008-06-12

This release contains one serious bug fix over 7.4.20. For information about new features in the 7.4 major release, see Section E.130.

E.109.1. Migration to Version 7.4.21

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.
E.109.2. Changes

- Make `pg_get_ruledef()` parenthesize negative constants (Tom)

Before this fix, a negative constant in a view or rule might be dumped as, say, `−42::integer`, which is subtly incorrect: it should be `(-42)::integer` due to operator precedence rules. Usually this would make little difference, but it could interact with another recent patch to cause PostgreSQL to reject what had been a valid `SELECT DISTINCT` view query. Since this could result in `pg_dump` output failing to reload, it is being treated as a high-priority fix. The only released versions in which dump output is actually incorrect are 8.3.1 and 8.2.7.

E.110. Release 7.4.20

Release date: never released

This release contains a variety of fixes from 7.4.19. For information about new features in the 7.4 major release, see Section E.130.

E.110.1. Migration to Version 7.4.20

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.110.2. Changes

- Fix conversions between ISO-8859-5 and other encodings to handle Cyrillic “Yo” characters (e and E with two dots) (Sergey Burladyan)
- Fix a few datatype input functions that were allowing unused bytes in their results to contain uninitialized, unpredictable values (Tom)
  
  This could lead to failures in which two apparently identical literal values were not seen as equal, resulting in the parser complaining about unmatched ORDER BY and DISTINCT expressions.
- Fix a corner case in regular-expression substring matching (`substring(string from pattern)`) (Tom)
  
  The problem occurs when there is a match to the pattern overall but the user has specified a parenthesized subexpression and that subexpression hasn’t got a match. An example is `substring('foo' from 'foo(bar)?')`. This should return NULL, since `(bar)` isn’t matched, but it was mistakenly returning the whole-pattern match instead (ie, foo).
- Fix incorrect result from ecpg’s `PGTYPEStimestamp_sub()` function (Michael)
- `DatumGetBool` macro to not fail with gcc 4.3 (Tom)
  
  This problem affects “old style” (V0) C functions that return boolean. The fix is already in 8.3, but the need to back-patch it was not realized at the time.
- Fix longstanding LISTEN/NOTIFY race condition (Tom)
Appendix E. Release Notes

In rare cases a session that had just executed a \texttt{LISTEN} might not get a notification, even though one would be expected because the concurrent transaction executing \texttt{NOTIFY} was observed to commit later.

A side effect of the fix is that a transaction that has executed a not-yet-committed \texttt{LISTEN} command will not see any row in \texttt{pg_listener} for the \texttt{LISTEN}, should it choose to look; formerly it would have. This behavior was never documented one way or the other, but it is possible that some applications depend on the old behavior.

- Fix display of constant expressions in \texttt{ORDER BY} and \texttt{GROUP BY} (Tom)
  
  An explicitly casted constant would be shown incorrectly. This could for example lead to corruption of a view definition during dump and reload.

- Fix libpq to handle \texttt{NOTICE} messages correctly during COPY OUT (Tom)
  
  This failure has only been observed to occur when a user-defined datatype’s output routine issues a \texttt{NOTICE}, but there is no guarantee it couldn’t happen due to other causes.

\section*{E.111. Release 7.4.19}

\textbf{Release date:} 2008-01-07

This release contains a variety of fixes from 7.4.18, including fixes for significant security issues. For information about new features in the 7.4 major release, see Section E.130.

\subsection*{E.111.1. Migration to Version 7.4.19}

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

\subsection*{E.111.2. Changes}

- Prevent functions in indexes from executing with the privileges of the user running \texttt{VACUUM}, \texttt{ANALYZE}, etc (Tom)

  Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as \texttt{VACUUM FULL}, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including \texttt{VACUUM}, \texttt{ANALYZE}, \texttt{REINDEX}, and \texttt{CLUSTER}) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for \texttt{SECURITY DEFINER} functions. To prevent bypassing this security measure, execution of \texttt{SET SESSION AUTHORIZATION} and \texttt{SET ROLE} is now forbidden within a \texttt{SECURITY DEFINER} context. (CVE-2007-6600)
• Repair assorted bugs in the regular-expression package (Tom, Will Drewry)
  Suitably crafted regular-expression patterns could cause crashes, infinite or near-infinite looping, and/or massive memory consumption, all of which pose denial-of-service hazards for applications that accept regex search patterns from untrustworthy sources. (CVE-2007-4769, CVE-2007-4772, CVE-2007-6067)

• Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)
  The fix that appeared for this in 7.4.18 was incomplete, as it plugged the hole for only some dblink functions. (CVE-2007-6601, CVE-2007-3278)

• Fix planner failure in some cases of WHERE false AND var IN (SELECT ...) (Tom)

• Fix potential crash in translate() when using a multibyte database encoding (Tom)

• Fix PL/Python to not crash on long exception messages (Alvaro)

• ecpg parser fixes (Michael)

• Make contrib/tablefunc's crosstab() handle NULL rowid as a category in its own right, rather than crashing (Joe)

• Fix tsvector and tsquery output routines to escape backslashes correctly (Teodor, Bruce)

• Fix crash of to_tsvector() on huge input strings (Teodor)

• Require a specific version of Autoconf to be used when re-generating the configure script (Peter)
  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it’s your responsibility whether the result works or not.

E.112. Release 7.4.18

Release date: 2007-09-17

This release contains fixes from 7.4.17. For information about new features in the 7.4 major release, see Section E.130.

E.112.1. Migration to Version 7.4.18

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.112.2. Changes

• Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent VACUUM on the same table (Tom)

• Make CREATE DOMAIN ... DEFAULT NULL work properly (Tom)
Appendix E. Release Notes

- Fix excessive logging of SSL error messages (Tom)
- Fix crash when log_min_error_statement logging runs out of memory (Tom)
- Prevent CLUSTER from failing due to attempting to process temporary tables of other sessions (Alvaro)
- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

### E.113. Release 7.4.17

**Release date:** 2007-04-23

This release contains fixes from 7.4.16, including a security fix. For information about new features in the 7.4 major release, see Section E.130.

#### E.113.1. Migration to Version 7.4.17

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

#### E.113.2. Changes

- Support explicit placement of the temporary-table schema within search_path, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of search_path. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See `CREATE FUNCTION` for more information.

- /contrib/tsearch2 crash fixes (Teodor)
- Fix potential-data-corruption bug in how VACUUM FULL handles UPDATE chains (Tom, Pavan Deolasee)
- Fix PANIC during enlargement of a hash index (bug introduced in 7.4.15) (Tom)

### E.114. Release 7.4.16

**Release date:** 2007-02-05

This release contains a variety of fixes from 7.4.15, including a security fix. For information about new features in the 7.4 major release, see Section E.130.
E.114.1. Migration to Version 7.4.16

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.114.2. Changes

- Remove security vulnerability that allowed connected users to read backend memory (Tom)
  The vulnerability involves suppressing the normal check that a SQL function returns the data type it’s declared to, or changing the data type of a table column used in a SQL function (CVE-2007-0555). This error can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
- Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
- Fix for rare Assert() crash triggered by UNION (Tom)
- Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.115. Release 7.4.15

Release date: 2007-01-08

This release contains a variety of fixes from 7.4.14. For information about new features in the 7.4 major release, see Section E.130.

E.115.1. Migration to Version 7.4.15

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.115.2. Changes

- Improve handling of getaddrinfo() on AIX (Tom)
  This fixes a problem with starting the statistics collector, among other things.
- Fix “failed to re-find parent key” errors in VACUUM (Tom)
- Fix bugs affecting multi-gigabyte hash indexes (Tom)
- Fix error when constructing an ARRAY[] made up of multiple empty elements (Tom)
- to_number() and to_char(numeric) are now STABLE, not IMMUTABLE, for new initdb installs (Tom)
This is because lc_numeric can potentially change the output of these functions.

- Improve index usage of regular expressions that use parentheses (Tom)
  This improves psql \d performance also.

### E.116. Release 7.4.14

**Release date:** 2006-10-16

This release contains a variety of fixes from 7.4.13. For information about new features in the 7.4 major release, see Section E.130.

#### E.116.1. Migration to Version 7.4.14

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

#### E.116.2. Changes

- Fix core dump when an untyped literal is taken as ANYARRAY
- Fix `string_to_array()` to handle overlapping matches for the separator string
  
  For example, `string_to_array('123xx456xxx789', 'xx')`.
- Fix corner cases in pattern matching for psql's \d commands
- Fix index-corrupting bugs in /contrib/ltree (Teodor)
- Fix backslash escaping in /contrib/dbmirror
- Adjust regression tests for recent changes in US DST laws

### E.117. Release 7.4.13

**Release date:** 2006-05-23

This release contains a variety of fixes from 7.4.12, including patches for extremely serious security issues. For information about new features in the 7.4 major release, see Section E.130.

#### E.117.1. Migration to Version 7.4.13

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.
Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq’s PQescapeStringConn()) to perform string escaping, rather than relying on ad hoc code to do it.

E.117.2. Changes

• Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)
  While PostgreSQL has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.

• Reject unsafe uses of ‘\’ in string literals
  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts ‘” and not ‘\’ as a representation of ASCII single quote in SQL string literals. By default, ‘\’ is rejected only when client_encoding is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter backslash_quote is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of backslash_quote is in part to make it obvious that insecure clients are insecure.

• Modify libpq’s string-escaping routines to be aware of encoding considerations and standard_conforming_strings
  This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314, and also future-proofs them against the planned changeover to SQL-standard string literal syntax. Applications that use multiple PostgreSQL connections concurrently should migrate to PQescapeStringConn() and PQescapeByteaConn() to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping “by hand” should be modified to rely on library routines instead.

• Fix some incorrect encoding conversion functions
  win1251_to_iso, alt_to_iso, euc_tw_to_big5, euc_tw_to_mic, mic_to_euc_tw were all broken to varying extents.

• Clean up stray remaining uses of ‘\’ in strings (Bruce, Jan)

• Fix bug that sometimes caused OR’d index scans to miss rows they should have returned

• Fix WAL replay for case where a btree index has been truncated

• Fix SIMILAR TO for patterns involving | (Tom)

• Fix server to use custom DH SSL parameters correctly (Michael Fuhr)

• Fix for Bonjour on Intel Macs (Ashley Clark)

• Fix various minor memory leaks
E.118. Release 7.4.12

Release date: 2006-02-14

This release contains a variety of fixes from 7.4.11. For information about new features in the 7.4 major release, see Section E.130.

E.118.1. Migration to Version 7.4.12

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.11, see the release notes for 7.4.11.

E.118.2. Changes

• Fix potential crash in SET SESSION AUTHORIZATION (CVE-2006-0553)
  An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.
• Fix bug with row visibility logic in self-inserted rows (Tom)
  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 7.4.9 and 7.3.11 releases.
• Fix race condition that could lead to “file already exists” errors during pg_clog file creation (Tom)
• Properly check DOMAIN constraints forUNKNOWNparameters in prepared statements (Neil)
• Fix to allow restoring dumps that have cross-schema references to custom operators (Tom)
• Portability fix for testing presence of finite and isinf during configure (Tom)

E.119. Release 7.4.11

Release date: 2006-01-09

This release contains a variety of fixes from 7.4.10. For information about new features in the 7.4 major release, see Section E.130.

E.119.1. Migration to Version 7.4.11

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see the release notes for 7.4.8. Also, you might need to REINDEX indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.
E.119.2. Changes

- Fix for protocol-level Describe messages issued outside a transaction or in a failed transaction (Tom)
- Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)
  
  This might require \texttt{REINDEX} to fix existing indexes on textual columns.
- Set locale environment variables during postmaster startup to ensure that \texttt{plperl} won’t change the locale later
  
  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what \texttt{initdb} had been told. Under these conditions, any use of \texttt{plperl} was likely to lead to corrupt indexes. You might need \texttt{REINDEX} to fix existing indexes on textual columns if this has happened to you.
- Fix longstanding bug in \texttt{strpos()} and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
- Fix bug in \texttt{/contrib/pgcrypto gen_salt}, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)
  
  Salts for Blowfish and standard DES are unaffected.
- Fix \texttt{/contrib/dblink} to throw an error, rather than crashing, when the number of columns specified is different from what’s actually returned by the query (Joe)

E.120. Release 7.4.10

\textbf{Release date}: 2005-12-12

This release contains a variety of fixes from 7.4.9. For information about new features in the 7.4 major release, see Section E.130.

E.120.1. Migration to Version 7.4.10

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see the release notes for 7.4.8.

E.120.2. Changes

- Fix race condition in transaction log management
  
  There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.
- Prevent failure if client sends Bind protocol message when current transaction is already aborted
- \texttt{/contrib/ltree} fixes (Teodor)
Appendix E. Release Notes

- AIX and HPUX compile fixes (Tom)
- Fix longstanding planning error for outer joins
  This bug sometimes caused a bogus error “RIGHT JOIN is only supported with merge-joinable join conditions”.
- Prevent core dump in pg_autovacuum when a table has been dropped

E.121. Release 7.4.9

Release date: 2005-10-04

This release contains a variety of fixes from 7.4.8. For information about new features in the 7.4 major release, see Section E.130.

E.121.1. Migration to Version 7.4.9

A dump/restore is not required for those running 7.4.X. However, if you are upgrading from a version earlier than 7.4.8, see the release notes for 7.4.8.

E.121.2. Changes

- Fix error that allowed VACUUM to remove ctid chains too soon, and add more checking in code that follows ctid links
  This fixes a long-standing problem that could cause crashes in very rare circumstances.
- Fix CHAR() to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)
  In prior releases, the padding of CHAR() was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.
- Fix the sense of the test for read-only transaction in COPY
  The code formerly prohibited COPY TO, where it should prohibit COPY FROM.
- Fix planning problem with outer-join ON clauses that reference only the inner-side relation
- Further fixes for x FULL JOIN y ON true corner cases
- Make array_in and array_recv more paranoid about validating their OID parameter
- Fix missing rows in queries like UPDATE a=... WHERE a... with GiST index on column a
- Improve robustness of datetime parsing
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
- Don’t try to open more than max_files_per_process files during postmaster startup
- Various memory leakage fixes
Appendix E. Release Notes

- Various portability improvements
- Fix PL/pgSQL to handle `var := var` correctly when the variable is of pass-by-reference type
- Update `contrib/tsearch2` to use current Snowball code

E.122. Release 7.4.8

Release date: 2005-05-09

This release contains a variety of fixes from 7.4.7, including several security-related issues. For information about new features in the 7.4 major release, see Section E.130.

E.122.1. Migration to Version 7.4.8

A dump/restore is not required for those running 7.4.X. However, it is one possible way of handling two significant security problems that have been found in the initial contents of 7.4.X system catalogs. A dump/initdb/reload sequence using 7.4.8’s initdb will automatically correct these problems.

The larger security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.)

The lesser problem is that the `contrib/tsearch2` module creates several functions that are misdeclared to return `internal` when they do not accept `internal` arguments. This breaks type safety for all functions using `internal` arguments.

It is strongly recommended that all installations repair these errors, either by initdb or by following the manual repair procedures given below. The errors at least allow unprivileged database users to crash their server process, and might allow unprivileged users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the following procedures instead. As the database superuser, do:

```sql
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
   AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

Next, if you have installed `contrib/tsearch2`, do:

```sql
BEGIN;
UPDATE pg_proc SET proargtypes[0] = 'internal'::regtype
WHERE oid IN ("dex_init(text)")::regprocedure,
    "snb_en_init(text)")::regprocedure,
    "snb_ru_init(text)")::regprocedure,
    "snb_ru_init(text)")::regprocedure,
```

1676
Appendix E. Release Notes

`'spell_init(text)'::regprocedure,
'syn_init(text)'::regprocedure
);`  
-- The command should report having updated 5 rows;  
-- if not, rollback and investigate instead of committing!  
COMMIT;

If this command fails with a message like “function "dex_init(text)" does not exist”, then either tsearch2 is not installed in this database, or you already did the update.

The above procedures must be carried out in each database of an installation, including template1, and ideally including template0 as well. If you do not fix the template databases then any subsequently created databases will contain the same errors. template1 can be fixed in the same way as any other database, but fixing template0 requires additional steps. First, from any database issue:

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to template0 and perform the above repair procedures. Finally, do:

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.122.2. Changes

- Change encoding function signature to prevent misuse
- Change contrib/tsearch2 to avoid unsafe use of INTERNAL function results
- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes
  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM
  This could theoretically have caused loss of a page’s worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix comparisons of TIME WITH TIME ZONE values
  The comparison code was wrong in the case where the --enable-integer-datetimes configuration switch had been used. NOTE: if you have an index on a TIME WITH TIME ZONE column, it will need to be REINDEXED after installing this update, because the fix corrects the sort order of column values.
- Fix EXTRACT(EPOCH) for TIME WITH TIME ZONE values
- Fix mis-display of negative fractional seconds in INTERVAL values
  This error only occurred when the --enable-integer-datetimes configuration switch had been used.
- Ensure operations done during backend shutdown are counted by statistics collector
Appendix E. Release Notes

This is expected to resolve reports of pg_autovacuum not vacuuming the system catalogs often enough — it was not being told about catalog deletions caused by temporary table removal during backend exit.

- Additional buffer overrun checks in plpgsql (Neil)
- Fix pg_dump to dump trigger names containing % correctly (Neil)
- Fix contrib/pgcrypto for newer OpenSSL builds (Marko Kreen)
- Still more 64-bit fixes for contrib/intagg
- Prevent incorrect optimization of functions returning RECORD
- Prevent to_char(interval) from dumping core for month-related formats
- Prevent crash on COALESCE(NULL,NULL)
- Fix array_map to call PL functions correctly
- Fix permission checking in ALTER DATABASE RENAME
- Fix ALTER LANGUAGE RENAME
- Make RemoveFromWaitQueue clean up after itself
  This fixes a lock management error that would only be visible if a transaction was kicked out of a wait for a lock (typically by query cancel) and then the holder of the lock released it within a very narrow window.
- Fix problem with untyped parameter appearing in INSERT ... SELECT
- Fix CLUSTER failure after ALTER TABLE SET WITHOUT OIDS

E.123. Release 7.4.7

Release date: 2005-01-31

This release contains a variety of fixes from 7.4.6, including several security-related issues. For information about new features in the 7.4 major release, see Section E.130.

E.123.1. Migration to Version 7.4.7

A dump/restore is not required for those running 7.4.X.

E.123.2. Changes

- Disallow LOAD to non-superusers

  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), LOAD can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.

- Check that creator of an aggregate function has the right to execute the specified transition functions
This oversight made it possible to bypass denial of EXECUTE permission on a function.

- Fix security and 64-bit issues in contrib/intagg
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins

The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- Fix plperl for quote marks in tuple fields
- Fix display of negative intervals in SQL and GERMAN datestyles
- Make age(timestamptz) do calculation in local timezone not GMT

### E.124. Release 7.4.6

**Release date:** 2004-10-22

This release contains a variety of fixes from 7.4.5. For information about new features in the 7.4 major release, see Section E.130.

#### E.124.1. Migration to Version 7.4.6

A dump/restore is not required for those running 7.4.X.

#### E.124.2. Changes

- Repair possible failure to update hint bits on disk
  Under rare circumstances this oversight could lead to “could not access transaction status” failures, which qualifies it as a potential-data-loss bug.
- Ensure that hashed outer join does not miss tuples
  Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.
- Disallow running pg_ctl as root
  This is to guard against any possible security issues.
- Avoid using temp files in /tmp in make_oidjoins_check
  This has been reported as a security issue, though it’s hardly worthy of concern since there is no reason for non-developers to use this script anyway.
- Prevent forced backend shutdown from re-emitting prior command result
In rare cases, a client might think that its last command had succeeded when it really had been aborted by forced database shutdown.

- Repair bug in `pg_stat_get_backend_idset`
  This could lead to misbehavior in some of the system-statistics views.
- Fix small memory leak in postmaster
- Fix "expected both swapped tables to have TOAST tables" bug
  This could arise in cases such as CLUSTER after ALTER TABLE DROP COLUMN.
- Prevent `pg_ctl restart` from adding `-D` multiple times
- Fix problem with NULL values in GiST indexes
- `::` is no longer interpreted as a variable in an ECPG prepare statement

### E.125. Release 7.4.5

**Release date:** 2004-08-18

This release contains one serious bug fix over 7.4.4. For information about new features in the 7.4 major release, see Section E.130.

#### E.125.1. Migration to Version 7.4.5

A dump/restore is not required for those running 7.4.X.

#### E.125.2. Changes

- Repair possible crash during concurrent B-tree index insertions
  This patch fixes a rare case in which concurrent insertions into a B-tree index could result in a server panic. No permanent damage would result, but it’s still worth a re-release. The bug does not exist in pre-7.4 releases.

### E.126. Release 7.4.4

**Release date:** 2004-08-16

This release contains a variety of fixes from 7.4.3. For information about new features in the 7.4 major release, see Section E.130.
E.126.1. Migration to Version 7.4.4
A dump/restore is not required for those running 7.4.X.

E.126.2. Changes

• Prevent possible loss of committed transactions during crash
  Due to insufficient interlocking between transaction commit and checkpointing, it was possible
  for transactions committed just before the most recent checkpoint to be lost, in whole or in part,
  following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.
• Check HAVING restriction before evaluating result list of an aggregate plan
• Avoid crash when session’s current user ID is deleted
• Fix hashed crosstab for zero-rows case (Joe)
• Force cache update after renaming a column in a foreign key
• Pretty-print UNION queries correctly
• Make psql handle \r\n newlines properly in COPY IN
• pg_dump handled ACLs with grant options incorrectly
• Fix thread support for OS X and Solaris
• Updated JDBC driver (build 215) with various fixes
• ECPG fixes
• Translation updates (various contributors)

E.127. Release 7.4.3

Release date: 2004-06-14

This release contains a variety of fixes from 7.4.2. For information about new features in the 7.4 major
release, see Section E.130.

E.127.1. Migration to Version 7.4.3
A dump/restore is not required for those running 7.4.X.

E.127.2. Changes

• Fix temporary memory leak when using non-hashed aggregates (Tom)
• ECPG fixes, including some for Informix compatibility (Michael)
• Fixes for compiling with thread-safety, particularly Solaris (Bruce)
Appendix E. Release Notes

- Fix error in COPY IN termination when using the old network protocol (ljb)
- Several important fixes in pg-autovacuum, including fixes for large tables, unsigned oids, stability, temp tables, and debug mode (Matthew T. O’Connor)
- Fix problem with reading tar-format dumps on NetBSD and BSD/OS (Bruce)
- Several JDBC fixes
- Fix ALTER SEQUENCE RESTART where last_value equals the restart value (Tom)
- Repair failure to recalculate nested sub-selects (Tom)
- Fix problems with non-constant expressions in LIMIT/OFFSET
- Support FULL JOIN with no join clause, such as X FULL JOIN Y ON TRUE (Tom)
- Fix another zero-column table bug (Tom)
- Improve handling of non-qualified identifiers in GROUP BY clauses in sub-selects (Tom)
  Select-list aliases within the sub-select will now take precedence over names from outer query levels.
- Do not generate “NATURAL CROSS JOIN” when decompiling rules (Tom)
- Add checks for invalid field length in binary COPY (Tom)
  This fixes a difficult-to-exploit security hole.
- Avoid locking conflict between ANALYZE and LISTEN/NOTIFY
- Numerous translation updates (various contributors)

E.128. Release 7.4.2

Release date: 2004-03-08

This release contains a variety of fixes from 7.4.1. For information about new features in the 7.4 major release, see Section E.130.

E.128.1. Migration to Version 7.4.2

A dump/restore is not required for those running 7.4.X. However, it might be advisable as the easiest method of incorporating fixes for two errors that have been found in the initial contents of 7.4.X system catalogs. A dump/initdb/reload sequence using 7.4.2’s initdb will automatically correct these problems.

The more severe of the two errors is that data type anyarray has the wrong alignment label; this is a problem because the pg_statistic system catalog uses anyarray columns. The mislabeling can cause planner misestimations and even crashes when planning queries that involve WHERE clauses on double-aligned columns (such as float8 and timestamp). It is strongly recommended that all installations repair this error, either by initdb or by following the manual repair procedure given below.

The lesser error is that the system view pg_settings ought to be marked as having public update access, to allow UPDATE pg_settings to be used as a substitute for SET. This can also be fixed either by initdb or manually, but it is not necessary to fix unless you want to use UPDATE pg_settings.
Appendix E. Release Notes

If you wish not to do an initdb, the following procedure will work for fixing pg_statistic. As the database superuser, do:

```sql
-- clear out old data in pg_statistic:
DELETE FROM pg_statistic;
VACUUM pg_statistic;
-- this should update 1 row:
UPDATE pg_type SET typalign = 'd' WHERE oid = 2277;
-- this should update 6 rows:
UPDATE pg_attribute SET attalign = 'd' WHERE atttypid = 2277;
--
-- At this point you MUST start a fresh backend to avoid a crash!
--
-- repopulate pg_statistic:
ANALYZE;
```

This can be done in a live database, but beware that all backends running in the altered database must be restarted before it is safe to repopulate pg_statistic.

To repair the pg_settings error, simply do:

```sql
GRANT SELECT, UPDATE ON pg_settings TO PUBLIC;
```

The above procedures must be carried out in each database of an installation, including template1, and ideally including template0 as well. If you do not fix the template databases then any subsequently created databases will contain the same errors. template1 can be fixed in the same way as any other database, but fixing template0 requires additional steps. First, from any database issue:

```sql
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to template0 and perform the above repair procedures. Finally, do:

```sql
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.128.2. Changes

Release 7.4.2 incorporates all the fixes included in release 7.3.6, plus the following fixes:

- Fix pg_statistics alignment bug that could crash optimizer
  See above for details about this problem.
- Allow non-super users to update pg_settings
- Fix several optimizer bugs, most of which led to “variable not found in subplan target lists” errors
- Avoid out-of-memory failure during startup of large multiple index scan
- Fix multibyte problem that could lead to “out of memory” error during COPY IN
- Fix problems with SELECT INTO/CREATE TABLE AS from tables without OIDs
- Fix problems with alter_table regression test during parallel testing
Appendix E. Release Notes

- Fix problems with hitting open file limit, especially on OS X (Tom)
- Partial fix for Turkish-locale issues
  initdb will succeed now in Turkish locale, but there are still some inconveniences associated with the i/I problem.
- Make pg_dump set client encoding on restore
- Other minor pg_dump fixes
- Allow ecpg to again use C keywords as column names (Michael)
- Added ecpg WHENEVER NOT_FOUND to SELECT/INSERT/UPDATE/DELETE (Michael)
- Fix ecpg crash for queries calling set-returning functions (Michael)
- Various other ecpg fixes (Michael)
- Fixes for Borland compiler
- Thread build improvements (Bruce)
- Various other build fixes
- Various JDBC fixes

E.129. Release 7.4.1

Release date: 2003-12-22

This release contains a variety of fixes from 7.4. For information about new features in the 7.4 major release, see Section E.130.

E.129.1. Migration to Version 7.4.1

A dump/restore is not required for those running 7.4.

If you want to install the fixes in the information schema you need to reload it into the database. This is either accomplished by initializing a new cluster by running initdb, or by running the following sequence of SQL commands in each database (ideally including template1) as a superuser in psql, after installing the new release:

```
DROP SCHEMA information_schema CASCADE;
\i /usr/local/pgsql/share/information_schema.sql
```

Substitute your installation path in the second command.

E.129.2. Changes

- Fixed bug in CREATE SCHEMA parsing in ECPG (Michael)
- Fix compile error when --enable-thread-safety and --with-perl are used together (Peter)
- Fix for subqueries that used hash joins (Tom)
Certain subqueries that used hash joins would crash because of improperly shared structures.

- Fix free space map compaction bug (Tom)
  This fixes a bug where compaction of the free space map could lead to a database server shutdown.
- Fix for Borland compiler build of libpq (Bruce)
- Fix netmask() and hostmask() to return the maximum-length masklen (Tom)
  Fix these functions to return values consistent with pre-7.4 releases.
- Several contrib/pg_autovacuum fixes
  Fixes include improper variable initialization, missing vacuum after TRUNCATE, and duration computation overflow for long vacuums.
- Allow compile of contrib/cube under Cygwin (Jason Tishler)
- Fix Solaris use of password file when no passwords are defined (Tom)
  Fix crash on Solaris caused by use of any type of password authentication when no passwords were defined.
- JDBC fix for thread problems, other fixes
- Fix for bytea index lookups (Joe)
- Fix information schema for bit data types (Peter)
- Force zero_damaged_pages to be on during recovery from WAL
- Prevent some obscure cases of “variable not in subplan target lists”
- Make PQescapeBytea and byteaout consistent with each other (Joe)
- Escape bytea output for bytes > 0x7e (Joe)
  If different client encodings are used for bytea output and input, it is possible for bytea values to be corrupted by the differing encodings. This fix escapes all bytes that might be affected.
- Added missing SPI_finish() calls to dblink’s get_tuple_of_interest() (Joe)
- New Czech FAQ
- Fix information schema view constraint_column_usage for foreign keys (Peter)
- ECPG fixes (Michael)
- Fix bug with multiple IN subqueries and joins in the subqueries (Tom)
- Allow COUNT(‘x’) to work (Tom)
- Install ECPG include files for Informix compatibility into separate directory (Peter)
  Some names of ECPG include files for Informix compatibility conflicted with operating system include files. By installing them in their own directory, name conflicts have been reduced.
- Fix SSL memory leak (Neil)
  This release fixes a bug in 7.4 where SSL didn’t free all memory it allocated.
- Prevent pg_service.conf from using service name as default dbname (Bruce)
- Fix local ident authentication on FreeBSD (Tom)
E.130. Release 7.4

Release date: 2003-11-17

E.130.1. Overview

Major changes in this release:

IN / NOT IN subqueries are now much more efficient

In previous releases, IN/NOT IN subqueries were joined to the upper query by sequentially scanning the subquery looking for a match. The 7.4 code uses the same sophisticated techniques used by ordinary joins and so is much faster. An IN will now usually be as fast as or faster than an equivalent EXISTS subquery; this reverses the conventional wisdom that applied to previous releases.

Improved GROUP BY processing by using hash buckets

In previous releases, rows to be grouped had to be sorted first. The 7.4 code can do GROUP BY without sorting, by accumulating results into a hash table with one entry per group. It will still use the sort technique, however, if the hash table is estimated to be too large to fit in sort_mem.

New multikey hash join capability

In previous releases, hash joins could only occur on single keys. This release allows multicolonm hash joins.

Queries using the explicit JOIN syntax are now better optimized

Prior releases evaluated queries using the explicit JOIN syntax only in the order implied by the syntax. 7.4 allows full optimization of these queries, meaning the optimizer considers all possible join orderings and chooses the most efficient. Outer joins, however, must still follow the declared ordering.

Faster and more powerful regular expression code

The entire regular expression module has been replaced with a new version by Henry Spencer, originally written for Tcl. The code greatly improves performance and supports several flavors of regular expressions.

Function-inlining for simple SQL functions

Simple SQL functions can now be inlined by including their SQL in the main query. This improves performance by eliminating per-call overhead. That means simple SQL functions now behave like macros.

Full support for IPv6 connections and IPv6 address data types

Previous releases allowed only IPv4 connections, and the IP data types only supported IPv4 addresses. This release adds full IPv6 support in both of these areas.

Major improvements in SSL performance and reliability

Several people very familiar with the SSL API have overhauled our SSL code to improve SSL key negotiation and error recovery.
Appendix E. Release Notes

Make free space map efficiently reuse empty index pages, and other free space management improvements.

In previous releases, B-tree index pages that were left empty because of deleted rows could only be reused by rows with index values similar to the rows originally indexed on that page. In 7.4, *VACUUM* records empty index pages and allows them to be reused for any future index rows.

SQL-standard information schema

The information schema provides a standardized and stable way to access information about the schema objects defined in a database.

Cursors conform more closely to the SQL standard

The commands *FETCH* and *MOVE* have been overhauled to conform more closely to the SQL standard.

Cursors can exist outside transactions

These cursors are also called holdable cursors.

New client-to-server protocol

The new protocol adds error codes, more status information, faster startup, better support for binary data transmission, parameter values separated from SQL commands, prepared statements available at the protocol level, and cleaner recovery from *COPY* failures. The older protocol is still supported by both server and clients.

libpq and ECPG applications are now fully thread-safe

While previous libpq releases already supported threads, this release improves thread safety by fixing some non-thread-safe code that was used during database connection startup. The configure option `--enable-thread-safety` must be used to enable this feature.

New version of full-text indexing

A new full-text indexing suite is available in `contrib/tsearch2`.

New autovacuum tool

The new autovacuum tool in `contrib/autovacuum` monitors the database statistics tables for `INSERT/UPDATE/DELETE` activity and automatically vacuums tables when needed.

Array handling has been improved and moved into the server core

Many array limitations have been removed, and arrays behave more like fully-supported data types.

E.130.2. Migration to Version 7.4

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- The server-side autocommit setting was removed and reimplemented in client applications and languages. Server-side autocommit was causing too many problems with languages and applications that wanted to control their own autocommit behavior, so autocommit was removed from the server and added to individual client APIs as appropriate.
- Error message wording has changed substantially in this release. Significant effort was invested to make the messages more consistent and user-oriented. If your applications try to detect different
error conditions by parsing the error message, you are strongly encouraged to use the new error code facility instead.

- Inner joins using the explicit JOIN syntax might behave differently because they are now better optimized.
- A number of server configuration parameters have been renamed for clarity, primarily those related to logging.
- FETCH 0 or MOVE 0 now does nothing. In prior releases, FETCH 0 would fetch all remaining rows, and MOVE 0 would move to the end of the cursor.
- FETCH and MOVE now return the actual number of rows fetched/moved, or zero if at the beginning/end of the cursor. Prior releases would return the row count passed to the command, not the number of rows actually fetched or moved.
- COPY now can process files that use carriage-return or carriage-return/line-feed end-of-line sequences. Literal carriage-returns and line-feeds are no longer accepted in data values; use \r and \n instead.
- Trailing spaces are now trimmed when converting from type char(n) to varchar(n) or text. This is what most people always expected to happen anyway.
- The data type float(p) now measures p in binary digits, not decimal digits. The new behavior follows the SQL standard.
- Ambiguous date values now must match the ordering specified by the datestyle setting. In prior releases, a date specification of 10/20/03 was interpreted as a date in October even if datestyle specified that the day should be first. 7.4 will throw an error if a date specification is invalid for the current setting of datestyle.
- The functions oidrand, oidsrand, and userfntest have been removed. These functions were determined to be no longer useful.
- String literals specifying time-varying date/time values, such as 'now' or 'today' will no longer work as expected in column default expressions; they now cause the time of the table creation to be the default, not the time of the insertion. Functions such as now(), current_timestamp, or current_date should be used instead.

In previous releases, there was special code so that strings such as 'now' were interpreted at INSERT time and not at table creation time, but this work around didn’t cover all cases. Release 7.4 now requires that defaults be defined properly using functions such as now() or current_timestamp. These will work in all situations.

- The dollar sign ($) is no longer allowed in operator names. It can instead be a non-first character in identifiers. This was done to improve compatibility with other database systems, and to avoid syntax problems when parameter placeholders ($n) are written adjacent to operators.

E.130.3. Changes

Below you will find a detailed account of the changes between release 7.4 and the previous major release.

E.130.3.1. Server Operation Changes

- Allow IPv6 server connections (Nigel Kukard, Johan Jordaan, Bruce, Tom, Kurt Roeckx, Andrew Dunstan)
• Fix SSL to handle errors cleanly (Nathan Mueller)
  In prior releases, certain SSL API error reports were not handled correctly. This release fixes those
  problems.
• SSL protocol security and performance improvements (Sean Chittenden)
  SSL key renegotiation was happening too frequently, causing poor SSL performance. Also, initial
  key handling was improved.
• Print lock information when a deadlock is detected (Tom)
  This allows easier debugging of deadlock situations.
• Update /tmp socket modification times regularly to avoid their removal (Tom)
  This should help prevent /tmp directory cleaner administration scripts from removing server socket
  files.
• Enable PAM for Mac OS X (Aaron Hillegass)
• Make B-tree indexes fully WAL-safe (Tom)
  In prior releases, under certain rare cases, a server crash could cause B-tree indexes to become
  corrupt. This release removes those last few rare cases.
• Allow B-tree index compaction and empty page reuse (Tom)
• Fix inconsistent index lookups during split of first root page (Tom)
  In prior releases, when a single-page index split into two pages, there was a brief period when
  another database session could miss seeing an index entry. This release fixes that rare failure case.
• Improve free space map allocation logic (Tom)
• Preserve free space information between server restarts (Tom)
  In prior releases, the free space map was not saved when the postmaster was stopped, so newly
  started servers had no free space information. This release saves the free space map, and reloads it
  when the server is restarted.
• Add start time to pg_stat_activity (Neil)
• New code to detect corrupt disk pages; erase with zero_damaged_pages (Tom)
• New client/server protocol: faster, no username length limit, allow clean exit from COPY (Tom)
• Add transaction status, table ID, column ID to client/server protocol (Tom)
• Add binary I/O to client/server protocol (Tom)
• Remove autocommit server setting; move to client applications (Tom)
• New error message wording, error codes, and three levels of error detail (Tom, Joe, Peter)

**E.130.3.2. Performance Improvements**

• Add hashing for GROUP BY aggregates (Tom)
• Make nested-loop joins be smarter about multicoloumn indexes (Tom)
• Allow multikey hash joins (Tom)
• Improve constant folding (Tom)
• Add ability to inline simple SQL functions (Tom)
Appendix E. Release Notes

- Reduce memory usage for queries using complex functions (Tom)
  In prior releases, functions returning allocated memory would not free it until the query completed. This release allows the freeing of function-allocated memory when the function call completes, reducing the total memory used by functions.

- Improve GEOQ optimizer performance (Tom)
  This release fixes several inefficiencies in the way the GEOQ optimizer manages potential query paths.

- Allow IN/NOT IN to be handled via hash tables (Tom)

- Improve NOT IN (subquery) performance (Tom)

- Allow most IN subqueries to be processed as joins (Tom)

- Pattern matching operations can use indexes regardless of locale (Peter)
  There is no way for non-ASCII locales to use the standard indexes for LIKE comparisons. This release adds a way to create a special index for LIKE.

- Allow the postmaster to preload libraries using preload_libraries (Joe)
  For shared libraries that require a long time to load, this option is available so the library can be preloaded in the postmaster and inherited by all database sessions.

- Improve optimizer cost computations, particularly for subqueries (Tom)

- Avoid sort when subquery ORDER BY matches upper query (Tom)

- Deduce that WHERE a.x = b.y AND b.y = 42 also means a.x = 42 (Tom)

- Allow hash/merge joins on complex joins (Tom)

- Allow hash joins for more data types (Tom)

- Allow join optimization of explicit inner joins, disable with joinCollapseLimit (Tom)

- Add parameter fromCollapseLimit to control conversion of subqueries to joins (Tom)

- Use faster and more powerful regular expression code from Tcl (Henry Spencer, Tom)

- Use bit-mapped relation sets in the optimizer (Tom)

- Improve connection startup time (Tom)
  The new client/server protocol requires fewer network packets to start a database session.

- Improve trigger/constraint performance (Stephan)

- Improve speed of col IN (const, const, const, ...) (Tom)

- Fix hash indexes which were broken in rare cases (Tom)

- Improve hash index concurrency and speed (Tom)
  Prior releases suffered from poor hash index performance, particularly for high concurrency situations. This release fixes that, and the development group is interested in reports comparing B-tree and hash index performance.

- Align shared buffers on 32-byte boundary for copy speed improvement (Manfred Spraul)
  Certain CPU’s perform faster data copies when addresses are 32-byte aligned.

- Data type numeric reimplemented for better performance (Tom)
  numeric used to be stored in base 100. The new code uses base 10000, for significantly better performance.
Appendix E. Release Notes

E.130.3.3. Server Configuration Changes

• Rename server parameter `server_min_messages` to `log_min_messages` (Bruce)
  This was done so most parameters that control the server logs begin with `log_`.
• Rename `show_*_stats` to `log_*_stats` (Bruce)
• Rename `show_source_port` to `log_source_port` (Bruce)
• Rename `hostname_lookup` to `log_hostname` (Bruce)
• Add `checkpoint_warning` to warn of excessive checkpointing (Bruce)
  In prior releases, it was difficult to determine if checkpoint was happening too frequently. This feature adds a warning to the server logs when excessive checkpointing happens.
• New read-only server parameters for localization (Tom)
• Change debug server log messages to output as `DEBUG` rather than `LOG` (Bruce)
• Prevent server log variables from being turned off by non-superusers (Bruce)
  This is a security feature so non-superusers cannot disable logging that was enabled by the administrator.
• `log_min_messages/client_min_messages` now controls `debug_*` output (Bruce)
  This centralizes client debug information so all debug output can be sent to either the client or server logs.
• Add Mac OS X Rendezvous server support (Chris Campbell)
  This allows Mac OS X hosts to query the network for available PostgreSQL servers.
• Add ability to print only slow statements using `log_min_duration_statement` (Christopher)
  This is an often requested debugging feature that allows administrators to see only slow queries in their server logs.
• Allow `pg_hba.conf` to accept netmasks in CIDR format (Andrew Dunstan)
  This allows administrators to merge the host IP address and netmask fields into a single CIDR field in `pg_hba.conf`.
• New read-only parameter `is_superuser` (Tom)
• New parameter `log_error_verbosity` to control error detail (Tom)
  This works with the new error reporting feature to supply additional error information like hints, file names and line numbers.
• `postgres --describe-config` now dumps server config variables (Aizaz Ahmed, Peter)
  This option is useful for administration tools that need to know the configuration variable names and their minimums, maximums, defaults, and descriptions.
• Add new columns in `pg_settings`: `context`, `type`, `source`, `min_val`, `max_val` (Joe)
• Make default `shared_buffers 1000` and `max_connections 100`, if possible (Tom)
  Prior versions defaulted to 64 shared buffers so PostgreSQL would start on even very old systems. This release tests the amount of shared memory allowed by the platform and selects more reasonable default values if possible. Of course, users are still encouraged to evaluate their resource load and size `shared_buffers` accordingly.
• New `pg_hba.conf` record type `hostnossl` to prevent SSL connections (Jon Jensen)
In prior releases, there was no way to prevent SSL connections if both the client and server supported SSL. This option allows that capability.

- Remove parameter `geqo_random_seed` (Tom)
- Add server parameter `regex_flavor` to control regular expression processing (Tom)
- Make `pg_ctl` better handle nonstandard ports (Greg)

### E.130.3.4. Query Changes

- New SQL-standard information schema (Peter)
- Add read-only transactions (Peter)
- Print key name and value in foreign-key violation messages (Dmitry Tkach)
- Allow users to see their own queries in `pg_stat_activity` (Kevin Brown)
  
  In prior releases, only the superuser could see query strings using `pg_stat_activity`. Now ordinary users can see their own query strings.
- Fix aggregates in subqueries to match SQL standard (Tom)
  
  The SQL standard says that an aggregate function appearing within a nested subquery belongs to the outer query if its argument contains only outer-query variables. Prior PostgreSQL releases did not handle this fine point correctly.
- Add option to prevent auto-addition of tables referenced in query (Nigel J. Andrews)
  
  By default, tables mentioned in the query are automatically added to the `FROM` clause if they are not already there. This is compatible with historic POSTGRES behavior but is contrary to the SQL standard. This option allows selecting standard-compatible behavior.
- Allow `UPDATE ... SET col = DEFAULT` (Rod)
  
  This allows `UPDATE` to set a column to its declared default value.
- Allow expressions to be used in `LIMIT`/`OFFSET` (Tom)
  
  In prior releases, `LIMIT`/`OFFSET` could only use constants, not expressions.
- Implement `CREATE TABLE AS EXECUTE` (Neil, Peter)

### E.130.3.5. Object Manipulation Changes

- Make `CREATE SEQUENCE` grammar more conforming to SQL:2003 (Neil)
- Add statement-level triggers (Neil)
  
  While this allows a trigger to fire at the end of a statement, it does not allow the trigger to access all rows modified by the statement. This capability is planned for a future release.
- Add check constraints for domains (Rod)
  
  This greatly increases the usefulness of domains by allowing them to use check constraints.
- Add `ALTER DOMAIN` (Rod)
  
  This allows manipulation of existing domains.
- Fix several zero-column table bugs (Tom)
Appendix E. Release Notes

PostgreSQL supports zero-column tables. This fixes various bugs that occur when using such tables.

- **Have** `ALTER TABLE ... ADD PRIMARY KEY add not-null constraint** (Rod)
  
  In prior releases, `ALTER TABLE ... ADD PRIMARY` would add a unique index, but not a not-null constraint. That is fixed in this release.

- **Add** `ALTER TABLE ... WITHOUT OIDS` (Rod)
  
  This allows control over whether new and updated rows will have an OID column. This is most useful for saving storage space.

- **Add** `ALTER SEQUENCE to modify minimum, maximum, increment, cache, cycle values` (Rod)

- **Add** `ALTER TABLE ... CLUSTER ON` (Alvaro Herrera)
  
  This command is used by `pg_dump` to record the cluster column for each table previously clustered. This information is used by database-wide cluster to cluster all previously clustered tables.

- **Improve** automatic type casting for domains (Rod, Tom)

- **Allow** dollar signs in identifiers, except as first character (Tom)

- **Disallow** dollar signs in operator names, so `x=$1` works (Tom)

- **Allow** copying table schema using `LIKE subtable`, also SQL:2003 feature `INCLUDING DEFAULTS` (Rod)

- **Add** `WITH GRANT OPTION` clause to `GRANT` (Peter)
  
  This enabled `GRANT` to give other users the ability to grant privileges on a object.

E.130.3.6. Utility Command Changes

- **Add** `ON COMMIT` clause to `CREATE TABLE` for temporary tables (Gavin)
  
  This adds the ability for a table to be dropped or all rows deleted on transaction commit.

- **Allow** cursors outside transactions using `WITH HOLD` (Neil)
  
  In previous releases, cursors were removed at the end of the transaction that created them. Cursors can now be created with the `WITH HOLD` option, which allows them to continue to be accessed after the creating transaction has committed.

- **FETCH 0 and MOVE 0 now do nothing** (Bruce)
  
  In previous releases, `FETCH 0` fetched all remaining rows, and `MOVE 0` moved to the end of the cursor.

- **Cause** `FETCH` and `MOVE` to return the number of rows fetched/moved, or zero if at the beginning/end of cursor, per SQL standard (Bruce)
  
  In prior releases, the row count returned by `FETCH` and `MOVE` did not accurately reflect the number of rows processed.

- **Properly handle** `SCROLL` with cursors, or report an error (Neil)
  
  Allowing random access (both forward and backward scrolling) to some kinds of queries cannot be done without some additional work. If `SCROLL` is specified when the cursor is created, this additional work will be performed. Furthermore, if the cursor has been created with `NO SCROLL`, no random access is allowed.

- **Implement** SQL-compatible options `FIRST`, `LAST`, `ABSOLUTE $n$, `RELATIVE $n` for `FETCH` and `MOVE` (Tom)
Appendix E. Release Notes

- Allow `EXPLAIN` on `DECLARE CURSOR` (Tom)
- Allow `CLUSTER` to use index marked as pre-clustered by default (Alvaro Herrera)
- Allow `CLUSTER` to cluster all tables (Alvaro Herrera)
  This allows all previously clustered tables in a database to be reclustered with a single command.
- Prevent `CLUSTER` on partial indexes (Tom)
- Allow DOS and Mac line-endings in `COPY` files (Bruce)
- Disallow literal carriage return as a data value, backslash carriage-return and `` are still allowed (Bruce)
- `COPY` changes (binary, `.`) (Tom)
- Recover from `COPY` failure cleanly (Tom)
- Prevent possible memory leaks in `COPY` (Tom)
- Make `TRUNCATE` transaction-safe (Rod)
  `TRUNCATE` can now be used inside a transaction. If the transaction aborts, the changes made by the `TRUNCATE` are automatically rolled back.
- Allow `prepare/bind` of utility commands like `FETCH` and `EXPLAIN` (Tom)
- Add `EXPLAIN EXECUTE` (Neil)
- Improve `VACUUM` performance on indexes by reducing WAL traffic (Tom)
- Functional indexes have been generalized into indexes on expressions (Tom)
  In prior releases, functional indexes only supported a simple function applied to one or more column names. This release allows any type of scalar expression.
- Have `SHOW TRANSACTION ISOLATION` match input to `SET TRANSACTION ISOLATION` (Tom)
- Have `COMMENT ON DATABASE` on nonlocal database generate a warning, rather than an error (Rod)
  Database comments are stored in database-local tables so comments on a database have to be stored in each database.
- Improve reliability of `LISTEN/NOTIFY` (Tom)
- Allow `REINDEX` to reliably reindex nonshared system catalog indexes (Tom)
  This allows system tables to be reindexed without the requirement of a standalone session, which was necessary in previous releases. The only tables that now require a standalone session for reindexing are the global system tables `pg_database`, `pg_shadow`, and `pg_group`.

E.130.3.7. Data Type and Function Changes

- New server parameter `extra_float_digits` to control precision display of floating-point numbers (Pedro Ferreira, Tom)
  This controls output precision which was causing regression testing problems.
- Allow `+1300` as a numeric time-zone specifier, for FJST (Tom)
- Remove rarely used functions `oidrand`, `oidsrand`, and `userfntest` functions (Neil)
- Add `md5()` function to main server, already in `contrib/pgcrypto` (Joe)
  An MD5 function was frequently requested. For more complex encryption capabilities, use `contrib/pgcrypto`.

1694
Appendix E. Release Notes

- Increase date range of timestamp (John Cochran)
- Change \texttt{EXTRACT(EPOCH FROM timestamp)} so timestamp without time zone is assumed to be in local time, not GMT (Tom)
- Trap division by zero in case the operating system doesn’t prevent it (Tom)
- Change the numeric data type internally to base 10000 (Tom)
- New \texttt{hostmask()} function (Greg Wickham)
- Fixes for \texttt{to_char()} and \texttt{to_timestamp()} (Karel)
- Allow functions that can take any argument data type and return any data type, using \texttt{anyelement} and \texttt{anyarray} (Joe)

This allows the creation of functions that can work with any data type.

- Arrays can now be specified as \texttt{ARRAY[1,2,3]}, \texttt{ARRAY[['a','b'],['c','d']]} or \texttt{ARRAY[ARRAY[ARRAY[2]]]} (Joe)
- Allow proper comparisons for arrays, including \texttt{ORDER BY} and \texttt{DISTINCT} support (Joe)
- Allow indexes on array columns (Joe)
- Allow array concatenation with || (Joe)
- Allow \texttt{WHERE} qualification \texttt{expr op ANY/SOME/ALL (array_expr)} (Joe)

This allows arrays to behave like a list of values, for purposes like \texttt{SELECT * FROM tab WHERE col IN (array_val)}.

- New array functions \texttt{array_append}, \texttt{array_cat}, \texttt{array_lower}, \texttt{array_prepend}, \texttt{array_to_string}, \texttt{array_upper}, \texttt{string_to_array} (Joe)
- Allow user defined aggregates to use polymorphic functions (Joe)
- Allow assignments to empty arrays (Joe)
- Allow 60 in seconds fields of \texttt{time}, \texttt{timestamp}, and \texttt{interval} input values (Tom)

Sixty-second values are needed for leap seconds.

- Allow \texttt{cidr} data type to be cast to \texttt{text} (Tom)
- Disallow invalid time zone names in \texttt{SET TIMEZONE}
- Trim trailing spaces when \texttt{char} is cast to \texttt{varchar} or \texttt{text} (Tom)
- Make \texttt{float(p)} measure the precision \texttt{p} in binary digits, not decimal digits (Tom)
- Add IPv6 support to the \texttt{inet} and \texttt{cidr} data types (Michael Graff)
- Add \texttt{family()} function to report whether address is IPv4 or IPv6 (Michael Graff)
- Have \texttt{SHOW} \texttt{datestyle} generate output similar to that used by \texttt{SET datestyle} (Tom)
- Make \texttt{EXTRACT(TIMEZONE)} and \texttt{SET/SHOW TIME ZONE} follow the SQL convention for the sign of time zone offsets, i.e., positive is east from UTC (Tom)
- Fix \texttt{date_trunc(‘quarter’, ...)} (Bójthe Zoltán)

Prior releases returned an incorrect value for this function call.

- Make \texttt{initcap()} more compatible with Oracle (Mike Nolan)

\texttt{initcap()} now uppercases a letter appearing after any non-alphanumeric character, rather than only after whitespace.

- Allow only \texttt{datestyle} field order for date values not in ISO-8601 format (Greg)
• Add new `datestyle` values `MDY`, `DMY`, and `YMD` to set input field order; honor US and European for backward compatibility (Tom)
• String literals like `'now'` or `'today'` will no longer work as a column default. Use functions such as `now()`, `current_timestamp` instead. (change required for prepared statements) (Tom)
• Treat NaN as larger than any other value in `min()`/`max()` (Tom)
  NaN was already sorted after ordinary numeric values for most purposes, but `min()` and `max()` didn’t get this right.
• Prevent interval from suppressing :00 seconds display
• New functions `pg_get_triggerdef(prettyprint)` and `pg_conversion_is_visible()` (Christopher)
• Allow time to be specified as `040506` or `0405` (Tom)
• Input date order must now be `YYYY-MM-DD` (with 4-digit year) or match `datestyle`
• Make `pg_get_constraintdef` support unique, primary-key, and check constraints (Christopher)

E.130.3.8. Server-Side Language Changes

• Prevent PL/pgSQL crash when `RETURN NEXT` is used on a zero-row record variable (Tom)
• Make PL/Python’s `spi_execute` interface handle null values properly (Andrew Bosma)
• Allow PL/pgSQL to declare variables of composite types without `%ROWTYPE` (Tom)
• Fix PL/Python’s `_quote()` function to handle big integers
• Make PL/Python an untrusted language, now called `plpythonu` (Kevin Jacobs, Tom)
  The Python language no longer supports a restricted execution environment, so the trusted version of PL/Python was removed. If this situation changes, a version of PL/Python that can be used by non-superusers will be readded.
• Allow polymorphic PL/pgSQL functions (Joe, Tom)
• Allow polymorphic SQL functions (Joe)
• Improved compiled function caching mechanism in PL/pgSQL with full support for polymorphism (Joe)
• Add new parameter `$0` in PL/pgSQL representing the function’s actual return type (Joe)
• Allow PL/Tcl and PL/Python to use the same trigger on multiple tables (Tom)
• Fixed PL/Tcl’s `spi_prepare` to accept fully qualified type names in the parameter type list (Jan)

E.130.3.9. `psql` Changes

• Add `\pset pager always` to always use pager (Greg)
  This forces the pager to be used even if the number of rows is less than the screen height. This is valuable for rows that wrap across several screen rows.
• Improve tab completion (Rod, Ross Reedstrom, Ian Barwick)
• Reorder `\?` help into groupings (Harald Armin Massa, Bruce)
• Add backslash commands for listing schemas, casts, and conversions (Christopher)
Appendix E. Release Notes

• \encoding now changes based on the server parameter client_encoding (Tom)
  In previous versions, \encoding was not aware of encoding changes made using SET client_encoding.
• Save editor buffer into readline history (Ross)
  When \e is used to edit a query, the result is saved in the readline history for retrieval using the up arrow.
• Improve \d display (Christopher)
• Enhance HTML mode to be more standards-conforming (Greg)
• New \set AUTOCOMMIT off capability (Tom)
  This takes the place of the removed server parameter autocommit.
• New \set VERBOSITY to control error detail (Tom)
  This controls the new error reporting details.
• New prompt escape sequence %x to show transaction status (Tom)
• Long options for psql are now available on all platforms

E.130.3.10. pg_dump Changes

• Multiple pg_dump fixes, including tar format and large objects
• Allow pg_dump to dump specific schemas (Neil)
• Make pg_dump preserve column storage characteristics (Christopher)
  This preserves ALTER TABLE ... SET STORAGE information.
• Make pg_dump preserve CLUSTER characteristics (Christopher)
• Have pg_dumpall use GRANT/REVOKE to dump database-level privileges (Tom)
• Allow pg_dumpall to support the options -a, -s, -x of pg_dump (Tom)
• Prevent pg_dump from lowercasing identifiers specified on the command line (Tom)
• pg_dump options --use-set-session-authorization and --no-reconnect now do nothing, all dumps use SET SESSION AUTHORIZATION
  pg_dump no longer reconnects to switch users, but instead always uses SET SESSION AUTHORIZATION. This will reduce password prompting during restores.
• Long options for pg_dump are now available on all platforms
  PostgreSQL now includes its own long-option processing routines.

E.130.3.11. libpq Changes

• Add function PQfreemem for freeing memory on Windows, suggested for NOTIFY (Bruce)
  Windows requires that memory allocated in a library be freed by a function in the same library,
  hence free() doesn’t work for freeing memory allocated by libpq. PQfreemem is the proper way
  to free libpq memory, especially on Windows, and is recommended for other platforms as well.
• Document service capability, and add sample file (Bruce)
This allows clients to look up connection information in a central file on the client machine.

- Make `PQsetdbLogin` have the same defaults as `PQconnectdb` (Tom)
- Allow `libpq` to cleanly fail when result sets are too large (Tom)
- Improve performance of function `PQunescapeBytea` (Ben Lamb)
- Allow thread-safe `libpq` with configure option `--enable-thread-safety` (Lee Kindness, Philip Yarra)
- Allow function `pqInternalNotice` to accept a format string and arguments instead of just a preformatted message (Tom, Sean Chittenden)
- Control SSL negotiation with `sslmode` values `disable`, `allow`, `prefer`, and `require` (Jon Jensen)
- Allow new error codes and levels of text (Tom)
- Allow access to the underlying table and column of a query result (Tom)
  This is helpful for query-builder applications that want to know the underlying table and column names associated with a specific result set.
- Allow access to the current transaction status (Tom)
- Add ability to pass binary data directly to the server (Tom)
- Add function `PQexecPrepared` and `PQsendQueryPrepared` functions which perform bind/execute of previously prepared statements (Tom)

### E.130.3.12. JDBC Changes

- Allow `setNull` on updateable result sets
- Allow `executeBatch` on a prepared statement (Barry)
- Support SSL connections (Barry)
- Handle schema names in result sets (Paul Sorenson)
- Add refcursor support (Nic Ferrier)

### E.130.3.13. Miscellaneous Interface Changes

- Prevent possible memory leak or core dump during libpqctl shutdown (Tom)
- Add Informix compatibility to ECPG (Michael)
  This allows ECPG to process embedded C programs that were written using certain Informix extensions.
- Add type `decimal` to ECPG that is fixed length, for Informix (Michael)
- Allow thread-safe embedded SQL programs with configure option `--enable-thread-safety` (Lee Kindness, Bruce)
  This allows multiple threads to access the database at the same time.
- Moved Python client PyGreSQL to http://www.pygresql.org (Marc)
E.130.3.14. Source Code Changes

- Prevent need for separate platform geometry regression result files (Tom)
- Improved PPC locking primitive (Reinhard Max)
- New function `palloc0` to allocate and clear memory (Bruce)
- Fix locking code for s390x CPU (64-bit) (Tom)
- Allow OpenBSD to use local ident credentials (William Ahern)
- Make query plan trees read-only to executor (Tom)
- Add Darwin startup scripts (David Wheeler)
- Allow libpq to compile with Borland C++ compiler (Lester Godwin, Karl Waclawek)
- Use our own version of `getopt_long()` if needed (Peter)
- Convert administration scripts to C (Peter)
- Bison >= 1.85 is now required to build the PostgreSQL grammar, if building from CVS
- Merge documentation into one book (Peter)
- Add Windows compatibility functions (Bruce)
- Allow client interfaces to compile under MinGW (Bruce)
- New `ereport()` function for error reporting (Tom)
- Support Intel compiler on Linux (Peter)
- Improve Linux startup scripts (Slawomir Sudnik, Darko Prenosil)
- Add support for AMD Opteron and Itanium (Jeffrey W. Baker, Bruce)
- Remove `--enable-recode` option from `configure`
  This was no longer needed now that we have `CREATE CONVERSION`.
- Generate a compile error if spinlock code is not found (Bruce)
  Platforms without spinlock code will now fail to compile, rather than silently using semaphores. This failure can be disabled with a new `configure` option.

E.130.3.15. Contrib Changes

- Change dbmirror license to BSD
- Improve earthdistance (Bruno Wolff III)
- Portability improvements to pgcrypto (Marko Kreen)
- Prevent crash in xml (John Gray, Michael Richards)
- Update oracle
- Update mysql
- Update cube (Bruno Wolff III)
- Update earthdistance to use cube (Bruno Wolff III)
- Update btree gist (Oleg)
- New tsearch2 full-text search module (Oleg, Teodor)
• Add hash-based crosstab function to tablefuncs (Joe)
• Add serial column to order connectby() siblings in tablefuncs (Nabil Sayegh, Joe)
• Add named persistent connections to dblink (Shridhar Daithanka)
• New pg_autovacuum allows automatic VACUUM (Matthew T. O’Connor)
• Make pgbench honor environment variables PGHOST, PGPORT, PGUSER (Tatsuo)
• Improve intarray (Teodor Sigaev)
• Improve pgstattuple (Rod)
• Fix bug in metaphone() in fuzzystarmatch
• Improve adddepend (Rod)
• Update spi/timetravel (Bőjthe Zoltán)
• Fix dbase -s option and improve non-ASCII handling (Thomas Behr, Márcio Smiderle)
• Remove array module because features now included by default (Joe)

E.131. Release 7.3.21

Release date: 2008-01-07

This release contains a variety of fixes from 7.3.20, including fixes for significant security issues. This is expected to be the last PostgreSQL release in the 7.3.X series. Users are encouraged to update to a newer release branch soon.

E.131.1. Migration to Version 7.3.21

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.131.2. Changes

• Prevent functions in indexes from executing with the privileges of the user running VACUUM, ANALYZE, etc (Tom)

Functions used in index expressions and partial-index predicates are evaluated whenever a new table entry is made. It has long been understood that this poses a risk of trojan-horse code execution if one modifies a table owned by an untrustworthy user. (Note that triggers, defaults, check constraints, etc. pose the same type of risk.) But functions in indexes pose extra danger because they will be executed by routine maintenance operations such as VACUUM FULL, which are commonly performed automatically under a superuser account. For example, a nefarious user can execute code with superuser privileges by setting up a trojan-horse index definition and waiting for the next routine vacuum. The fix arranges for standard maintenance operations (including VACUUM,
ANALYZE, REINDEX, and CLUSTER) to execute as the table owner rather than the calling user, using the same privilege-switching mechanism already used for SECURITY DEFINER functions. To prevent bypassing this security measure, execution of SET SESSION AUTHORIZATION and SET ROLE is now forbidden within a SECURITY DEFINER context. (CVE-2007-6600)

- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)

  The fix that appeared for this in 7.3.20 was incomplete, as it plugged the hole for only some dblink functions. (CVE-2007-6601, CVE-2007-3278)

- Fix potential crash in translate() when using a multibyte database encoding (Tom)

- Make contrib/tablefunc's crosstab() handle NULL rowid as a category in its own right, rather than crashing (Joe)

- Require a specific version of Autoconf to be used when re-generating the configure script (Peter)

  This affects developers and packagers only. The change was made to prevent accidental use of untested combinations of Autoconf and PostgreSQL versions. You can remove the version check if you really want to use a different Autoconf version, but it’s your responsibility whether the result works or not.

E.132. Release 7.3.20

Release date: 2007-09-17

This release contains fixes from 7.3.19.

E.132.1. Migration to Version 7.3.20

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.132.2. Changes

- Prevent index corruption when a transaction inserts rows and then aborts close to the end of a concurrent VACUUM on the same table (Tom)

- Make CREATE DOMAIN ... DEFAULT NULL work properly (Tom)

- Fix crash when log_min_error_statement logging runs out of memory (Tom)

- Require non-superusers who use /contrib/dblink to use only password authentication, as a security measure (Joe)
E.133. Release 7.3.19

Release date: 2007-04-23

This release contains fixes from 7.3.18, including a security fix.

E.133.1. Migration to Version 7.3.19

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.133.2. Changes

• Support explicit placement of the temporary-table schema within search_path, and disable searching it for functions and operators (Tom)

  This is needed to allow a security-definer function to set a truly secure value of search_path. Without it, an unprivileged SQL user can use temporary objects to execute code with the privileges of the security-definer function (CVE-2007-2138). See CREATE FUNCTION for more information.

• Fix potential-data-corruption bug in how VACUUM FULL handles UPDATE chains (Tom, Pavan Deolasee)

E.134. Release 7.3.18

Release date: 2007-02-05

This release contains a variety of fixes from 7.3.17, including a security fix.

E.134.1. Migration to Version 7.3.18

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.134.2. Changes

• Remove security vulnerability that allowed connected users to read backend memory (Tom)

  The vulnerability involves changing the data type of a table column used in a SQL function (CVE-2007-0555). This error can easily be exploited to cause a backend crash, and in principle might be used to read database content that the user should not be able to access.
• Fix rare bug wherein btree index page splits could fail due to choosing an infeasible split point (Heikki Linnakangas)
• Tighten security of multi-byte character processing for UTF8 sequences over three bytes long (Tom)

E.135. Release 7.3.17

Release date: 2007-01-08

This release contains a variety of fixes from 7.3.16.

E.135.1. Migration to Version 7.3.17

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.135.2. Changes

• to_number() and to_char(numeric) are now STABLE, not IMMUTABLE, for new initdb installs (Tom)
  This is because lc_numeric can potentially change the output of these functions.
• Improve index usage of regular expressions that use parentheses (Tom)
  This improves psql \d performance also.

E.136. Release 7.3.16

Release date: 2006-10-16

This release contains a variety of fixes from 7.3.15.

E.136.1. Migration to Version 7.3.16

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.
Appendix E. Release Notes

E.136.2. Changes

• Fix corner cases in pattern matching for psql’s \d commands
• Fix index-corrupting bugs in /contrib/ltrie (Teodor)
• Back-port 7.4 spinlock code to improve performance and support 64-bit architectures better
• Fix SSL-related memory leak in libpq
• Fix backslash escaping in /contrib/dbmirror
• Adjust regression tests for recent changes in US DST laws

E.137. Release 7.3.15

Release date: 2006-05-23

This release contains a variety of fixes from 7.3.14, including patches for extremely serious security issues.

E.137.1. Migration to Version 7.3.15

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

Full security against the SQL-injection attacks described in CVE-2006-2313 and CVE-2006-2314 might require changes in application code. If you have applications that embed untrustworthy strings into SQL commands, you should examine them as soon as possible to ensure that they are using recommended escaping techniques. In most cases, applications should be using subroutines provided by libraries or drivers (such as libpq’s PQescapeStringConn()) to perform string escaping, rather than relying on ad hoc code to do it.

E.137.2. Changes

• Change the server to reject invalidly-encoded multibyte characters in all cases (Tatsuo, Tom)
  While PostgreSQL has been moving in this direction for some time, the checks are now applied uniformly to all encodings and all textual input, and are now always errors not merely warnings. This change defends against SQL-injection attacks of the type described in CVE-2006-2313.
• Reject unsafe uses of \’ in string literals
  As a server-side defense against SQL-injection attacks of the type described in CVE-2006-2314, the server now only accepts ‘’ and not \’ as a representation of ASCII single quote in SQL string literals. By default, \’ is rejected only when client_encoding is set to a client-only encoding (SJIS, BIG5, GBK, GB18030, or UHC), which is the scenario in which SQL injection is possible. A new configuration parameter backslash_quote is available to adjust this behavior when needed. Note that full security against CVE-2006-2314 might require client-side changes; the purpose of backslash_quote is in part to make it obvious that insecure clients are insecure.
• Modify libpq’s string-escaping routines to be aware of encoding considerations
This fixes libpq-using applications for the security issues described in CVE-2006-2313 and CVE-2006-2314. Applications that use multiple PostgreSQL connections concurrently should migrate to PQescapeStringConn() and PQescapeByteaConn() to ensure that escaping is done correctly for the settings in use in each database connection. Applications that do string escaping “by hand” should be modified to rely on library routines instead.
• Fix some incorrect encoding conversion functions
  win1251_to_iso, alt_to_iso, euc_tw_to_big5, euc_tw_to_mic, mic_to_euc_tw were all broken to varying extents.
• Clean up stray remaining uses of ‘\’ in strings (Bruce, Jan)
• Fix server to use custom DH SSL parameters correctly (Michael Fuhr)
• Fix various minor memory leaks

E.138. Release 7.3.14

Release date: 2006-02-14

This release contains a variety of fixes from 7.3.13.

E.138.1. Migration to Version 7.3.14

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.13, see the release notes for 7.3.13.

E.138.2. Changes

• Fix potential crash in SET SESSION AUTHORIZATION (CVE-2006-0553)
  An unprivileged user could crash the server process, resulting in momentary denial of service to other users, if the server has been compiled with Asserts enabled (which is not the default). Thanks to Akio Ishida for reporting this problem.
• Fix bug with row visibility logic in self-inserted rows (Tom)
  Under rare circumstances a row inserted by the current command could be seen as already valid, when it should not be. Repairs bug created in 7.3.11 release.
• Fix race condition that could lead to “file already exists” errors during pg_clog file creation (Tom)
• Fix to allow restoring dumps that have cross-schema references to custom operators (Tom)
• Portability fix for testing presence of finite and isinf during configure (Tom)
E.139. Release 7.3.13

Release date: 2006-01-09

This release contains a variety of fixes from 7.3.12.

E.139.1. Migration to Version 7.3.13

A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see the release notes for 7.3.10. Also, you might need to REINDEX indexes on textual columns after updating, if you are affected by the locale or plperl issues described below.

E.139.2. Changes

• Fix character string comparison for locales that consider different character combinations as equal, such as Hungarian (Tom)
  This might require REINDEX to fix existing indexes on textual columns.
• Set locale environment variables during postmaster startup to ensure that plperl won’t change the locale later
  This fixes a problem that occurred if the postmaster was started with environment variables specifying a different locale than what initdb had been told. Under these conditions, any use of plperl was likely to lead to corrupt indexes. You might need REINDEX to fix existing indexes on textual columns if this has happened to you.
• Fix longstanding bug in strpos() and regular expression handling in certain rarely used Asian multi-byte character sets (Tatsuo)
• Fix bug in /contrib/pgcrypto gen_salt, which caused it not to use all available salt space for MD5 and XDES algorithms (Marko Kreen, Solar Designer)
  Salts for Blowfish and standard DES are unaffected.
• Fix /contrib/dblink to throw an error, rather than crashing, when the number of columns specified is different from what’s actually returned by the query (Joe)

E.140. Release 7.3.12

Release date: 2005-12-12

This release contains a variety of fixes from 7.3.11.
Appendix E. Release Notes

E.140.1. Migration to Version 7.3.12
A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see the release notes for 7.3.10.

E.140.2. Changes

- Fix race condition in transaction log management
  There was a narrow window in which an I/O operation could be initiated for the wrong page, leading to an Assert failure or data corruption.
- /contrib/ltree fixes (Teodor)
- Fix longstanding planning error for outer joins
  This bug sometimes caused a bogus error “RIGHT JOIN is only supported with merge-joinable join conditions”.
- Prevent core dump in pg_autovacuum when a table has been dropped

E.141. Release 7.3.11

Release date: 2005-10-04

This release contains a variety of fixes from 7.3.10.

E.141.1. Migration to Version 7.3.11
A dump/restore is not required for those running 7.3.X. However, if you are upgrading from a version earlier than 7.3.10, see the release notes for 7.3.10.

E.141.2. Changes

- Fix error that allowed VACUUM to remove ctid chains too soon, and add more checking in code that follows ctid links
  This fixes a long-standing problem that could cause crashes in very rare circumstances.
- Fix CHAR() to properly pad spaces to the specified length when using a multiple-byte character set (Yoshiyuki Asaba)
  In prior releases, the padding of CHAR() was incorrect because it only padded to the specified number of bytes without considering how many characters were stored.
- Fix missing rows in queries like UPDATE a=... WHERE a... with GiST index on column a
- Improve checking for partially-written WAL pages
- Improve robustness of signal handling when SSL is enabled
• Various memory leakage fixes
• Various portability improvements
• Fix PL/PgSQL to handle `var := var` correctly when the variable is of pass-by-reference type

E.142. Release 7.3.10

Release date: 2005-05-09

This release contains a variety of fixes from 7.3.9, including several security-related issues.

E.142.1. Migration to Version 7.3.10

A dump/restore is not required for those running 7.3.X. However, it is one possible way of handling a significant security problem that has been found in the initial contents of 7.3.X system catalogs. A dump/initdb/reload sequence using 7.3.10's initdb will automatically correct this problem.

The security problem is that the built-in character set encoding conversion functions can be invoked from SQL commands by unprivileged users, but the functions were not designed for such use and are not secure against malicious choices of arguments. The fix involves changing the declared parameter list of these functions so that they can no longer be invoked from SQL commands. (This does not affect their normal use by the encoding conversion machinery.) It is strongly recommended that all installations repair this error, either by initdb or by following the manual repair procedure given below. The error at least allows unprivileged database users to crash their server process, and might allow unprivileged users to gain the privileges of a database superuser.

If you wish not to do an initdb, perform the following procedure instead. As the database superuser, do:

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
    AND proargtypes[2] = 'cstring'::regtype;
-- The command should report having updated 90 rows;
-- if not, rollback and investigate instead of committing!
COMMIT;
```

The above procedure must be carried out in each database of an installation, including template1, and ideally including template0 as well. If you do not fix the template databases then any subsequently created databases will contain the same error. template1 can be fixed in the same way as any other database, but fixing template0 requires additional steps. First, from any database issue:

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

Next connect to template0 and perform the above repair procedure. Finally, do:

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
```
Appendix E. Release Notes

UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';

E.142.2. Changes

• Change encoding function signature to prevent misuse

• Repair ancient race condition that allowed a transaction to be seen as committed for some purposes
  (eg SELECT FOR UPDATE) slightly sooner than for other purposes
  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.

• Repair race condition between relation extension and VACUUM
  This could theoretically have caused loss of a page’s worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.

• Fix comparisons of TIME WITH TIME ZONE values
  The comparison code was wrong in the case where the --enable-integer-datetimes configuration switch had been used. NOTE: if you have an index on a TIME WITH TIME ZONE column, it will need to be REINDEXed after installing this update, because the fix corrects the sort order of column values.

• Fix EXTRACT(EPOCH) for TIME WITH TIME ZONE values

• Fix mis-display of negative fractional seconds in INTERVAL values
  This error only occurred when the --enable-integer-datetimes configuration switch had been used.

• Additional buffer overrun checks in plpgsql (Neil)

• Fix pg_dump to dump trigger names containing % correctly (Neil)

• Prevent to_char(interval) from dumping core for month-related formats

• Fix contrib/pgcrypto for newer OpenSSL builds (Marko Kreen)

• Still more 64-bit fixes for contrib/intagg

• Prevent incorrect optimization of functions returning RECORD

E.143. Release 7.3.9

Release date: 2005-01-31

This release contains a variety of fixes from 7.3.8, including several security-related issues.
E.143.1. Migration to Version 7.3.9

A dump/restore is not required for those running 7.3.X.

E.143.2. Changes

- **Disallow LOAD to non-superusers**
  
  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), LOAD can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.

- **Check that creator of an aggregate function has the right to execute the specified transition functions**
  
  This oversight made it possible to bypass denial of EXECUTE permission on a function.

- **Fix security and 64-bit issues in contrib/intagg**

- **Add needed STRICT marking to some contrib functions (Kris Jurka)**

- **Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)**

- **Fix planning error for FULL and RIGHT outer joins**
  
  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- **Fix plperl for quote marks in tuple fields**

- **Fix display of negative intervals in SQL and GERMAN datestyles**

E.144. Release 7.3.8

**Release date:** 2004-10-22

This release contains a variety of fixes from 7.3.7.

E.144.1. Migration to Version 7.3.8

A dump/restore is not required for those running 7.3.X.

E.144.2. Changes

- **Repair possible failure to update hint bits on disk**
  
  Under rare circumstances this oversight could lead to “could not access transaction status” failures, which qualifies it as a potential-data-loss bug.

- **Ensure that hashed outer join does not miss tuples**
Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.

- Disallow running `pg_ctl` as root
  This is to guard against any possible security issues.
- Avoid using temp files in `/tmp` in `make_oidjoins_check`
  This has been reported as a security issue, though it’s hardly worthy of concern since there is no reason for non-developers to use this script anyway.

### E.145. Release 7.3.7

**Release date:** 2004-08-16

This release contains one critical fix over 7.3.6, and some minor items.

#### E.145.1. Migration to Version 7.3.7

A dump/restore is not required for those running 7.3.X.

#### E.145.2. Changes

- Prevent possible loss of committed transactions during crash
  Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.
- Remove asymmetrical word processing in tsearch (Teodor)
- Properly schema-qualify function names when `pg_dump`'ing a CAST

### E.146. Release 7.3.6

**Release date:** 2004-03-02

This release contains a variety of fixes from 7.3.5.

#### E.146.1. Migration to Version 7.3.6

A dump/restore is *not* required for those running 7.3.*.
E.146.2. Changes

- Revert erroneous changes in rule permissions checking
  A patch applied in 7.3.3 to fix a corner case in rule permissions checks turns out to have disabled rule-related permissions checks in many not-so-corner cases. This would for example allow users to insert into views they weren’t supposed to have permission to insert into. We have therefore reverted the 7.3.3 patch. The original bug will be fixed in 8.0.
- Repair incorrect order of operations in GetNewTransactionId()
  This bug could result in failure under out-of-disk-space conditions, including inability to restart even after disk space is freed.
- Ensure configure selects -fno-strict-aliasing even when an external value for CFLAGS is supplied
  On some platforms, building with -fstrict-aliasing causes bugs.
- Make pg_restore handle 64-bit off_t correctly
  This bug prevented proper restoration from archive files exceeding 4 GB.
- Make contrib/dblink not assume that local and remote type OIDs match (Joe)
- Quote connectby()'s start_with argument properly (Joe)
- Don’t crash when a rowtype argument to a plpgsql function is NULL
- Avoid generating invalid character encoding sequences in corner cases when planning LIKE operations
- Ensure text_position() cannot scan past end of source string in multibyte cases (Korea PostgreSQL Users’ Group)
- Fix index optimization and selectivity estimates for LIKE operations on bytea columns (Joe)

E.147. Release 7.3.5

Release date: 2003-12-03

This has a variety of fixes from 7.3.4.

E.147.1. Migration to Version 7.3.5

A dump/restore is not required for those running 7.3.*.

E.147.2. Changes

- Force zero_damaged_pages to be on during recovery from WAL
- Prevent some obscure cases of “variable not in subplan target lists”
- Force stats processes to detach from shared memory, ensuring cleaner shutdown
• Make PQescapeBytea and byteaout consistent with each other (Joe)
• Added missing SPI_finish() calls to dblink’s get_tuple_of_interest() (Joe)
• Fix for possible foreign key violation when rule rewrites INSERT (Jan)
• Support qualified type names in PL/Tcl’s spi_prepare command (Jan)
• Make pg_dump handle a procedural language handler located in pg_catalog
• Make pg_dump handle cases where a custom opclass is in another schema
• Make pg_dump dump binary-compatible casts correctly (Jan)
• Fix insertion of expressions containing subqueries into rule bodies
• Fix incorrect argument processing in clusterdb script (Anand Ranganathan)
• Fix problems with dropped columns in plpython triggers
• Repair problems with to_char() reading past end of its input string (Karel)
• Fix GB18030 mapping errors (Tatsuo)
• Fix several problems with SSL error handling and asynchronous SSL I/O
• Remove ability to bind a list of values to a single parameter in JDBC (prevents possible SQL-injection attacks)
• Fix some errors in HAVE_INT64_TIMESTAMP code paths
• Fix corner case for btree search in parallel with first root page split

E.148. Release 7.3.4

Release date: 2003-07-24

This has a variety of fixes from 7.3.3.

E.148.1. Migration to Version 7.3.4

A dump/restore is not required for those running 7.3.*.

E.148.2. Changes

• Repair breakage in timestamp-to-date conversion for dates before 2000
• Prevent rare possibility of server startup failure (Tom)
• Fix bugs in interval-to-time conversion (Tom)
• Add constraint names in a few places in pg_dump (Rod)
• Improve performance of functions with many parameters (Tom)
• Fix to_ascii() buffer overruns (Tom)
• Prevent restore of database comments from throwing an error (Tom)
• Work around buggy strxfrm() present in some Solaris releases (Tom)
• Properly escape jdbc setObject() strings to improve security (Barry)

E.149. Release 7.3.3

Release date: 2003-05-22

This release contains a variety of fixes for version 7.3.2.

E.149.1. Migration to Version 7.3.3
A dump/restore is not required for those running version 7.3.*.

E.149.2. Changes

• Repair sometimes-incorrect computation of StartUpID after a crash
• Avoid slowness with lots of deferred triggers in one transaction (Stephan)
• Don’t lock referenced row when UPDATE doesn’t change foreign key’s value (Jan)
• Use -fPIC not -fpic on Sparc (Tom Callaway)
• Repair lack of schema-awareness in contrib/reindexdb
• Fix contrib/intarray error for zero-element result array (Teodor)
• Ensure createuser script will exit on control-C (Oliver)
• Fix errors when the type of a dropped column has itself been dropped
• CHECKPOINT does not cause database panic on failure in noncritical steps
• Accept 60 in seconds fields of timestamp, time, interval input values
• Issue notice, not error, if TIMESTAMP, TIME, or INTERVAL precision too large
• Fix abstime-to-time cast function (fix is not applied unless you initdb)
• Fix pg_proc entry for timestamptz (fix is not applied unless you initdb)
• Make EXTRACT (EPOCH FROM timestamp without time zone) treat input as local time
• ’now’::timestamptz gave wrong answer if timezone changed earlier in transaction
• HAVE_INT64_TIMESTAMP code for time with timezone overwrote its input
• Accept GLOBAL TEMP/TEMPORARY as a synonym for TEMPORARY
• Avoid improper schema-privilege-check failure in foreign-key triggers
• Fix bugs in foreign-key triggers for SET DEFAULT action
• Fix incorrect time-qual check in row fetch for UPDATE and DELETE triggers
• Foreign-key clauses were parsed but ignored in ALTER TABLE ADD COLUMN
Appendix E. Release Notes

• Fix createlang script breakage for case where handler function already exists
• Fix misbehavior on zero-column tables in pg_dump, COPY, ANALYZE, other places
• Fix misbehavior of func_error() on type names containing ‘%’
• Fix misbehavior of replace() on strings containing ‘%’
• Regular-expression patterns containing certain multibyte characters failed
• Account correctly for NULLs in more cases in join size estimation
• Avoid conflict with system definition of isblank() function or macro
• Fix failure to convert large code point values in EUC_TW conversions (Tatsuo)
• Fix error recovery for SSL_read/SSL_write calls
• Don’t do early constant-folding of type coercion expressions
• Validate page header fields immediately after reading in any page
• Repair incorrect check for ungrouped variables in unnamed joins
• Fix buffer overrun in to_ascii (Guido Notari)
• contrib/ltree fixes (Teodor)
• Fix core dump in deadlock detection on machines where char is unsigned
• Avoid running out of buffers in many-way indexscan (bug introduced in 7.3)
• Fix planner’s selectivity estimation functions to handle domains properly
• Fix dbmirror memory-allocation bug (Steven Singer)
• Prevent infinite loop in ln(numeric) due to roundoff error
  GROUP BY got confused if there were multiple equal GROUP BY items
• Fix bad plan when inherited UPDATE/DELETE references another inherited table
• Prevent clustering on incomplete (partial or non-NUL-storing) indexes
• Service shutdown request at proper time if it arrives while still starting up
• Fix left-links in temporary indexes (could make backwards scans miss entries)
• Fix incorrect handling of client_encoding setting in postgresql.conf (Tatsuo)
• Fix failure to respond to pg_ctl stop -m fast after Async_NotifyHandler runs
• Fix SPI for case where rule contains multiple statements of the same type
• Fix problem with checking for wrong type of access privilege in rule query
• Fix problem with EXCEPT in CREATE RULE
• Prevent problem with dropping temp tables having serial columns
• Fix replace_vars_with_subplan.refs failure in complex views
• Fix regexp slowness in single-byte encodings (Tatsuo)
• Allow qualified type names in CREATE CAST and DROP CAST
• Accept SE TOF type[], which formerly had to be written SETOF _type
• Fix pg_dump core dump in some cases with procedural languages
• Force ISO datestyle in pg_dump output, for portability (Oliver)
• pg_dump failed to handle error return from lo_read (Oleg Drokin)
Appendix E. Release Notes

- pg_dumpall failed with groups having no members (Nick Eskelinen)
- pg_dumpall failed to recognize --globals-only switch
- pg_restore failed to restore blobs if -X disable-triggers is specified
- Repair intrafunction memory leak in plpgsql
- pltcl’s elog command dumped core if given wrong parameters (Ian Harding)
- plpython used wrong value of atttypmod (Brad McLean)
- Fix improper quoting of boolean values in Python interface (D’Arcy)
- Added addDataType() method to PGConnection interface for JDBC
- Fixed various problems with updateable ResultSets for JDBC (Shawn Green)
- Fixed various problems with DatabaseMetaData for JDBC (Kris Jurka, Peter Royal)
- Fixed problem with parsing table ACLs in JDBC
- Better error message for character set conversion problems in JDBC

E.150. Release 7.3.2

Release date: 2003-02-04

This release contains a variety of fixes for version 7.3.1.

E.150.1. Migration to Version 7.3.2

A dump/restore is not required for those running version 7.3.*.

E.150.2. Changes

- Restore creation of OID column in CREATE TABLE AS / SELECT INTO
- Fix pg_dump core dump when dumping views having comments
- Dump DEFERRABLE/INITIALLY DEFERRED constraints properly
- Fix UPDATE when child table’s column numbering differs from parent
- Increase default value of max_fsm_relations
- Fix problem when fetching backwards in a cursor for a single-row query
- Make backward fetch work properly with cursor on SELECT DISTINCT query
- Fix problems with loading pg_dump files containing contrib/lo usage
- Fix problem with all-numeric user names
- Fix possible memory leak and core dump during disconnect in libpq
- Make plpython’s spi_execute command handle nulls properly (Andrew Bosma)
Appendix E. Release Notes

- Adjust plpython error reporting so that its regression test passes again
- Work with bison 1.875
- Handle mixed-case names properly in plpgsql’s %type (Neil)
- Fix core dump in pltcl when executing a query rewritten by a rule
- Repair array subscript overruns (per report from Yichen Xie)
- Reduce MAX_TIME_PRECISION from 13 to 10 in floating-point case
- Correctly case-fold variable names in per-database and per-user settings
- Fix coredump in plpgsql’s RETURN NEXT when SELECT into record returns no rows
- Fix outdated use of pg_type.typprtlen in python client interface
- Correctly handle fractional seconds in timestamps in JDBC driver
- Improve performance of getImportedKeys() in JDBC
- Make shared-library symlinks work standardly on HPUX (Giles)
- Repair inconsistent rounding behavior for timestamp, time, interval
- SSL negotiation fixes (Nathan Mueller)
- Make libpq’s ~/.pgpass feature work when connecting with PQconnectDB
- Update my2pg, ora2pg
- Translation updates
- Add casts between types lo and oid in contrib/lo
- fastpath code now checks for privilege to call function

E.151. Release 7.3.1

Release date: 2002-12-18

This release contains a variety of fixes for version 7.3.

E.151.1. Migration to Version 7.3.1

A dump/restore is not required for those running version 7.3. However, it should be noted that the main PostgreSQL interface library, libpq, has a new major version number for this release, which might require recompilation of client code in certain cases.

E.151.2. Changes

- Fix a core dump of COPY TO when client/server encodings don’t match (Tom)
- Allow pg_dump to work with pre-7.2 servers (Philip)
- contrib/adddepend fixes (Tom)
Appendix E. Release Notes

- Fix problem with deletion of per-user/per-database config settings (Tom)
- contrib/vacuumlo fix (Tom)
- Allow 'password' encryption even when pg_shadow contains MD5 passwords (Bruce)
- contrib/dbmirror fix (Steven Singer)
- Optimizer fixes (Tom)
- contrib/tsearch fixes (Teodor Sigaev, Magnus)
- Allow locale names to be mixed case (Nicolai Tufar)
- Increment libpq library’s major version number (Bruce)
- pg_hba.conf error reporting fixes (Bruce, Neil)
- Add SCO Openserver 5.0.4 as a supported platform (Bruce)
- Prevent EXPLAIN from crashing server (Tom)
- SSL fixes (Nathan Mueller)
- Prevent composite column creation via ALTER TABLE (Tom)

E.152. Release 7.3

Release date: 2002-11-27

E.152.1. Overview

Major changes in this release:

Schemas

Schemas allow users to create objects in separate namespaces, so two people or applications can have tables with the same name. There is also a public schema for shared tables. Table/index creation can be restricted by removing privileges on the public schema.

Drop Column

PostgreSQL now supports the \texttt{ALTER TABLE \ldots DROP COLUMN} functionality.

Table Functions

Functions returning multiple rows and/or multiple columns are now much easier to use than before. You can call such a “table function” in the \texttt{SELECT FROM} clause, treating its output like a table. Also, PL/pgSQL functions can now return sets.

Prepared Queries

PostgreSQL now supports prepared queries, for improved performance.

Dependency Tracking

PostgreSQL now records object dependencies, which allows improvements in many areas. \texttt{DROP} statements now take either \texttt{CASCADE} or \texttt{RESTRICT} to control whether dependent objects are also dropped.
Appendix E. Release Notes

Privileges

Functions and procedural languages now have privileges, and functions can be defined to run with the privileges of their creator.

Internationalization

Both multibyte and locale support are now always enabled.

Logging

A variety of logging options have been enhanced.

Interfaces

A large number of interfaces have been moved to http://gborg.postgresql.org where they can be developed and released independently.

Functions/Identifiers

By default, functions can now take up to 32 parameters, and identifiers can be up to 63 bytes long. Also, OPAQUE is now deprecated: there are specific “pseudo-datatypes” to represent each of the former meanings of OPAQUE in function argument and result types.

E.152.2. Migration to Version 7.3

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release. If your application examines the system catalogs, additional changes will be required due to the introduction of schemas in 7.3; for more information, see: http://developer.postgresql.org/~momjian/upgrade_tips_7.3.

Observe the following incompatibilities:

- Pre-6.3 clients are no longer supported.
- pg_hba.conf now has a column for the user name and additional features. Existing files need to be adjusted.
- Several postgresql.conf logging parameters have been renamed.
- LIMIT #,# has been disabled; use LIMIT # OFFSET #.
- INSERT statements with column lists must specify a value for each specified column. For example, INSERT INTO tab (coll, col2) VALUES (‘val1’) is now invalid. It’s still allowed to supply fewer columns than expected if the INSERT does not have a column list.
- serial columns are no longer automatically UNIQUE; thus, an index will not automatically be created.
- A SET command inside an aborted transaction is now rolled back.
- COPY no longer considers missing trailing columns to be null. All columns need to be specified. (However, one can achieve a similar effect by specifying a column list in the COPY command.)
- The data type timestamp is now equivalent to timestamp without time zone, instead of timestamp with time zone.
- Pre-7.3 databases loaded into 7.3 will not have the new object dependencies for serial columns, unique constraints, and foreign keys. See the directory contrib/adddepend/ for a detailed description and a script that will add such dependencies.
- An empty string ("") is no longer allowed as the input into an integer field. Formerly, it was silently interpreted as 0.
E.152.3. Changes

E.152.3.1. Server Operation

- Add pg_locks view to show locks (Neil)
- Security fixes for password negotiation memory allocation (Neil)
- Remove support for version 0 FE/BE protocol (PostgreSQL 6.2 and earlier) (Tom)
- Reserve the last few backend slots for superusers, add parameter superuser_reserved_connections to control this (Nigel J. Andrews)

E.152.3.2. Performance

- Improve startup by calling localtime() only once (Tom)
- Cache system catalog information in flat files for faster startup (Tom)
- Improve caching of index information (Tom)
- Optimizer improvements (Tom, Fernando Nasser)
- Catalog caches now store failed lookups (Tom)
- Hash function improvements (Neil)
- Improve performance of query tokenization and network handling (Peter)
- Speed improvement for large object restore (Mario Weilguni)
- Mark expired index entries on first lookup, saving later heap fetches (Tom)
- Avoid excessive NULL bitmap padding (Manfred Koizar)
- Add BSD-licensed qsort() for Solaris, for performance (Bruce)
- Reduce per-row overhead by four bytes (Manfred Koizar)
- Fix GEOQ optimizer bug (Neil Conway)
- Make WITHOUT OID actually save four bytes per row (Manfred Koizar)
- Add default_statistics_target variable to specify ANALYZE buckets (Neil)
- Use local buffer cache for temporary tables so no WAL overhead (Tom)
- Improve free space map performance on large tables (Stephen Marshall, Tom)
- Improved WAL write concurrency (Tom)

E.152.3.3. Privileges

- Add privileges on functions and procedural languages (Peter)
- Add OWNER to CREATE DATABASE so superusers can create databases on behalf of unprivileged users (Gavin Sherry, Tom)
- Add new object privilege bits EXECUTE and USAGE (Tom)
- Add SET SESSION AUTHORIZATION DEFAULT and RESET SESSION AUTHORIZATION (Tom)
Appendix E. Release Notes

• Allow functions to be executed with the privilege of the function owner (Peter)

E.152.3.4. Server Configuration

• Server log messages now tagged with LOG, not DEBUG (Bruce)
• Add user column to pg_hba.conf (Bruce)
• Have log_connections output two lines in log file (Tom)
• Remove debug_level from postgresql.conf, now server_min_messages (Bruce)
• New ALTER DATABASE/USER ... SET command for per-user/database initialization (Peter)
• New parameters server_min_messages and client_min_messages to control which messages are sent to the server logs or client applications (Bruce)
• Allow pg_hba.conf to specify lists of users/databases separated by commas, group names prepended with +, and file names prepended with @ (Bruce)
• Remove secondary password file capability and pg_password utility (Bruce)
• Add variable db_user_namespace for database-local user names (Bruce)
• SSL improvements (Bear Giles)
• Make encryption of stored passwords the default (Bruce)
• Allow pg_statistics to be reset by calling pg_stat_reset() (Christopher)
• Add log_duration parameter (Bruce)
• Rename debug_print_query to log_statement (Bruce)
• Rename show_query_stats to show_statement_stats (Bruce)
• Add param log_min_error_statement to print commands to logs on error (Gavin)

E.152.3.5. Queries

• Make cursors insensitive, meaning their contents do not change (Tom)
• Disable LIMIT #,# syntax; now only LIMIT # OFFSET # supported (Bruce)
• Increase identifier length to 63 (Neil, Bruce)
• UNION fixes for merging >= 3 columns of different lengths (Tom)
• Add DEFAULT key word to INSERT, e.g., INSERT ... (...) DEFAULT, ...) (Rod)
• Allow views to have default values using ALTER COLUMN ... SET DEFAULT (Neil)
• Fail on INSERTs with column lists that don’t supply all column values, e.g., INSERT INTO tab (col1, col2) VALUES ('val1'); (Rod)
• Fix for join aliases (Tom)
• Fix for FULL OUTER JOINs (Tom)
• Improve reporting of invalid identifier and location (Tom, Gavin)
• Fix OPEN cursor(args) (Tom)
• Allow ‘ctid’ to be used in a view and currtid(viewname) (Hiroshi)
Appendix E. Release Notes

- Fix for CREATE TABLE AS with UNION (Tom)
- SQL99 syntax improvements (Thomas)
- Add statement_timeout variable to cancel queries (Bruce)
- Allow prepared queries with PREPARE/EXECUTE (Neil)
- Allow FOR UPDATE to appear after LIMIT/OFFSET (Bruce)
- Add variable autocommit (Tom, David Van Wie)

E.152.3.6. Object Manipulation

- Make equals signs optional in CREATE DATABASE (Gavin Sherry)
- Make ALTER TABLE OWNER change index ownership too (Neil)
- New ALTER TABLE tabname ALTER COLUMN colname SET STORAGE controls TOAST storage, compression (John Gray)
- Add schema support, CREATE/DROP SCHEMA (Tom)
- Create schema for temporary tables (Tom)
- Add variable search_path for schema search (Tom)
- Add ALTER TABLE SET/DROP NOT NULL (Christopher)
- New CREATE FUNCTION volatility levels (Tom)
- Make rule names unique only per table (Tom)
- Add 'ON tabname' clause to DROP RULE and COMMENT ON RULE (Tom)
- Add ALTER TRIGGER RENAME (Joe)
- New current_schema() and current_schemas() inquiry functions (Tom)
- Allow functions to return multiple rows (table functions) (Joe)
- Make WITH optional in CREATE DATABASE, for consistency (Bruce)
- Add object dependency tracking (Rod, Tom)
- Add RESTRICT/CASCADE to DROP commands (Rod)
- Add ALTER TABLE DROP for non-CHECK CONSTRAINT (Rod)
- Autodestroy sequence on DROP of table with SERIAL (Rod)
- Prevent column dropping if column is used by foreign key (Rod)
- Automatically drop constraints/functions when object is dropped (Rod)
- Add CREATE/DROP OPERATOR CLASS (Bill Studenmund, Tom)
- Add ALTER TABLE DROP COLUMN (Christopher, Tom, Hiroshi)
- Prevent inherited columns from being removed or renamed (Alvaro Herrera)
- Fix foreign key constraints to not error on intermediate database states (Stephan)
- Propagate column or table renaming to foreign key constraints
- Add CREATE OR REPLACE VIEW (Gavin, Neil, Tom)
- Add CREATE OR REPLACE RULE (Gavin, Neil, Tom)
- Have rules execute alphabetically, returning more predictable values (Tom)
Appendix E. Release Notes

- Triggers are now fired in alphabetical order (Tom)
- Add /contrib/adddepend to handle pre-7.3 object dependencies (Rod)
- Allow better casting when inserting/updating values (Tom)

E.152.3.7. Utility Commands

- Have COPY TO output embedded carriage returns and newlines as \r and \n (Tom)
- Allow DELIMITER in COPY FROM to be 8-bit clean (Tatsuo)
- Make pg_dump use ALTER TABLE ADD PRIMARY KEY, for performance (Neil)
- Disable brackets in multistatement rules (Bruce)
- Disable VACUUM from being called inside a function (Bruce)
- Allow dropdb and other scripts to use identifiers with spaces (Bruce)
- Restrict database comment changes to the current database
- Allow comments on operators, independent of the underlying function (Rod)
- Rollback SET commands in aborted transactions (Tom)
- EXPLAIN now outputs as a query (Tom)
- Display condition expressions and sort keys in EXPLAIN (Tom)
- Add ‘SET LOCAL var = value’ to set configuration variables for a single transaction (Tom)
- Allow ANALYZE to run in a transaction (Bruce)
- Improve COPY syntax using new WITH clauses, keep backward compatibility (Bruce)
- Fix pg_dump to consistently output tags in non-ASCII dumps (Bruce)
- Make foreign key constraints clearer in dump file (Rod)
- Add COMMENT ON CONSTRAINT (Rod)
- Allow COPY TO/FROM to specify column names (Brent Verner)
- Dump UNIQUE and PRIMARY KEY constraints as ALTER TABLE (Rod)
- Have SHOW output a query result (Joe)
- Generate failure on short COPY lines rather than pad NULLs (Neil)
- Fix CLUSTER to preserve all table attributes (Alvaro Herrera)
- New pg_settings table to view/modify GUC settings (Joe)
- Add smart quoting, portability improvements to pg_dump output (Peter)
- Dump serial columns out as SERIAL (Tom)
- Enable large file support, >2G for pg_dump (Peter, Philip Warner, Bruce)
- Disallow TRUNCATE on tables that are involved in referential constraints (Rod)
- Have TRUNCATE also auto-truncate the toast table of the relation (Tom)
- Add clusterdb utility that will auto-cluster an entire database based on previous CLUSTER operations (Alvaro Herrera)
- Overhaul pg_dumpall (Peter)
- Allow REINDEX of TOAST tables (Tom)
Appendix E. Release Notes

• Implemented START TRANSACTION, per SQL99 (Neil)
• Fix rare index corruption when a page split affects bulk delete (Tom)
• Fix ALTER TABLE ... ADD COLUMN for inheritance (Alvaro Herrera)

E.152.3.8. Data Types and Functions

• Fix factorial(0) to return 1 (Bruce)
• Date/time/timezone improvements (Thomas)
• Fix for array slice extraction (Tom)
• Fix extract/date_part to report proper microseconds for timestamp (Tatsuo)
• Allow text_substr() and bytea_substr() to read TOAST values more efficiently (John Gray)
• Add domain support (Rod)
• Make WITHOUT TIME ZONE the default for TIMESTAMP and TIME data types (Thomas)
• Allow alternate storage scheme of 64-bit integers for date/time types using
  --enable-integer-datetimes in configure (Thomas)
• Make timezone(timestamptz) return timestamp rather than a string (Thomas)
• Allow fractional seconds in date/time types for dates prior to 1BC (Thomas)
• Limit timestamp data types to 6 decimal places of precision (Thomas)
• Change timezone conversion functions from timetz() to timezone() (Thomas)
• Add configuration variables datestyle and timezone (Tom)
• Add OVERLAY(), which allows substitution of a substring in a string (Thomas)
• Add SIMILAR TO (Thomas, Tom)
• Add regular expression SUBSTRING(string FROM pat FOR escape) (Thomas)
• Add LOCALTIME and LOCALTIMESTAMP functions (Thomas)
• Add named composite types using CREATE TYPE typename AS (column) (Joe)
• Allow composite type definition in the table alias clause (Joe)
• Add new API to simplify creation of C language table functions (Joe)
• Remove ODBC-compatible empty parentheses from calls to SQL99 functions for which these
  parentheses do not match the standard (Thomas)
• Allow macaddr data type to accept 12 hex digits with no separators (Mike Wyer)
• Add CREATE/DROP CAST (Peter)
• Add IS DISTINCT FROM operator (Thomas)
• Add SQL99 TREAT() function, synonym for CAST() (Thomas)
• Add pg_backend_pid() to output backend pid (Bruce)
• Add IS OF / IS NOT OF type predicate (Thomas)
• Allow bit string constants without fully-specified length (Thomas)
• Allow conversion between 8-byte integers and bit strings (Thomas)
• Implement hex literal conversion to bit string literal (Thomas)
Appendix E. Release Notes

- Allow table functions to appear in the FROM clause (Joe)
- Increase maximum number of function parameters to 32 (Bruce)
- No longer automatically create index for SERIAL column (Tom)
- Add current_database() (Rod)
- Fix cash_words() to not overflow buffer (Tom)
- Add functions replace(), split_part(), to_hex() (Joe)
- Fix LIKE for bytea as a right-hand argument (Joe)
- Prevent crashes caused by SELECT cash_out(2) (Tom)
- Fix to_char(1,'FM999.99') to return a period (Karel)
- Fix trigger/type/language functions returning OPAQUE to return proper type (Tom)

E.152.3.9. Internationalization

- Add additional encodings: Korean (JOHAB), Thai (WIN874), Vietnamese (TCVN), Arabic (WIN1256), Simplified Chinese (GBK), Korean (UHC) (Eiji Tokuya)
- Enable locale support by default (Peter)
- Add locale variables (Peter)
- Escape bytes >= 0x7f for multibyte in PQescapeBytea/PQunescapeBytea (Tatsuo)
- Add locale awareness to regular expression character classes
- Enable multibyte support by default (Tatsuo)
- Add GB18030 multibyte support (Bill Huang)
- Add CREATE/DROP CONVERSION, allowing loadable encodings (Tatsuo, Kaori)
- Add pg_conversion table (Tatsuo)
- Add SQL99 CONVERT() function (Tatsuo)
- pg_dumpall, pg_controldata, and pg_resetxlog now national-language aware (Peter)
- New and updated translations

E.152.3.10. Server-side Languages

- Allow recursive SQL function (Peter)
- Change PL/Tcl build to use configured compiler and Makefile.shlib (Peter)
- Overhaul the PL/pgSQL FOUND variable to be more Oracle-compatible (Neil, Tom)
- Allow PL/pgSQL to handle quoted identifiers (Tom)
- Allow set-returning PL/pgSQL functions (Neil)
- Make PL/pgSQL schema-aware (Joe)
- Remove some memory leaks (Nigel J. Andrews, Tom)
Appendix E. Release Notes

E.152.3.11. psql

• Don’t lowercase psql \connect database name for 7.2.0 compatibility (Tom)
• Add psql \timing to time user queries (Greg Sabino Mullane)
• Have psql \d show index information (Greg Sabino Mullane)
• New psql \dD shows domains (Jonathan Eisler)
• Allow psql to show rules on views (Paul ?)
• Fix for psql variable substitution (Tom)
• Allow psql \d to show temporary table structure (Tom)
• Allow psql \d to show foreign keys (Rod)
• Fix \? to honor \pset pager (Bruce)
• Have psql reports its version number on startup (Tom)
• Allow \copy to specify column names (Tom)

E.152.3.12. libpq

• Add ~/.pgpass to store host/user password combinations (Alvaro Herrera)
• Add PQunescapeBytea() function to libpq (Patrick Welche)
• Fix for sending large queries over non-blocking connections (Bernhard Herzog)
• Fix for libpq using timers on Win9X (David Ford)
• Allow libpq notify to handle servers with different-length identifiers (Tom)
• Add libpq PQescapeString() and PQescapeBytea() to Windows (Bruce)
• Fix for SSL with non-blocking connections (Jack Bates)
• Add libpq connection timeout parameter (Denis A Ustimenko)

E.152.3.13. JDBC

• Allow JDBC to compile with JDK 1.4 (Dave)
• Add JDBC 3 support (Barry)
• Allows JDBC to set loglevel by adding ?loglevel=X to the connection URL (Barry)
• Add Driver.info() message that prints out the version number (Barry)
• Add updateable result sets (Raghu Nidagal, Dave)
• Add support for callable statements (Paul Bethe)
• Add query cancel capability
• Add refresh row (Dave)
• Fix MD5 encryption handling for multibyte servers (Jun Kawai)
• Add support for prepared statements (Barry)
E.152.3.14. Miscellaneous Interfaces

- Fixed ECPG bug concerning octal numbers in single quotes (Michael)
- Move src/interfaces/libpgeasy to http://gborg.postgresql.org (Marc, Bruce)
- Improve Python interface (Elliot Lee, Andrew Johnson, Greg Copeland)
- Add libpgtcl connection close event (Gerhard Hintermayer)
- Move src/interfaces/libpq++ to http://gborg.postgresql.org (Marc, Bruce)
- Move src/interfaces/odbc to http://gborg.postgresql.org (Marc)
- Move src/interfaces/libpgeasy to http://gborg.postgresql.org (Marc, Bruce)
- Move src/interfaces/perl5 to http://gborg.postgresql.org (Marc, Bruce)
- Remove src/bin/pgaccess from main tree, now at http://www.pgaccess.org (Bruce)
- Add pg_on_connection_loss command to libpgtcl (Gerhard Hintermayer, Tom)

E.152.3.15. Source Code

- Fix for parallel make (Peter)
- AIX fixes for linking Tcl (Andreas Zeugswetter)
- Allow PL/Perl to build under Cygwin (Jason Tishler)
- Improve MIPS compiles (Peter, Oliver Elphick)
- Require Autoconf version 2.53 (Peter)
- Require readline and zlib by default in configure (Peter)
- Allow Solaris to use Intimate Shared Memory (ISM), for performance (Scott Brunza, P.J. Josh Rovero)
- Always enable syslog in compile, remove --enable-syslog option (Tatsuo)
- Always enable multibyte in compile, remove --enable-multibyte option (Tatsuo)
- Always enable locale in compile, remove --enable-locale option (Peter)
- Fix for Win9x DLL creation (Magnus Naeslund)
- Fix for link() usage by WAL code on Windows, BeOS (Jason Tishler)
- Add sys/types.h to c.h, remove from main files (Peter, Bruce)
- Fix AIX hang on SMP machines (Tomoyuki Niijima)
- AIX SMP hang fix (Tomoyuki Niijima)
- Fix pre-1970 date handling on newer glibc libraries (Tom)
- Fix PowerPC SMP locking (Tom)
- Prevent gcc -ffast-math from being used (Peter, Tom)
- Bison >= 1.50 now required for developer builds
- Kerberos 5 support now builds with Heimdal (Peter)
- Add appendix in the User’s Guide which lists SQL features (Thomas)
- Improve loadable module linking to use RTLD_NOW (Tom)
Appendix E. Release Notes

- New error levels WARNING, INFO, LOG, DEBUG[1-5] (Bruce)
- New src/port directory holds replaced libc functions (Peter, Bruce)
- New pg_namespace system catalog for schemas (Tom)
- Add pg_class.relnamespace for schemas (Tom)
- Add pg_type.typnamespace for schemas (Tom)
- Add pg_proc.pronamespace for schemas (Tom)
- Restructure aggregates to have pg_proc entries (Tom)
- System relations now have their own namespace, pg_* test not required (Fernando Nasser)
- Rename TOAST index names to be *_index rather than *_idx (Neil)
- Add namespaces for operators, opclasses (Tom)
- Add additional checks to server control file (Thomas)
- New Polish FAQ (Marcin Mazurek)
- Add Posix semaphore support (Tom)
- Document need for reindex (Bruce)
- Rename some internal identifiers to simplify Windows compile (Jan, Katherine Ward)
- Add documentation on computing disk space (Bruce)
- Remove KSQO from GUC (Bruce)
- Fix memory leak in rtree (Kenneth Been)
- Modify a few error messages for consistency (Bruce)
- Remove unused system table columns (Peter)
- Make system columns NOT NULL where appropriate (Tom)
- Clean up use of sprintf in favor of snprintf() (Neil, Jukka Holappa)
- Remove OPAQUE and create specific subtypes (Tom)
- Cleanups in array internal handling (Joe, Tom)
- Disallow pg_atoi("") (Bruce)
- Remove parameter wal_files because WAL files are now recycled (Bruce)
- Add version numbers to heap pages (Tom)

E.152.3.16. Contrib

- Allow inet arrays in /contrib/array (Neil)
- GiST fixes (Teodor Sigaev, Neil)
- Upgrade /contrib/mysql
- Add /contrib/dbsize which shows table sizes without vacuum (Peter)
- Add /contrib/intagg, integer aggregator routines (mlw)
- Improve /contrib/oid2name (Neil, Bruce)
- Improve /contrib/tsearch (Oleg, Teodor Sigaev)
E.153. Release 7.2.8

Release date: 2005-05-09

This release contains a variety of fixes from 7.2.7, including one security-related issue.

E.153.1. Migration to Version 7.2.8

A dump/restore is not required for those running 7.2.X.

E.153.2. Changes

- Repair ancient race condition that allowed a transaction to be seen as committed for some purposes (eg SELECT FOR UPDATE) slightly sooner than for other purposes
  This is an extremely serious bug since it could lead to apparent data inconsistencies being briefly visible to applications.
- Repair race condition between relation extension and VACUUM
  This could theoretically have caused loss of a page’s worth of freshly-inserted data, although the scenario seems of very low probability. There are no known cases of it having caused more than an Assert failure.
- Fix \texttt{EXTRACT(EPOCH) for TIME WITH TIME ZONE} values
- Additional buffer overrun checks in plpgsql (Neil)
Appendix E. Release Notes

- Fix pg_dump to dump index names and trigger names containing % correctly (Neil)
- Prevent to_char(interval) from dumping core for month-related formats
- Fix contrib/pgcrypto for newer OpenSSL builds (Marko Kreen)

E.154. Release 7.2.7

Release date: 2005-01-31

This release contains a variety of fixes from 7.2.6, including several security-related issues.

E.154.1. Migration to Version 7.2.7

A dump/restore is not required for those running 7.2.X.

E.154.2. Changes

- Disallow LOAD to non-superusers
  On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), LOAD can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.
- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins
  The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.
- Fix display of negative intervals in SQL and GERMAN datestyles

E.155. Release 7.2.6

Release date: 2004-10-22

This release contains a variety of fixes from 7.2.5.
**E.155.1. Migration to Version 7.2.6**

A dump/restore is not required for those running 7.2.X.

**E.155.2. Changes**

- Repair possible failure to update hint bits on disk
  Under rare circumstances this oversight could lead to “could not access transaction status” failures, which qualifies it as a potential-data-loss bug.
- Ensure that hashed outer join does not miss tuples
  Very large left joins using a hash join plan could fail to output unmatched left-side rows given just the right data distribution.
- Disallow running pg_ctl as root
  This is to guard against any possible security issues.
- Avoid using temp files in /tmp in make_oidjoins_check
  This has been reported as a security issue, though it’s hardly worthy of concern since there is no reason for non-developers to use this script anyway.
- Update to newer versions of Bison

**E.156. Release 7.2.5**

**Release date:** 2004-08-16

This release contains a variety of fixes from 7.2.4.

**E.156.1. Migration to Version 7.2.5**

A dump/restore is not required for those running 7.2.X.

**E.156.2. Changes**

- Prevent possible loss of committed transactions during crash
  Due to insufficient interlocking between transaction commit and checkpointing, it was possible for transactions committed just before the most recent checkpoint to be lost, in whole or in part, following a database crash and restart. This is a serious bug that has existed since PostgreSQL 7.1.
- Fix corner case for btree search in parallel with first root page split
- Fix buffer overrun in to_ascii (Guido Notari)
- Fix core dump in deadlock detection on machines where char is unsigned
• Fix failure to respond to `pg_ctl stop -m fast` after Async_NotifyHandler runs
• Repair memory leaks in `pg_dump`
• Avoid conflict with system definition of `isblank()` function or macro

**E.157. Release 7.2.4**

**Release date:** 2003-01-30

This release contains a variety of fixes for version 7.2.3, including fixes to prevent possible data loss.

**E.157.1. Migration to Version 7.2.4**

A dump/restore is *not* required for those running version 7.2.*.

**E.157.2. Changes**

• Fix some additional cases of `VACUUM "No one parent tuple was found"` error
• Prevent `VACUUM` from being called inside a function (Bruce)
• Ensure `pg_clog` updates are sync’d to disk before marking checkpoint complete
• Avoid integer overflow during large hash joins
• Make `GROUP` commands work when `pg_group.grolist` is large enough to be toasted
• Fix errors in `datetime` tables; some timezone names weren’t being recognized
• Fix integer overflows in `circle_poly()`, `path_encode()`, `path_add()` (Neil)
• Repair long-standing logic errors in `lseg_eq()`, `lseg_ne()`, `lseg_center()`

**E.158. Release 7.2.3**

**Release date:** 2002-10-01

This release contains a variety of fixes for version 7.2.2, including fixes to prevent possible data loss.

**E.158.1. Migration to Version 7.2.3**

A dump/restore is *not* required for those running version 7.2.*.
E.158.2. Changes

- Prevent possible compressed transaction log loss (Tom)
- Prevent non-superuser from increasing most recent vacuum info (Tom)
- Handle pre-1970 date values in newer versions of glibc (Tom)
- Fix possible hang during server shutdown
- Prevent spinlock hangs on SMP PPC machines (Tomoyuki Niijima)
- Fix pg_dump to properly dump FULL JOIN USING (Tom)

E.159. Release 7.2.2

**Release date:** 2002-08-23

This release contains a variety of fixes for version 7.2.1.

E.159.1. Migration to Version 7.2.2

A dump/restore is *not* required for those running version 7.2.*.

E.159.2. Changes

- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)
- Fix for compressed transaction log id wraparound (Tom)
- Fix PQescapeBytea/PQunescapeBytea so that they handle bytes > 0x7f (Tatsuo)
- Fix for psql and pg_dump crashing when invoked with non-existent long options (Tatsuo)
- Fix crash when invoking geometric operators (Tom)
- Allow OPEN cursor(args) (Tom)
- Fix for rtree_gist index build (Teodor)
- Fix for dumping user-defined aggregates (Tom)
- contrib/intarray fixes (Oleg)
- Fix for complex UNION/EXCEPT/INTERSECT queries using parens (Tom)
- Fix to pg_convert (Tatsuo)
- Fix for crash with long DATA strings (Thomas, Neil)
- Fix for repeat(), lpad(), rpad() and long strings (Neil)
E.160. Release 7.2.1

Release date: 2002-03-21

This release contains a variety of fixes for version 7.2.

E.160.1. Migration to Version 7.2.1

A dump/restore is not required for those running version 7.2.

E.160.2. Changes

- Ensure that sequence counters do not go backwards after a crash (Tom)
- Fix pgaccess kanji-conversion key binding (Tatsuo)
- Optimizer improvements (Tom)
- Cash I/O improvements (Tom)
- New Russian FAQ
- Compile fix for missing AuthBlockSig (Heiko)
- Additional time zones and time zone fixes (Thomas)
- Allow psql \connect to handle mixed case database and user names (Tom)
- Return proper OID on command completion even with ON INSERT rules (Tom)
- Allow COPY FROM to use 8-bit DELIMITERS (Tatsuo)
- Fix bug in extract/date_part for milliseconds/microseconds (Tatsuo)
- Improve handling of multiple UNIONs with different lengths (Tom)
- contrib/btree_gist improvements (Teodor Sigaev)
- contrib/tsearch dictionary improvements, see README.tsearch for an additional installation step (Thomas T. Thai, Teodor Sigaev)
- Fix for array subscripts handling (Tom)
- Allow EXECUTE of “CREATE TABLE AS ... SELECT” in PL/pgSQL (Tom)

E.161. Release 7.2

Release date: 2002-02-04
E.161.1. Overview

This release improves PostgreSQL for use in high-volume applications.

Major changes in this release:

VACUUM

Vacuuming no longer locks tables, thus allowing normal user access during the vacuum. A new VACUUM FULL command does old-style vacuum by locking the table and shrinking the on-disk copy of the table.

Transactions

There is no longer a problem with installations that exceed four billion transactions.

OIDs

OIDs are now optional. Users can now create tables without OIDs for cases where OID usage is excessive.

Optimizer

The system now computes histogram column statistics during ANALYZE, allowing much better optimizer choices.

Security

A new MD5 encryption option allows more secure storage and transfer of passwords. A new Unix-domain socket authentication option is available on Linux and BSD systems.

Statistics

Administrators can use the new table access statistics module to get fine-grained information about table and index usage.

Internationalization

Program and library messages can now be displayed in several languages.

E.161.2. Migration to Version 7.2

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

Observe the following incompatibilities:

- The semantics of the VACUUM command have changed in this release. You might wish to update your maintenance procedures accordingly.
- In this release, comparisons using = NULL will always return false (or NULL, more precisely). Previous releases automatically transformed this syntax to IS NULL. The old behavior can be re-enabled using a postgresql.conf parameter.
- The pg_hba.conf and pg_ident.conf configuration is now only reloaded after receiving a SIGHUP signal, not with each connection.
- The function octet_length() now returns the uncompressed data length.
- The date/time value ‘current’ is no longer available. You will need to rewrite your applications.
- The timestamp(), time(), and interval() functions are no longer available. Instead of timestamp(), use timestamp ‘string’ or CAST.
The `SELECT ... LIMIT #,#` syntax will be removed in the next release. You should change your queries to use separate `LIMIT` and `OFFSET` clauses, e.g. `LIMIT 10 OFFSET 20`.

### E.161.3. Changes

#### E.161.3.1. Server Operation

- Create temporary files in a separate directory (Bruce)
- Delete orphaned temporary files on postmaster startup (Bruce)
- Added unique indexes to some system tables (Tom)
- System table operator reorganization (Oleg Bartunov, Teodor Sigaev, Tom)
- Renamed `pg_log` to `pg_clog` (Tom)
- Enable `SIGTERM, SIGQUIT` to kill backends (Jan)
- Removed compile-time limit on number of backends (Tom)
- Better cleanup for semaphore resource failure (Tatsuo, Tom)
- Allow safe transaction ID wraparound (Tom)
- Removed OIDs from some system tables (Tom)
- Removed "triggered data change violation" error check (Tom)
- SPI portal creation of prepared/saved plans (Jan)
- Allow SPI column functions to work for system columns (Tom)
- Long value compression improvement (Tom)
- Statistics collector for table, index access (Jan)
- Truncate extra-long sequence names to a reasonable value (Tom)
- Measure transaction times in milliseconds (Thomas)
- Fix TID sequential scans (Hiroshi)
- Superuser ID now fixed at 1 (Peter E)
- New `pg_ctl "reload"` option (Tom)

#### E.161.3.2. Performance

- Optimizer improvements (Tom)
- New histogram column statistics for optimizer (Tom)
- Reuse write-ahead log files rather than discarding them (Tom)
- Cache improvements (Tom)
- IS NULL, IS NOT NULL optimizer improvement (Tom)
- Improve lock manager to reduce lock contention (Tom)
- Keep relcache entries for index access support functions (Tom)
- Allow better selectivity with NaN and infinities in NUMERIC (Tom)
• R-tree performance improvements (Kenneth Been)
• B-tree splits more efficient (Tom)

E.161.3.3. Privileges

• Change UPDATE, DELETE privileges to be distinct (Peter E)
• New REFERENCES, TRIGGER privileges (Peter E)
• Allow GRANT/REVOKE to/from more than one user at a time (Peter E)
• New has_table_privilege() function (Joe Conway)
• Allow non-superuser to vacuum database (Tom)
• New SET SESSION AUTHORIZATION command (Peter E)
• Fix bug in privilege modifications on newly created tables (Tom)
• Disallow access to pg_statistic for non-superuser, add user-accessible views (Tom)

E.161.3.4. Client Authentication

• Fork postmaster before doing authentication to prevent hangs (Peter E)
• Add ident authentication over Unix domain sockets on Linux, *BSD (Helge Bahmann, Oliver Elphick, Teodor Sigaev, Bruce)
• Add a password authentication method that uses MD5 encryption (Bruce)
• Allow encryption of stored passwords using MD5 (Bruce)
• PAM authentication (Dominic J. Eidson)
• Load pg_hba.conf and pg_ident.conf only on startup and SIGHUP (Bruce)

E.161.3.5. Server Configuration

• Interpretation of some time zone abbreviations as Australian rather than North American now settable at run time (Bruce)
• New parameter to set default transaction isolation level (Peter E)
• New parameter to enable conversion of "expr = NULL" into "expr IS NULL", off by default (Peter E)
• New parameter to control memory usage by VACUUM (Tom)
• New parameter to set client authentication timeout (Tom)
• New parameter to set maximum number of open files (Tom)

E.161.3.6. Queries

• Statements added by INSERT rules now execute after the INSERT (Jan)
Appendix E. Release Notes

- Prevent unadorned relation names in target list (Bruce)
- NULLs now sort after all normal values in ORDER BY (Tom)
- New IS UNKNOWN, IS NOT UNKNOWN Boolean tests (Tom)
- New SHARE UPDATE EXCLUSIVE lock mode (Tom)
- New EXPLAIN ANALYZE command that shows run times and row counts (Martijn van Oosterhout)
- Fix problem with LIMIT and subqueries (Tom)
- Fix for LIMIT, DISTINCT ON pushed into subqueries (Tom)
- Fix nested EXCEPT/INTERSECT (Tom)

E.161.3.7. Schema Manipulation

- Fix SERIAL in temporary tables (Bruce)
- Allow temporary sequences (Bruce)
- Sequences now use int8 internally (Tom)
- New SERIAL8 creates int8 columns with sequences, default still SERIAL4 (Tom)
- Make OIDs optional using WITHOUT OIDS (Tom)
- Add %TYPE syntax to CREATE TYPE (Ian Lance Taylor)
- Add ALTER TABLE / DROP CONSTRAINT for CHECK constraints (Christopher Kings-Lynne)
- New CREATE OR REPLACE FUNCTION to alter existing function (preserving the function OID) (Gavin Sherry)
- Add ALTER TABLE / ADD [ UNIQUE | PRIMARY ] (Christopher Kings-Lynne)
- Allow column renaming in views
- Make ALTER TABLE / RENAME COLUMN update column names of indexes (Brent Verner)
- Fix for ALTER TABLE / DROP CONSTRAINT ... CHECK with inherited tables (Stephan Szabo)
- ALTER TABLE RENAME update foreign-key trigger arguments correctly (Brent Verner)
- DROP AGGREGATE and COMMENT ON AGGREGATE now accept an aggtype (Tom)
- Add automatic return type data casting for SQL functions (Tom)
- Allow GiST indexes to handle NULLs and multikey indexes (Oleg Bartunov, Teodor Sigaev, Tom)
- Enable partial indexes (Martijn van Oosterhout)

E.161.3.8. Utility Commands

- Add RESET ALL, SHOW ALL (Marko Kreen)
- CREATE/ALTER USER/GROUP now allow options in any order (Vince)
- Add LOCK A, B, C functionality (Neil Padgett)
- New ENCRYPTED/UNENCRYPTED option to CREATE/ALTER USER (Bruce)
Appendix E. Release Notes

- New light-weight VACUUM does not lock table; old semantics are available as VACUUM FULL (Tom)
- Disable COPY TO/FROM on views (Bruce)
- COPY DELIMITERS string must be exactly one character (Tom)
- VACUUM warning about index tuples fewer than heap now only appears when appropriate (Martijn van Oosterhout)
- Fix privilege checks for CREATE INDEX (Tom)
- Disallow inappropriate use of CREATE/DROP INDEX/TRIGGER/VIEW (Tom)

E.161.3.9. Data Types and Functions

- SUM(), AVG(), COUNT() now uses int8 internally for speed (Tom)
- Add convert(), convert2() (Tatsuo)
- New function bit_length() (Peter E)
- Make the “n” in CHAR(n)/VARCHAR(n) represents letters, not bytes (Tatsuo)
- CHAR(), VARCHAR() now reject strings that are too long (Peter E)
- BIT VARYING now rejects bit strings that are too long (Peter E)
- BIT now rejects bit strings that do not match declared size (Peter E)
- INET, CIDR text conversion functions (Alex Pilosov)
- INET, CIDR operators << and <<= indexable (Alex Pilosov)
- Bytea \### now requires valid three digit octal number
- Bytea comparison improvements, now supports =, <>, >, >=, <, and <=
- Bytea now supports B-tree indexes
- Bytea now supports LIKE, LIKE...ESCAPE, NOT LIKE, NOT LIKE...ESCAPE
- Bytea now supports concatenation
- New bytea functions: position, substring, trim, btrim, and length
- New encode() function mode, "escaped", converts minimally escaped bytea to/from text
- Add pg_database_encoding_max_length() (Tatsuo)
- Add pg_client_encoding() function (Tatsuo)
- now() returns time with millisecond precision (Thomas)
- New TIMESTAMP WITHOUT TIMEZONE data type (Thomas)
- Add ISO date/time specification with "T", yyyy-mm-ddThh:mm:ss (Thomas)
- New xid/int comparison functions (Hirosi)
- Add precision to TIME, TIMESTAMP, and INTERVAL data types (Thomas)
- Modify type coercion logic to attempt binary-compatible functions first (Tom)
- New encode() function installed by default (Marko Kreen)
- Improved to_*( ) conversion functions (Karel Zak)
- Optimize LIKE/ILIKE when using single-byte encodings (Tatsuo)
Appendix E. Release Notes

- New functions in contrib/pgcrypto: crypt(), hmac(), encrypt(), gen_salt() (Marko Kreen)
- Correct description of translate() function (Bruce)
- Add INTERVAL argument for SET TIME ZONE (Thomas)
- Add INTERVAL YEAR TO MONTH (etc.) syntax (Thomas)
- Optimize length functions when using single-byte encodings (Tatsuo)
- Fix path_inter, path_distance, path_length, dist_ppath to handle closed paths (Curtis Barrett, Tom)
- octet_length(text) now returns non-compressed length (Tatsuo, Bruce)
- Handle "July" full name in date/time literals (Greg Sabino Mullane)
- Some datatype() function calls now evaluated differently
- Add support for Julian and ISO time specifications (Thomas)

E.161.3.10. Internationalization

- National language support in psql, pg_dump, libpq, and server (Peter E)
- Message translations in Chinese (simplified, traditional), Czech, French, German, Hungarian, Russian, Swedish (Peter E, Serguei A. Mokhov, Karel Zak, Weiping He, Zhenbang Wei, Kovacs Zoltan)
- Make trim, ltrim, rtrim, btrim, lpad, rpad, translate multibyte aware (Tatsuo)
- Add LATIN5,6,7,8,9,10 support (Tatsuo)
- Add ISO 8859-5,6,7,8 support (Tatsuo)
- Correct LATIN5 to mean ISO-8859-9, not ISO-8859-5 (Tatsuo)
- Make mic2ascii() non-ASCII aware (Tatsuo)
- Reject invalid multibyte character sequences (Tatsuo)

E.161.3.11. PL/pgSQL

- Now uses portals for SELECT loops, allowing huge result sets (Jan)
- CURSOR and REFCURSOR support (Jan)
- Can now return open cursors (Jan)
- Add ELSEIF (Klaus Reger)
- Improve PL/pgSQL error reporting, including location of error (Tom)
- Allow IS or FOR key words in cursor declaration, for compatibility (Bruce)
- Fix for SELECT ... FOR UPDATE (Tom)
- Fix for PERFORM returning multiple rows (Tom)
- Make PL/pgSQL use the server’s type coercion code (Tom)
- Memory leak fix (Jan, Tom)
- Make trailing semicolon optional (Tom)
Appendix E. Release Notes

E.161.3.12. PL/Perl

• New untrusted PL/Perl (Alex Pilosov)
• PL/Perl is now built on some platforms even if libperl is not shared (Peter E)

E.161.3.13. PL/Tcl

• Now reports errorInfo (Vsevolod Lobko)
• Add spi_lastoid function (bob@redivi.com)

E.161.3.14. PL/Python

• ...is new (Andrew Bosma)

E.161.3.15. psql

• \d displays indexes in unique, primary groupings (Christopher Kings-Lynne)
• Allow trailing semicolons in backslash commands (Greg Sabino Mullane)
• Read password from /dev/tty if possible
• Force new password prompt when changing user and database (Tatsuo, Tom)
• Format the correct number of columns for Unicode (Patrice)

E.161.3.16. libpq

• New function PQescapeString() to escape quotes in command strings (Florian Weimer)
• New function PQescapeBytea() escapes binary strings for use as SQL string literals

E.161.3.17. JDBC

• Return OID of INSERT (Ken K)
• Handle more data types (Ken K)
• Handle single quotes and newlines in strings (Ken K)
• Handle NULL variables (Ken K)
• Fix for time zone handling (Barry Lind)
• Improved Druid support
• Allow eight-bit characters with non-multibyte server (Barry Lind)
• Support BIT, BINARY types (Ned Wolpert)
• Reduce memory usage (Michael Stephens, Dave Cramer)
Appendix E. Release Notes

- Update DatabaseMetaData (Peter E)
- Add DatabaseMetaData.getCatalogs() (Peter E)
- Encoding fixes (Anders Bengtsson)
- Get/setCatalog methods (Jason Davies)
- DatabaseMetaData.getColumns() now returns column defaults (Jason Davies)
- DatabaseMetaData.getColumns() performance improvement (Jeroen van Vianen)
- Some JDBC1 and JDBC2 merging (Anders Bengtsson)
- Transaction performance improvements (Barry Lind)
- Array fixes (Greg Zoller)
- Serialize addition
- Fix batch processing (Rene Pijlman)
- ExecSQL method reorganization (Anders Bengtsson)
- GetColumn() fixes (Jeroen van Vianen)
- Fix isWriteable() function (Rene Pijlman)
- Improved passage of JDBC2 conformance tests (Rene Pijlman)
- Add bytea type capability (Barry Lind)
- Add isNullable() (Rene Pijlman)
- JDBC date/time test suite fixes (Liam Stewart)
- Fix for SELECT 'id' AS xxx FROM table (Dave Cramer)
- Fix DatabaseMetaData to show precision properly (Mark Lillywhite)
- New getImported/getExported keys (Jason Davies)
- MD5 password encryption support (Jeremy Wohl)
- Fix to actually use type cache (Ned Wolpert)

E.161.3.18. ODBC

- Remove query size limit (Hiroshi)
- Remove text field size limit (Hiroshi)
- Fix for SQLPrimaryKeys in multibyte mode (Hiroshi)
- Allow ODBC procedure calls (Hiroshi)
- Improve boolean handing (Aidan Mountford)
- Most configuration options now settable via DSN (Hiroshi)
- Multibyte, performance fixes (Hiroshi)
- Allow driver to be used with iODBC or unixODBC (Peter E)
- MD5 password encryption support (Bruce)
- Add more compatibility functions to odbc.sql (Peter E)
E.161.3.19. ECPG

- EXECUTE ... INTO implemented (Christof Petig)
- Multiple row descriptor support (e.g. CARDINALITY) (Christof Petig)
- Fix for GRANT parameters (Lee Kindness)
- Fix INITIALLY DEFERRED bug
- Various bug fixes (Michael, Christof Petig)
- Auto allocation for indicator variable arrays (int *ind_p=NULL)
- Auto allocation for string arrays (char **foo_pp=NULL)
- ECPGfree_auto_mem fixed
- All function names with external linkage are now prefixed by ECPG
- Fixes for arrays of structures (Michael)

E.161.3.20. Misc. Interfaces

- Python fix fetchone() (Gerhard Haring)
- Use UTF, Unicode in Tcl where appropriate (Vsevolod Lobko, Reinhard Max)
- Add Tcl COPY TO/FROM (ljb)
- Prevent output of default index op class in pg_dump (Tom)
- Fix libpgeasy memory leak (Bruce)

E.161.3.21. Build and Install

- Configure, dynamic loader, and shared library fixes (Peter E)
- Fixes in QNX 4 port (Bernd Tegge)
- Fixes in Cygwin and Windows ports (Jason Tishler, Gerhard Haring, Dmitry Yurtaev, Darko Prenosil, Mikhail Terekhov)
- Fix for Windows socket communication failures (Magnus, Mikhail Terekhov)
- Hurd compile fix (Oliver Elphick)
- BeOS fixes (Cyril Velter)
- Remove configure --enable-unicode-conversion, now enabled by multibyte (Tatsuo)
- AIX Fixes (Tatsuo, Andreas)
- Fix parallel make (Peter E)
- Install SQL language manual pages into OS-specific directories (Peter E)
- Rename config.h to pg_config.h (Peter E)
- Reorganize installation layout of header files (Peter E)
E.161.3.22. Source Code

- Remove SEP_CHAR (Bruce)
- New GUC hooks (Tom)
- Merge GUC and command line handling (Marko Kreen)
- Remove EXTEND INDEX (Martijn van Oosterhout, Tom)
- New pgindent utility to indent java code (Bruce)
- Remove define of true/false when compiling under C++ (Leandro Fanzone, Tom)
- pgindent fixes (Bruce, Tom)
- Replace strcasecmp() with strcmp() where appropriate (Peter E)
- Dynahash portability improvements (Tom)
- Add 'volatile' usage in spinlock structures
- Improve signal handling logic (Tom)

E.161.3.23. Contrib

- New contrib/rtree_gist (Oleg Bartunov, Teodor Sigaev)
- New contrib/tsearch full-text indexing (Oleg, Teodor Sigaev)
- Add contrib/dblink for remote database access (Joe Conway)
- contrib/ora2pg Oracle conversion utility (Gilles Darold)
- contrib/xml XML conversion utility (John Gray)
- contrib/fulltextindex fixes (Christopher Kings-Lynne)
- New contrib/fuzzystrmatch with levenshtein and metaphone, soundex merged (Joe Conway)
- Add contrib/intarray boolean queries, binary search, fixes (Oleg Bartunov)
- New pg_upgrade utility (Bruce)
- Add new pg_resetxlog options (Bruce, Tom)

E.162. Release 7.1.3

Release date: 2001-08-15

E.162.1. Migration to Version 7.1.3

A dump/restore is not required for those running 7.1.X.
E.162.2. Changes

Remove unused WAL segments of large transactions (Tom)
Multiactiohn rule fix (Tom)
PL/pgSQL memory allocation fix (Jan)
VACUUM buffer fix (Tom)
Regression test fixes (Tom)
pg_dump fixes for GRANT/REVOKE/comments on views, user-defined types (Tom)
Fix subselects with DISTINCT ON or LIMIT (Tom)
BeOS fix
Disable COPY TO/FROM a view (Tom)
Cygwin build (Jason Tishler)

E.163. Release 7.1.2

Release date: 2001-05-11

This has one fix from 7.1.1.

E.163.1. Migration to Version 7.1.2

A dump/restore is not required for those running 7.1.X.

E.163.2. Changes

Fix PL/pgSQL SELECTs when returning no rows
Fix for psql backslash core dump
Referential integrity privilege fix
Optimizer fixes
pg_dump cleanups

E.164. Release 7.1.1

Release date: 2001-05-05

This has a variety of fixes from 7.1.
Appendix E. Release Notes

E.164.1. Migration to Version 7.1.1

A dump/restore is not required for those running 7.1.

E.164.2. Changes

- Fix for numeric MODULO operator (Tom)
- pg_dump fixes (Philip)
- pg_dump can dump 7.0 databases (Philip)
- readline 4.2 fixes (Peter E)
- JOIN fixes (Tom)
- AIX, MSWIN, VAX, N32K fixes (Tom)
- Multibytes fixes (Tom)
- Unicode fixes (Tatsuo)
- Optimizer improvements (Tom)
- Fix for whole rows in functions (Tom)
- Fix for pg_ctl and option strings with spaces (Peter E)
- ODBC fixes (Hiroshi)
- EXTRACT can now take string argument (Thomas)
- Python fixes (Darcy)

E.165. Release 7.1

Release date: 2001-04-13

This release focuses on removing limitations that have existed in the PostgreSQL code for many years.

Major changes in this release:

Write-ahead Log (WAL)

To maintain database consistency in case of an operating system crash, previous releases of PostgreSQL have forced all data modifications to disk before each transaction commit. With WAL, only one log file must be flushed to disk, greatly improving performance. If you have been using -F in previous releases to disable disk flushes, you might want to consider discontinuing its use.

TOAST

TOAST - Previous releases had a compiled-in row length limit, typically 8k - 32k. This limit made storage of long text fields difficult. With TOAST, long rows of any length can be stored with good performance.

Outer Joins

We now support outer joins. The UNION/NOT IN workaround for outer joins is no longer required. We use the SQL92 outer join syntax.
Function Manager

The previous C function manager did not handle null values properly, nor did it support 64-bit CPU's (Alpha). The new function manager does. You can continue using your old custom functions, but you might want to rewrite them in the future to use the new function manager call interface.

Complex Queries

A large number of complex queries that were unsupported in previous releases now work. Many combinations of views, aggregates, UNION, LIMIT, cursors, subqueries, and inherited tables now work properly. Inherited tables are now accessed by default. Subqueries in FROM are now supported.

E.165.1. Migration to Version 7.1

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release.

E.165.2. Changes

Bug Fixes

---------
Many multibyte/Unicode/locale fixes (Tatsuo and others)
More reliable ALTER TABLE RENAME (Tom)
Kerberos V fixes (David Wragg)
Fix for INSERT INTO...SELECT where targetlist has subqueries (Tom)
Prompt username/password on standard error (Bruce)
Large objects inv_read/inv_write fixes (Tom)
Fixes for to_char(), to_date(), to_ascii(), and to_timestamp() (Karel, Daniel Baldoni)
Prevent query expressions from leaking memory (Tom)
Allow UPDATE of arrays elements (Tom)
Wake up lock waiters during cancel (Hiroshi)
Fix rare cursor crash when using hash join (Tom)
Fix for DROP TABLE/INDEX in rolled-back transaction (Hiroshi)
Fix psql crash from \l+ if MULTIBYTE enabled (Peter E)
Fix truncation of rule names during CREATE VIEW (Ross Reedstrom)
Fix PL/perl (Alex Kapranoff)
Disallow LOCK on views (Mark Hollomon)
Disallow INSERT/UPDATE/DELETE on views (Mark Hollomon)
Disallow DROP RULE, CREATE INDEX, TRUNCATE on views (Mark Hollomon)
Allow PL/pgSQL accept non-ASCII identifiers (Tatsuo)
Allow views to proper handle GROUP BY, aggregates, DISTINCT (Tom)
Fix rare failure with TRUNCATE command (Tom)
Allow UNION/INTERSECT/EXCEPT to be used with ALL, subqueries, views, DISTINCT, ORDER BY, SELECT...INTO (Tom)
Fix parser failures during aborted transactions (Tom)
Allow temporary relations to properly clean up indexes (Bruce)
Fix VACUUM problem with moving rows in same page (Tom)
Modify pg_dump to better handle user-defined items in template1 (Philip)
Allow LIMIT in VIEW (Tom)
Require cursor FETCH to honor LIMIT (Tom)
Allow PRIMARY/FOREIGN Key definitions on inherited columns (Stephan)
Appendix E. Release Notes

Allow ORDER BY, LIMIT in subqueries (Tom)
Allow UNION in CREATE RULE (Tom)
Make ALTER/DROP TABLE rollback-able (Vadim, Tom)
Store initdb collation in pg_control so collation cannot be changed (Tom)
Fix INSERT...SELECT with rules (Tom)
Fix FOR UPDATE inside views and subselects (Tom)
Fix OVERLAPS operators conform to SQL92 spec regarding NULLs (Tom)
Fix lpad() and rpad() to handle length less than input string (Tom)
Fix use of NOTIFY in some rules (Tom)
Overhaul btree code (Tom)
Fix NOT NULL use in Pl/pgSQL variables (Tom)
Overhaul GIST code (Oleg)
Fix CLUSTER to preserve constraints and column default (Tom)
Improved deadlock detection handling (Tom)
Allow multiple SERIAL columns in a table (Tom)
Prevent occasional index corruption (Vadim)

Enhancements
------------
Add OUTER JOINs (Tom)
Function manager overhaul (Tom)
Allow ALTER TABLE RENAME on indexes (Tom)
Improve CLUSTER (Tom)
Improve ps status display for more platforms (Peter E, Marc)
Improve CREATE FUNCTION failure message (Ross)
JDBC improvements (Peter, Travis Bauer, Christopher Cain, William Webber, Gunnar)
Grand Unified Configuration scheme/GUC. Many options can now be set in
data/postgresql.conf, postmaster/postgres flags, or SET commands (Peter E)
Improved handling of file descriptor cache (Tom)
New warning code about auto-created table alias entries (Bruce)
Overhaul initdb process (Tom, Peter E)
Overhaul of inherited tables; inherited tables now accessed by default;
   new ONLY key word prevents it (Chris Bitmead, Tom)
ODBC cleanups/improvements (Nick Gorham, Stephan Szabo, Zoltan Kovacs,
   Michael Fork)
Allow renaming of temp tables (Tom)
Overhaul memory manager contexts (Tom)
pg_dumpall uses CREATE USER or CREATE GROUP rather using COPY (Peter E)
Overhaul pg_dump (Philip Warner)
Allow pg_hba.conf secondary password file to specify only username (Peter E)
Allow TEMPORARY or TEMP key word when creating temporary tables (Bruce)
New memory leak checker (Karel)
New SET SESSION CHARACTERISTICS (Thomas)
Allow nested block comments (Thomas)
Add WITHOUT TIME ZONE type qualifier (Thomas)
New ALTER TABLE ADD CONSTRAINT (Stephan)
Use NUMERIC accumulators for INTEGER aggregates (Tom)
Overhaul aggregate code (Tom)
New VARIANCE and STDDEV() aggregates
Improve dependency ordering of pg_dump (Philip)
New pg_restore command (Philip)
New pg_dump tar output option (Philip)
New pg_dump of large objects (Philip)
New ESCAPE option to LIKE (Thomas)
New case-insensitive LIKE - ILIKE (Thomas)
Allow functional indexes to use binary-compatible type (Tom)
Appendix E. Release Notes

Allow SQL functions to be used in more contexts (Tom)
New pg_config utility (Peter E)
New PL/pgSQL EXECUTE command which allows dynamic SQL and utility statements (Jan)
New PL/pgSQL GET DIAGNOSTICS statement for SPI value access (Jan)
New quote_identifiers() and quote_literal() functions (Jan)
New ALTER TABLE table OWNER TO user command (Mark Hollomon)
Allow subselects in FROM, i.e. FROM (SELECT ...) [AS] alias (Tom)
Update PyGreSQL to version 3.1 (D’Arcy)
Store tables as files named by OID (Vadim)
New SQL function setval(seq,val,bool) for use in pg_dump (Philip)
Require DROP VIEW to remove views, no DROP TABLE (Mark)
Allow DROP VIEW view1, view2 (Mark)
Allow multiple objects in DROP INDEX, DROP RULE, and DROP TYPE (Tom)
Allow automatic conversion to/from Unicode (Tatsuo, Eiji)
New /contrib/pgcrypto hashing functions (Marko Kreen)
New pg_dumpall --globals-only option (Peter E)
New CHECKPOINT command for WAL which creates new WAL log file (Vadim)
New AT TIME ZONE syntax (Thomas)
Allow location of Unix domain socket to be configurable (David J. MacKenzie)
Allow postmaster to listen on a specific IP address (David J. MacKenzie)
Allow socket path name to be specified in hostname by using leading slash (David J. MacKenzie)
Allow CREATE DATABASE to specify template database (Tom)
New utility to convert MySQL schema dumps to SQL92 and PostgreSQL (Thomas)
New /contrib/rserv replication toolkit (Vadim)
New file format for COPY BINARY (Tom)
New /contrib/oid2name to map numeric files to table names (B Palmer)
New "idle in transaction" ps status message (Marc)
Update to pgaccess 0.98.7 (Constantin Teodorescu)
pg_ctl now defaults to -w (wait) on shutdown, new -l (log) option
Add rudimentary dependency checking to pg_dump (Philip)

Types
-----
Fix INET/CIDR type ordering and add new functions (Tom)
Make OID behave as an unsigned type (Tom)
Allow BIGINT as synonym for INT8 (Peter E)
New int2 and int8 comparison operators (Tom)
New BIT and BIT VARYING types (Adriaan Joubert, Tom, Peter E)
CHAR() no longer faster than VARCHAR() because of TOAST (Tom)
New GIST seg/cube examples (Gene Selkov)
Improved round(numeric) handling (Tom)
Fix CIDR output formatting (Tom)
New CIDR abbrev() function (Tom)

Performance
-------------
Write-Ahead Log (WAL) to provide crash recovery with less performance overhead (Vadim)
ANALYZE stage of VACUUM no longer exclusively locks table (Bruce)
Reduced file seeks (Denis Perchine)
Improve BTREE code for duplicate keys (Tom)
Store all large objects in a single table (Denis Perchine, Tom)
Improve memory allocation performance (Karel, Tom)

Source Code

1749
Appendix E. Release Notes

---

New function manager call conventions (Tom)
SGI portability fixes (David Kaelbling)
New configure --enable-syslog option (Peter E)
New BSDI README (Bruce)
configure script moved to top level, not /src (Peter E)
Makefile/configuration/compilation overhaul (Peter E)
New configure --with-python option (Peter E)
Solaris cleanups (Peter E)
Overhaul /contrib Makefiles (Karel)
New OpenSSL configuration option (Magnus, Peter E)
AIX fixes (Andreas)
QNIX fixes (Maurizio)
New heap_open(), heap_openr() API (Tom)
Remove colon and semi-colon operators (Thomas)
New pg_class.relkind value for views (Mark Hollomon)
Rename ichtar() to chr() (Karel)
New documentation for btrim(), ascii(), chr(), repeat() (Karel)
Fixes for NT/Cygwin (Pete Forman)
AIX port fixes (Andreas)
New BeOS port (David Reid, Cyril Velter)
Add proofreader's changes to docs (Addison-Wesley, Bruce)
New Alpha spinlock code (Adriaan Joubert, Compaq)
UnixWare port overhaul (Peter E)
New Darwin/MacOS X port (Peter Bierman, Bruce Hartzler)
New FreeBSD Alpha port (Alfred)
Overhaul shared memory segments (Tom)
Add IBM S/390 support (Neale Ferguson)
Moved macmanuf to /contrib (Larry Rosenman)
Syslog improvements (Larry Rosenman)
New template0 database that contains no user additions (Tom)
New /contrib/cube and /contrib/seg GIST sample code (Gene Selkov)
Allow NetBSD's libedit instead of readline (Peter)
Improved assembly language source code format (Bruce)
New contrib/pg_logger
New --template option to createdb
New contrib/pg_control utility (Oliver)
New FreeBSD tools ipc_check, start-scripts/freebsd

---

E.166. Release 7.0.3

Release date: 2000-11-11

This has a variety of fixes from 7.0.2.
Appendix E. Release Notes

E.166.1. Migration to Version 7.0.3

A dump/restore is *not* required for those running 7.0.*.

E.166.2. Changes

Jdbc fixes (Peter)
Large object fix (Tom)
Fix lean in COPY WITH OIDS leak (Tom)
Fix backwards-index-scan (Tom)
Fix SELECT ... FOR UPDATE so it checks for duplicate keys (Hiroshi)
Add --enable-syslog to configure (Marc)
Fix abort transaction at backend exit in rare cases (Tom)
Fix for psql \l+ when multibyte enabled (Tatsuo)
Allow PL/pgSQL to accept non ascii identifiers (Tatsuo)
Make vacuum always flush buffers (Tom)
Fix to allow cancel while waiting for a lock (Hiroshi)
Fix for memory allocation problem in user authentication code (Tom)
Remove bogus use of int4out() (Tom)
Fixes for multiple subqueries in COALESCE or BETWEEN (Tom)
Fix for failure of triggers on heap open in certain cases (Jeroen van Vianen)
Fix for erroneous selectivity of not-equals (Tom)
Fix for erroneous use of strcmp() (Tom)
Fix for bug where storage manager accesses items beyond end of file (Tom)
Fix to include kernel errno message in all smgr elog messages (Tom)
Fix for '.' not in PATH at build time (SL Baur)
Fix for out-of-file-descriptors error (Tom)
Fix to make pg_dump dump 'iscachable' flag for functions (Tom)
Fix for subselect in targetlist of Append node (Tom)
Fix for mergejoin plans (Tom)
Fix TRUNCATE failure on relations with indexes (Tom)
Avoid database-wide restart on write error (Hiroshi)
Fix nodeMaterial to honor chgParam by recomputing its output (Tom)
Fix VACUUM problem with moving chain of update row versions when source and destination of a row version lie on the same page (Tom)
Fix user.c CommandCounterIncrement (Tom)
Fix for AM/PM boundary problem in to_char() (Karel Zak)
Fix TIME aggregate handling (Tom)
Fix to_char() to avoid coredump on NULL input (Tom)
Buffer fix (Tom)
Fix for inserting/copying longer multibyte strings into char() data types (Tatsuo)
Fix for crash of backend, on abort (Tom)
Appendix E. Release Notes

E.167. Release 7.0.2

Release date: 2000-06-05

This is a repackaging of 7.0.1 with added documentation.

E.167.1. Migration to Version 7.0.2

A dump/restore is *not* required for those running 7.*.

E.167.2. Changes

Added documentation to tarball.

E.168. Release 7.0.1

Release date: 2000-06-01

This is a cleanup release for 7.0.

E.168.1. Migration to Version 7.0.1

A dump/restore is *not* required for those running 7.0.

E.168.2. Changes

- Fix many CLUSTER failures (Tom)
- Allow ALTER TABLE RENAME works on indexes (Tom)
- Fix pgsql to handle datetime->timestamp and timespan->interval (Bruce)
- New configure --with-setproctitle switch to use setproctitle() (Marc, Bruce)
- Fix the off by one errors in ResultSet from 6.5.3, and more.
- jdbc ResultSet fixes (Joseph Shraibman)
- optimizer tunings (Tom)
- Fix create user for pgaccess
- Fix for UNLISTEN failure
- IRIX fixes (David Kaelbling)
- QNX fixes (Andreas Kardos)
- Reduce COPY IN lock level (Tom)
- Change libpqeasy to use PQconnectdb() style parameters (Bruce)
- Fix pg_dump to handle OID indexes (Tom)
Fix small memory leak (Tom)
Solaris fix for createdb/dropdb (Tatsuo)
Fix for non-blocking connections (Alfred Perlstein)
Fix improper recovery after RENAME TABLE failures (Tom)
Copy pg_ident.conf.sample into /lib directory in install (Bruce)
Add SJIS UDC (NEC selection IBM kanji) support (Eiji Tokuya)
Fix too long syslog message (Tatsuo)
Fix problem with quoted indexes that are too long (Tom)
JDBC ResultSet.getTimestamp() fix (Gregory Krasnow & Floyd Marinescu)
ecpg changes (Michael)

E.169. Release 7.0

Release date: 2000-05-08

This release contains improvements in many areas, demonstrating the continued growth of PostgreSQL. There are more improvements and fixes in 7.0 than in any previous release. The developers have confidence that this is the best release yet; we do our best to put out only solid releases, and this one is no exception.

Major changes in this release:

Foreign Keys

Foreign keys are now implemented, with the exception of PARTIAL MATCH foreign keys. Many users have been asking for this feature, and we are pleased to offer it.

Optimizer Overhaul

Continuing on work started a year ago, the optimizer has been improved, allowing better query plan selection and faster performance with less memory usage.

Updated psql

psql, our interactive terminal monitor, has been updated with a variety of new features. See the psql manual page for details.

Join Syntax

SQL92 join syntax is now supported, though only as INNER JOIN for this release. JOIN, NATURAL JOIN, JOIN/USING, and JOIN/ON are available, as are column correlation names.

E.169.1. Migration to Version 7.0

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of PostgreSQL. For those upgrading from 6.5.*, you can instead use pg_upgrade to upgrade to this release; however, a full dump/reload installation is always the most robust method for upgrades.

Interface and compatibility issues to consider for the new release include:
Appendix E. Release Notes

- The date/time types `datetime` and `timespan` have been superseded by the SQL92-defined types `timestamp` and `interval`. Although there has been some effort to ease the transition by allowing PostgreSQL to recognize the deprecated type names and translate them to the new type names, this mechanism cannot be completely transparent to your existing application.

- The optimizer has been substantially improved in the area of query cost estimation. In some cases, this will result in decreased query times as the optimizer makes a better choice for the preferred plan. However, in a small number of cases, usually involving pathological distributions of data, your query times might go up. If you are dealing with large amounts of data, you might want to check your queries to verify performance.

- The JDBC and ODBC interfaces have been upgraded and extended.

- The string function `CHAR_LENGTH` is now a native function. Previous versions translated this into a call to `LENGTH`, which could result in ambiguity with other types implementing `LENGTH` such as the geometric types.

E.169.2. Changes

Bug Fixes
--------
Prevent function calls exceeding maximum number of arguments (Tom)
Improve CASE construct (Tom)
Fix SELECT `coalesce(f1,0)` FROM `int4_tbl` GROUP BY `f1` (Tom)
Fix SELECT `sentence.words[0]` FROM `sentence` GROUP BY `sentence.words[0]` (Tom)
Fix GROUP BY scan bug (Tom)
Improvements in SQL grammar processing (Tom)
Fix for views involved in INSERT ... SELECT ... (Tom)
Fix for SELECT `a/2, a/2` FROM `test_missing_target` GROUP BY `a/2` (Tom)
Fix for subselects in INSERT ... SELECT (Tom)
Prevent INSERT ... SELECT ... ORDER BY (Tom)
Fixes for relations greater than 2GB, including vacuum
Improve propagating system table changes to other backends (Tom)
Improve propagating user table changes to other backends (Tom)
Fix handling of temp tables in complex situations (Bruce, Tom)
Allow table locking at table open, improving concurrent reliability (Tom)
Properly quote sequence names in `pg_dump` (Ross J. Reedstrom)
Prevent DROP DATABASE while others accessing
Prevent any rows from being returned by GROUP BY if no rows processed (Tom)
Fix SELECT `COUNT(1)` FROM `table` WHERE ...' if no rows matching WHERE (Tom)
Fix `pg_upgrade` so it works for MVCC (Tom)
Fix for "f1 datetime DEFAULT 'now" (Tom)
Fix problems with `CURRENT_DATE` used in `DEFAULT` (Tom)
Allow comment-only lines, and `;;` lines too. (Tom)
Improve recovery after failed disk writes, disk full (Hiroshi)
Fix cases where table is mentioned in `FROM` but not joined (Tom)
Allow HAVING clause without aggregate functions (Tom)
Fix for "--" comment and no trailing newline, as seen in perl interface
Improve `pg_dump` failure error reports (Bruce)
Allow sorts and hashes to exceed 2GB file sizes (Tom)
Fix for `pg_dump` dumping of inherited rules (Tom)
Fix for NULL handling comparisons (Tom)
Fix inconsistent state caused by failed CREATE/DROP commands (Hiroshi)
Fix for dbname with dash
Appendix E. Release Notes

Prevent DROP INDEX from interfering with other backends (Tom)
Fix file descriptor leak in verify_password()
Fix for "Unable to identify an operator -5" problem
Fix ODBC so no segfault if CommLog and Debug enabled (Dirk Niggemann)
Fix for recursive exit call (Massimo)
Fix for extra-long timezones (Jeroen van Vianen)
Make pg_dump preserve primary key information (Peter E)
Prevent databases with single quotes (Peter E)
Prevent DROP DATABASE inside transaction (Peter E)
ecpg memory leak fixes (Stephen Birch)
Fix for SELECT null::text, SELECT int4fac(null) and SELECT 2 + (null) (Tom)
Y2K timestamp fix (Massimo)
Fix for VACUUM 'HEAP_MOVED_IN was not expected’ errors (Tom)
Fix for views with tables/columns containing spaces (Tom)
Prevent privileges on indexes (Peter E)
Fix for spinlock stuck problem when error is generated (Hiroshi)
Fix ipcclean on Linux
Fix handling of NULL constraint conditions (Tom)
Fix memory leak in odbc driver (Nick Gorham)
Fix for privilege check on UNION tables (Tom)
Fix to allow SELECT ’a’ LIKE ’a’ (Tom)
Fix for SELECT 1 + NULL (Tom)
Fixes to CHAR
Fix log() on numeric type (Tom)
Deprecate ’:’ and ’;’ operators
Allow vacuum of temporary tables
Disallow inherited columns with the same name as new columns
Recover or force failure when disk space is exhausted (Hiroshi)
Fix INSERT INTO ... SELECT with AS columns matching result columns
Fix INSERT ... SELECT ... GROUP BY groups by target columns not source columns (Tom)
Fix CREATE TABLE test (a char(5) DEFAULT text ", b int4) with INSERT (Tom)
Fix UNION with LIMIT
Fix CREATE TABLE x AS SELECT 1 UNION SELECT 2
Fix CREATE TABLE test(col char(2) DEFAULT user)
Fix mismatched types in CREATE TABLE ... DEFAULT
Fix SELECT * FROM pg_class where oid in (0,-1)
Fix SELECT COUNT(’asdf’) FROM pg_class WHERE oid=12
Prevent user who can create databases can modifying pg_database table (Peter E)
Fix btree to give a useful elog when key > 1/2 (page - overhead) (Tom)
Fix INSERT of 0.0 into DECIMAL(4,4) field (Tom)

Enhancements
-------------
New CLI interface include file sqlcli.h, based on SQL3/SQI98
Remove all limits on query length, row length limit still exists (Tom)
Update jdbc protocol to 2.0 (Jens Glaser <jens@jens.de>)
Add TRUNCATE command to quickly truncate relation (Mike Mascari)
Fix to give super user and createdb user proper update catalog rights (Peter E)
Allow ecpg bool variables to have NULL values (Christof)
Issue ecpg error if NULL value for variable with no NULL indicator (Christof)
Allow ~C to cancel COPY command (Massimo)
Add SET FSYNC and SHOW PG_OPTIONS commands (Massimo)
Function name overloading for dynamically-loaded C functions (Frankpitt)
Add CmdTuples() to libpq++ (Vince)
New CREATE CONSTRAINT TRIGGER and SET CONSTRAINTS commands (Jan)
Allow CREATE FUNCTION/WITH clause to be used for all language types
configure --enable-debug adds -g (Peter E)
configure --disable-debug removes -g (Peter E)
Allow more complex default expressions (Tom)
First real FOREIGN KEY constraint trigger functionality (Jan)
Add FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
Add FOREIGN KEY ... MATCH <Unspecified> referential actions (Don Baccus)
Allow WHERE restriction on ctid (physical heap location) (Hiroshi)
Move pginterface from contrib to interface directory, rename to pgeasy (Bruce)
Change pgeasy connectdb() parameter ordering (Bruce)
Require SELECT DISTINCT target list to have all ORDER BY columns (Tom)
Add Oracle’s COMMENT ON command (Mike Mascari <mascarim@yahoo.com>)
libpq’s PQsetNoticeProcessor function now returns previous hook (Peter E)
Prevent PQsetNoticeProcessor from being set to NULL (Peter E)
Make USING in COPY optional (Bruce)
Allow subselects in the target list (Tom)
Allow subselects on the left side of comparison operators (Tom)
New parallel regression test (Jan)
Change backend-side COPY to write files with permissions 644 not 666 (Tom)
Force permissions on PGDATA directory to be secure, even if it exists (Tom)
Added psqi LASTOID variable to return last inserted oid (Peter E)
Allow concurrent vacuum and remove pg_vlock vacuum lock file (Tom)
Add privilege check for vacuum (Peter E)
New libpq functions to allow asynchronous connections: PQconnectStart(),
  PQconnectPoll(), PQresetStart(), PQresetPoll(), PQsetenvStart(),
  PQsetenvPoll(), PQsetenvAbort (Ewan Mellor)
New libpq PQsetenv() function (Ewan Mellor)
create/alter user extension (Peter E)
New postmaster.pid and postmaster.opts under $PGDATA (Tatsuo)
New scripts for create/drop user/db (Peter E)
Major psqi overhaul (Peter E)
Add const to libpq interface (Peter E)
New libpq function PQoidValue (Peter E)
Show specific non-aggregate causing problem with GROUP BY (Tom)
Make changes to pg_shadow recreate pg_pwd file (Peter E)
Add aggregate(DISTINCT ...) (Tom)
Allow flag to control COPY input/output of NULLs (Peter E)
Make postgres user have a password by default (Peter E)
Add CREATE/ALTER/DROP GROUP (Peter E)
All administration scripts now support --long options (Peter E, Karel)
Vacuumdb script now supports --all option (Peter E)
ecpg new portable FETCH syntax
Add ecpg EXEC SQL IFDEF, EXEC SQL IFNDEF, EXEC SQL ELSE, EXEC SQL ELIF
  and EXEC SQL ENDF directives
Add pg_ctl script to control backend start-up (Tatsuo)
Add postmaster.opts.default file to store start-up flags (Tatsuo)
Allow --with-mb=SQL_ASCII
Increase maximum number of index keys to 16 (Bruce)
Increase maximum number of function arguments to 16 (Bruce)
Allow configuration of maximum number of index keys and arguments (Bruce)
Allow unprivileged users to change their passwords (Peter E)
Password authentication enabled; required for new users (Peter E)
Disallow dropping a user who owns a database (Peter E)
Change initdb option --with-mb to --enable-multibyte
Add option for initdb to prompts for superuser password (Peter E)
Allow complex type casts like col::numeric(9,2) and col::int2::float8 (Tom)
Updated user interfaces on initdb, initlocation, pg_dump, ipcclean (Peter E)
New pg_char_to_encoding() and pg_encoding_to_char() functions (Tatsuo)
libpq non-blocking mode (Alfred Perlstein)
Appendix E. Release Notes

Improve conversion of types in casts that don't specify a length

New piperl internal programming language (Mark Hollomon)

Allow COPY IN to read file that do not end with a newline (Tom)

Indicate when long identifiers are truncated (Tom)

Allow aggregates to use type equivalency (Peter E)

Add Oracle's `to_char()`, `to_date()`, `to_datetime()`, `to_timestamp()`, `to_number()` conversion functions (Karel Zak <zakkr@zf.jcu.cz>)

Add SELECT DISTINCT ON (EXPR [, EXPR ...]) targetlist ... (Tom)

Check to be sure ORDER BY is compatible with the DISTINCT operation (Tom)

Add NUMERIC and int8 types to ODBC

Improve EXPLAIN results for Append, Group, Agg, Unique (Tom)

Add ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)

Allow SELECT .. FOR UPDATE in PL/pgSQL (Hiroshi)

Enable backward sequential scan even after reaching EOF (Hiroshi)

Add btree indexing of boolean values, > and <= (Don Baccus)

Print current line number when COPY FROM fails (Masimo)

Recognize POSIX time zone e.g. "PST+8" and "GMT-8" (Thomas)

Add DEC as synonym for DECIMAL (Thomas)

Add SESSION_USER as SQL92 key word, same as CURRENT_USER (Thomas)

Implement SQL92 column aliases (aka correlation names) (Thomas)

Implement SQL92 join syntax (Thomas)

Make INTERVAL reserved word allowed as a column identifier (Thomas)

Implement REINDEX command (Hiroshi)

Accept ALL in aggregate function SUM(ALL col) (Tom)

Prevent GROUP BY from using column aliases (Tom)

New `psql "encoding option (Tatsu)`

Allow PQrequestCancel() to terminate when in waiting-for-lock state (Hiroshi)

Allow negation of a negative number in all cases

Add ecpg descriptors (Christof, Michael)

Allow CREATE VIEW v AS SELECT f1::char(8) FROM tbl

Allow casts with length, like foo::char(8)

New libpq functions PQsetClientEncoding(), PQclientEncoding() (Tatsu)

Add support for SJIS user defined characters (Tatsu)

Larger views/rules supported

Make libpq’s PQconndefaults() thread-safe (Tom)

Disable // as comment to be ANSI conforming, should use -- (Tom)

Allow column aliases on views CREATE VIEW name (collist)

Fixes for views with subqueries (Tom)

Allow UPDATE table SET fld = (SELECT ...) (Tom)

SET command options no longer require quotes

Update pgaccess to 0.98.6

New SET SEED command

New `pg_options.sample file`

New `SET FSYNC command (Massimo)`

Allow `pg_descriptions` when creating tables

Allow `pg_descriptions` when creating types, columns, and functions

Allow `psql \copy` to allow delimiters (Peter E)

Allow `psql` to print nulls as distinct from "" [null] (Peter E)

Types

Many array fixes (Tom)

Allow bare column names to be subscripted as arrays (Tom)

Improve type casting of int and float constants (Tom)

Cleanup for int8 inputs, range checking, and type conversion (Tom)

Fix for SELECT timespan('21:11:26':time) (Tom)

netmask('x.x.x.x/0') is 255.255.255.255 instead of 0.0.0.0 (Oleg Sharoiko)
Appendix E. Release Notes

Add btree index on NUMERIC (Jan)
Perl fix for large objects containing NULL characters (Douglas Thomson)
ODBC fix for large objects (free)
Fix indexing of cidr data type
Fix for Ethernet MAC addresses (macaddr type) comparisons
Fix for date/time types when overflows happened in computations (Tom)
Allow array on int8 (Peter E)
Fix for rounding/overflow of NUMERIC type, like NUMERIC(4,4) (Tom)
Allow NUMERIC arrays
Fix bugs in NUMERIC ceil() and floor() functions (Tom)
Make char_length()/octet_length including trailing blanks (Tom)
Made abstime/reltime use int4 instead of time_t (Peter E)
New ltext data type for compressed text fields
Revise code to handle coercion of int and float constants (Tom)
Start at new code to implement a BIT and BIT VARYING type (Adriaan Joubert)
NUMERIC now accepts scientific notation (Tom)
NUMERIC to int4 rounds (Tom)
Convert float4/8 to NUMERIC properly (Tom)
Allow type conversion with NUMERIC (Thomas)
Make ISO date style (2000-02-16 09:33) the default (Thomas)
Add NATIONAL CHAR [ VARYING ] (Thomas)
Allow NUMERIC round and trunc to accept negative scales (Tom)
New TIME WITH TIME ZONE type (Thomas)
Add MAX() / MIN() on time type (Thomas)
Add abs(), mod(), fac() for int8 (Thomas)
Rename functions to round(), sqrt(), cbrt(), pow() for float8 (Thomas)
Add transcendental math functions (e.g. sin(), acos()) for float8 (Thomas)
Add exp() and ln() for NUMERIC type
Rename NUMERIC power() to pow() (Thomas)
Improved TRANSLATE() function (Edwin Ramirez, Tom)
Allow X=-Y operators (Tom)
Allow SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP BY f1; (Tom)
Allow LOCALE to use indexes in regular expression searches (Tom)
Allow creation of functional indexes to use default types

Performance
-----------
Prevent exponential space consumption with many AND’s and OR’s (Tom)
Collect attribute selectivity values for system columns (Tom)
Reduce memory usage of aggregates (Tom)
Fix for LIKE optimization to use indexes with multibyte encodings (Tom)
Fix r-tree index optimizer selectivity (Thomas)
Improve optimizer selectivity computations and functions (Tom)
Optimize btree searching for cases where many equal keys exist (Tom)
Enable fast LIKE index processing only if index present (Tom)
Re-use free space on index pages with duplicates (Tom)
Improve hash join processing (Tom)
Prevent descending sort if result is already sorted (Hiroshi)
Allow commuting of index scan query qualifications (Tom)
Prefer index scans in cases where ORDER BY/GROUP BY is required (Tom)
Allocate large memory requests in fix-sized chunks for performance (Tom)
Fix vacuum’s performance by reducing memory allocation requests (Tom)
Implement constant-expression simplification (Bernard Frankpitt, Tom)
Use secondary columns to be used to determine start of index scan (Hiroshi)
Prevent quadruple use of disk space when doing internal sorting (Tom)
Faster sorting by calling fewer functions (Tom)
Create system indexes to match all system caches (Bruce, Hiroshi)
Appendix E. Release Notes

Make system caches use system indexes (Bruce)
Make all system indexes unique (Bruce)
Improve pg_statistics management for VACUUM speed improvement (Tom)
Flush backend cache less frequently (Tom, Hiroshi)
COPY now reuses previous memory allocation, improving performance (Tom)
Improve optimization cost estimation (Tom)
Improve optimizer estimate of range queries \( x > \) lowbound AND \( x < \) highbound (Tom)
Use DNF instead of CNF where appropriate (Tom, Taral)
Further cleanup for OR-of-AND WHERE-clauses (Tom)
Make use of index in OR clauses \( (x = 1 \text{ AND } y = 2) \text{ OR } (x = 2 \text{ AND } y = 4) \) (Tom)
Smarter optimizer computations for random index page access (Tom)
New SET variable to control optimizer costs (Tom)
Optimizer queries based on LIMIT, OFFSET, and EXISTS qualifications (Tom)
Reduce optimizer internal housekeeping of join paths for speedup (Tom)
Major subquery speedup (Tom)
Fewer fsync writes when fsync is not disabled (Tom)
Improved LIKE optimizer estimates (Tom)
Prevent fsync in SELECT-only queries (Vadim)
Make index creation use psort code, because it is now faster (Tom)
Allow creation of sort temp tables > 1 Gig

Source Tree Changes
-------------------
Fix for linux PPC compile
New generic expression-tree-walker subroutine (Tom)
Change \texttt{form()} to \texttt{varargform()} to prevent portability problems
Improved range checking for large integers on Alphas
Clean up \texttt{#include} in \texttt{/include} directory (Bruce)
Add scripts for checking includes (Bruce)
Remove un-needed \texttt{#include}’s from *.c files (Bruce)
Change \texttt{#include}’s to use \texttt{<>} and \texttt{"} as appropriate (Bruce)
Enable Windows compilation of libpq
Alpha spinlock fix from Uncle George <gatgul@voicenet.com>
Overhaul of optimizer data structures (Tom)
Fix to cygipc library (Yutaka Tanida)
Allow pgsql to work on newer Cygwin snapshots (Dan)
New catalog version number (Tom)
Add Linux ARM
Rename heap\_replace to heap\_update
Update for QNX (Dr. Andreas Kardos)
New platform-specific regression handling (Tom)
Rename oid8 -> oidvector and int28 -> int2vector (Bruce)
Included all yacc and lex files into the distribution (Peter E.)
Remove lextest, no longer needed (Peter E)
Fix for libpq and psql on Windows (Magnus)
Internally change datetime and timespan into timestamp and interval (Thomas)
Fix for plpgsql on BSD/OS
Add SQL\_ASCII test case to the regression test (Tatsuo)
configure --with-mb now deprecated (Tatsuo)
NT fixes
NetBSD fixes (Johnny C. Lam <lamj@stat.cmu.edu>)
Fixes for Alpha compiles
New multibyte encodings

1759
Appendix E. Release Notes

E.170. Release 6.5.3

Release date: 1999-10-13

This is basically a cleanup release for 6.5.2. We have added a new PgAccess that was missing in 6.5.2, and installed an NT-specific fix.

E.170.1. Migration to Version 6.5.3

A dump/restore is not required for those running 6.5.*.

E.170.2. Changes

- Updated version of pgaccess 0.98
- NT-specific patch
- Fix dumping rules on inherited tables

E.171. Release 6.5.2

Release date: 1999-09-15

This is basically a cleanup release for 6.5.1. We have fixed a variety of problems reported by 6.5.1 users.

E.171.1. Migration to Version 6.5.2

A dump/restore is not required for those running 6.5.*.

E.171.2. Changes

- subselect+CASE fixes (Tom)
- Add SHLIB_LINK setting for solaris_i386 and solaris_sparc ports (Daren Sefcik)
- Fixes for CASE in WHERE join clauses (Tom)
- Fix BTScan abort (Tom)
- Repair the check for redundant UNIQUE and PRIMARY KEY indexes (Thomas)
- Improve it so that it checks for multicolumn constraints (Thomas)
- Fix for Windows making problem with MB enabled (Hiroki Kataoka)
- Allow BSD yacc and bison to compile pl code (Bruce)
- Fix SET NAMES working
- int8 fixes (Thomas)
Appendix E. Release Notes

Fix vacuum’s memory consumption (Hiroshi, Tatsuo)
Reduce the total memory consumption of vacuum (Tom)
Fix for timestamp (datetime)
Rule deparsing bugfixes (Tom)
Fix quoting problems in mkMakefile.tcldefs.sh.in and mkMakefile.tkdefs.sh.in (Tom)
This is to re-use space on index pages freed by vacuum (Vadim)
document -x for pg_dump (Bruce)
Fix for unary operators in rule deparser (Tom)
Comment out FileUnlink of excess segments during mdtruncate() (Tom)
IRIX linking fix from Yu Cao <yucao@falcon.kla-tencor.com>
Repair logic error in LIKE: should not return LIKE_ABORT when reach end of pattern before end of text (Tom)
Repair incorrect cleanup of heap memory allocation during transaction abort (Tom)
Updated version of pgaccess 0.98

E.172. Release 6.5.1

Release date: 1999-07-15

This is basically a cleanup release for 6.5. We have fixed a variety of problems reported by 6.5 users.

E.172.1. Migration to Version 6.5.1

A dump/restore is not required for those running 6.5.

E.172.2. Changes

Add NT README file
Portability fixes for linux_ppc, IRIX, linux_alpha, OpenBSD, alpha
Remove QUERY_LIMIT, use SELECT...LIMIT
Fix for EXPLAIN on inheritance (Tom)
Patch to allow vacuum on multisegment tables (Hiroshi)
R-Tree optimizer selectivity fix (Tom)
ACL file descriptor leak fix (Atsushi Ogawa)
New expression subtree code (Tom)
Avoid disk writes for read-only transactions (Vadim)
Fix for removal of temp tables if last transaction was aborted (Bruce)
Fix to prevent too large row from being created (Bruce)
plpgsql fixes
Allow port numbers 32k - 64k (Bruce)
Add ^ precedence (Bruce)
Rename sort files called pg_temp to pg_sorttemp (Bruce)
Fix for microseconds in time values (Tom)
Tutorial source cleanup
New linux_m68k port
Fix for sorting of NULL’s in some cases (Tom)
Appendix E. Release Notes

Shared library dependencies fixed (Tom)
Fixed glitches affecting GROUP BY in subselects (Tom)
Fix some compiler warnings (Tomoaki Nishiyama)
Add Win1250 (Czech) support (Pavel Behal)

E.173. Release 6.5

Release date: 1999-06-09

This release marks a major step in the development team’s mastery of the source code we inherited from Berkeley. You will see we are now easily adding major features, thanks to the increasing size and experience of our world-wide development team.

Here is a brief summary of the more notable changes:

Multiversion concurrency control (MVCC)

This removes our old table-level locking, and replaces it with a locking system that is superior to most commercial database systems. In a traditional system, each row that is modified is locked until committed, preventing reads by other users. MVCC uses the natural multiversion nature of PostgreSQL to allow readers to continue reading consistent data during writer activity. Writers continue to use the compact pg_log transaction system. This is all performed without having to allocate a lock for every row like traditional database systems. So, basically, we no longer are restricted by simple table-level locking; we have something better than row-level locking.

Hot backups from pg_dump

pg_dump takes advantage of the new MVCC features to give a consistent database dump/backup while the database stays online and available for queries.

Numeric data type

We now have a true numeric data type, with user-specified precision.

Temporary tables

Temporary tables are guaranteed to have unique names within a database session, and are destroyed on session exit.

New SQL features

We now have CASE, INTERSECT, and EXCEPT statement support. We have new LIMIT/OFFSET, SET TRANSACTION ISOLATION LEVEL, SELECT ... FOR UPDATE, and an improved LOCK TABLE command.

Speedups

We continue to speed up PostgreSQL, thanks to the variety of talents within our team. We have sped up memory allocation, optimization, table joins, and row transfer routines.

Ports

We continue to expand our port list, this time including Windows NT/ix86 and NetBSD/arm32.
Appendix E. Release Notes

Interfaces

Most interfaces have new versions, and existing functionality has been improved.

Documentation

New and updated material is present throughout the documentation. New FAQs have been contributed for SGI and AIX platforms. The Tutorial has introductory information on SQL from Stefan Simkovics. For the User’s Guide, there are reference pages covering the postmaster and more utility programs, and a new appendix contains details on date/time behavior. The Administrator’s Guide has a new chapter on troubleshooting from Tom Lane. And the Programmer’s Guide has a description of query processing, also from Stefan, and details on obtaining the PostgreSQL source tree via anonymous CVS and CVSup.

E.173.1. Migration to Version 6.5

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of PostgreSQL. pg_upgrade can not be used to upgrade to this release because the on-disk structure of the tables has changed compared to previous releases.

The new Multiversion Concurrency Control (MVCC) features can give somewhat different behaviors in multiuser environments. Read and understand the following section to ensure that your existing applications will give you the behavior you need.

E.173.1.1. Multiversion Concurrency Control

Because readers in 6.5 don’t lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another. In other words, if a row is returned by SELECT it doesn’t mean that this row really exists at the time it is returned (i.e. sometime after the statement or transaction began) nor that the row is protected from being deleted or updated by concurrent transactions before the current transaction does a commit or rollback.

To ensure the actual existence of a row and protect it against concurrent updates one must use SELECT FOR UPDATE or an appropriate LOCK TABLE statement. This should be taken into account when porting applications from previous releases of PostgreSQL and other environments.

Keep the above in mind if you are using contrib/refint.* triggers for referential integrity. Additional techniques are required now. One way is to use LOCK parent_table IN SHARE ROW EXCLUSIVE MODE command if a transaction is going to update/delete a primary key and use LOCK parent_table IN SHARE MODE command if a transaction is going to update/insert a foreign key.

Note: Note that if you run a transaction in SERIALIZE mode then you must execute the LOCK commands above before execution of any DML statement (SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO) in the transaction.

These inconveniences will disappear in the future when the ability to read dirty (uncommitted) data (regardless of isolation level) and true referential integrity will be implemented.
E.173.2. Changes

Bug Fixes

Fix text <-> float8 and text <-> float4 conversion functions (Thomas)
Fix for creating tables with mixed-case constraints (Billy)
Change exp()/pow() behavior to generate error on underflow/overflow (Jan)
Fix bug in pg_dumps -z
Memory overrun cleanups (Tatsuo)
Fix for lo_import crash (Tatsuo)
Adjust handling of data type names to suppress double quotes (Thomas)
Use type coercion for matching columns and DEFAULT (Thomas)
Fix deadlock so it only checks once after one second of sleep (Bruce)
Fixes for aggregates and PL/pgsql (Hiroshi)
Fix for subquery crash (Vadim)
Fix for libpq function PQfnumber and case-insensitive names (Bahman Rafatjoo)
Fix for large object write-in-middle, no extra block, memory consumption (Tatsuo)
Fix for pg_dumps -d or -D and quote special characters in INSERT
Repair serious problems with dynahash (Tom)
Fix INET/CIDR portability problems
Fix problem with selectivity error in ALTER TABLE ADD COLUMN (Bruce)
Fix executor so mergejoin of different column types works (Tom)
Fix for Alpha OR selectivity bug
Fix OR index selectivity problem (Bruce)
Fix so \d shows proper length for char()/varchar() (Ryan)
Fix tutorial code (Clark)
Improve destroyuser checking (Oliver)
Fix for Kerberos (Rodney McDuff)
Fix for dropping database while dirty buffers (Bruce)
Fix so sequence nextval() can be case-sensitive (Bruce)
Fix !!= operator
Drop buffers before destroying database files (Bruce)
Fix case where executor evaluates functions twice (Tatsuo)
Allow sequence nextval actions to be case-sensitive (Bruce)
Fix optimizer indexing not working for negative numbers (Bruce)
Fix for memory leak in executor with fjisNull
Fix for aggregate memory leaks (Erik Riedel)
Allow user name containing a dash to grant privileges
Cleanup of NULL in inet types
Clean up system table bugs (Tom)
Fix problems of PAGER and \? command (Masaaki Sakaida)
Reduce default multisegment file size limit to 1GB (Peter)
Fix for dumping of CREATE OPERATOR (Tom)
Fix for backward scanning of cursors (Hiroshi Inoue)
Fix for COPY FROM STDIN when using \i (Tom)
Fix for subselect is compared inside an expression (Jan)
Fix handling of error reporting while returning rows (Tom)
Fix problems with reference to array types (Tom, Jan)
Prevent UPDATE SET old (Jan)
Fix pg_dumps so -t option can handle case-sensitive tablenames
Fixes for GROUP BY in special cases (Tom, Jan)
Fix for memory leak in failed queries (Tom)
DEFAULT now supports mixed-case identifiers (Tom)
Fix for multisegment uses of DROP/RENAME table, indexes (Ole Gjerde)
Disable use of pg_dumps with both -o and -d options (Bruce)
Allow pg_dumps to properly dump group privileges (Bruce)
Fix GROUP BY in INSERT INTO table SELECT * FROM table2 (Jan)
Appendix E. Release Notes

Fix for computations in views (Jan)
Fix for aggregates on array indexes (Tom)
Fix for DEFAULT handles single quotes in value requiring too many quotes
Fix security problem with non-super users importing/exporting large objects (Tom)
Rollback of transaction that creates table cleaned up properly (Tom)
Fix to allow long table and column names to generate proper serial names (Tom)

Enhancements
-------------
Add "vacuumdb" utility
Speed up libpq by allocating memory better (Tom)
EXPLAIN all indexes used (Tom)
Implement CASE, COALESCE, NULLIF expression (Thomas)
New pg_dump table output format (Constantin)
Add string min()/max() functions (Thomas)
Extend new type coercion techniques to aggregates (Thomas)
New moddatetime contrib (Terry)
Update to pgaccess 0.96 (Constantin)
Add routines for single-byte "char" type (Thomas)
Improved substr() function (Thomas)
Improved multibyte handling (Tatsuo)
Multiversion concurrency control/MVCC (Vadim)
New Serialized mode (Vadim)
Fix for tables over 2gigs (Peter)
New SET TRANSACTION ISOLATION LEVEL (Vadim)
New LOCK TABLE IN ... MODE (Vadim)
Update ODBC driver (Byron)
New NUMERIC data type (Jan)
New SELECT FOR UPDATE (Vadim)
Handle "NaN" and "Infinity" for input values (Jan)
Improved date/year handling (Thomas)
Improved handling of backend connections (Magnus)
New options ELOG_TIMESTAMPS and USE_SYSLOG options for log files (Massimo)
New TCL_ARRAYS option (Massimo)
New INTERSECT and EXCEPT (Stefan)
New pg_index.indisprimary for primary key tracking (D’Arcy)
New pg_dump option to allow dropping of tables before creation (Brook)
Speedup of row output routines (Tom)
New READ COMMITTED isolation level (Vadim)
New TEMP tables/indexes (Bruce)
Prevent sorting if result is already sorted (Jan)
New memory allocation optimization (Jan)
Allow psql to do \p\g (Bruce)
Allow multiple rule actions (Jan)
Added LIMIT/OFFSET functionality (Jan)
Improve optimizer when joining a large number of tables (Bruce)
New intro to SQL from S. Simkovics’ Master’s Thesis (Stefan, Thomas)
New intro to backend processing from S. Simkovics’ Master’s Thesis (Stefan)
Improved int8 support (Ryan Bradetich, Thomas, Tom)
New routines to convert between int8 and text/varchar types (Thomas)
New bushy plans, where meta-tables are joined (Bruce)
Enable right-hand queries by default (Bruce)
Allow reliable maximum number of backends to be set at configure time
   (--with-maxbackends and postmaster switch --N backends) (Tom)
GEQO default now 10 tables because of optimizer speedups (Tom)
Allow NULL-Var for MS-SQL portability (Michael, Bruce)
Modify contrib check_primary_key() so either "automatic" or "dependent" (Anand)
Appendix E. Release Notes

Allow psql \d on a view show query(Ryan)
Speedup for LIKE(Bruce)
Ecpg fixes/features, see src/interfaces/ecpg/ChangeLog file(Michael)
JDBC fixes/features, see src/interfaces/jdbc/CHANGELOG(Peter)
Make % operator have precedence like /(Bruce)
Add new postgres -O option to allow system table structure changes(Bruce)
Update contrib/pginterface/findoidjoins script(Tom)
Major speedup in vacuum of deleted rows with indexes(Vadim)
Allow non-SQL functions to run different versions based on arguments(Tom)
Add -E option that shows actual queries sent by \dt and friends(Masaaki Sakaida)
Add version number in start-up banners for psql(Masaaki Sakaida)
New contrib/vacuumlo removes large objects not referenced(Peter)
New initialization for table sizes so non-vacuumed tables perform better(Tom)
Improve error messages when a connection is rejected(Tom)
Support for arrays of char() and varchar() fields(Massimo)
Overhaul of hash code to increase reliability and performance(Tom)
Update to PyGreSQL 2.4(D’Arcy)
Changed debug options so -d4 and -d5 produce different node displays(Jan)
New pg_options: pretty_plan, pretty_parse, pretty_rewritten(Jan)
Better optimization statistics for system table access(Tom)
Better handling of non-default block sizes(Massimo)
Improve GEOQ optimizer memory consumption(Tom)
UNION now supports ORDER BY of columns not in target list(Jan)
Major libpq++ improvements(Vince Vielhaber)
pg_dump now uses -z(ACL’s) as default(Bruce)
backend cache, memory speedups(Tom)
have pg_dump do everything in one snapshot transaction(Vadim)
fix for large object memory leakage, fix for pg_dumping(Tom)
INET type now respects netmask for comparisons
Make VACUUM ANALYZE only use a readlock(Vadim)
Allow VIEWS on UNIONS(Jan)
pq_dump now can generate consistent snapshots on active databases(Vadim)

Source Tree Changes
-------------------
Improve port matching(Tom)
Portability fixes for SunOS
Add Windows NT backend port and enable dynamic loading(Magnus and Daniel Horak)
New port to Cobalt Qube(Mips) running Linux(Tatsuo)
Port to NetBSD/m68k(Mr. Mutsuki Nakajima)
Port to NetBSD/sun3(Mr. Mutsuki Nakajima)
Port to NetBSD/macppc(Toshimi Aoki)
Fix for tcl/tk configuration(Vince)
Removed CURRENT key word for rule queries(Jan)
NT dynamic loading now works(Daniel Horak)
Add ARM32 support(Andrew McMurry)
Better support for HP-UX 11 and UnixWare
Improve file handling to be more uniform, prevent file descriptor leak(Tom)
New install commands for plpgsql(Jan)
Appendix E. Release Notes

E.174. Release 6.4.2

Release date: 1998-12-20

The 6.4.1 release was improperly packaged. This also has one additional bug fix.

E.174.1. Migration to Version 6.4.2

A dump/restore is not required for those running 6.4.*.

E.174.2. Changes

Fix for datetime constant problem on some platforms (Thomas)

E.175. Release 6.4.1

Release date: 1998-12-18

This is basically a cleanup release for 6.4. We have fixed a variety of problems reported by 6.4 users.

E.175.1. Migration to Version 6.4.1

A dump/restore is not required for those running 6.4.

E.175.2. Changes

Add pg_dump -N flag to force double quotes around identifiers. This is the default (Thomas)
Fix for NOT in where clause causing crash (Bruce)
EXPLAIN VERBOSE coredump fix (Vadim)
Fix shared-library problems on Linux
Fix test for table existence to allow mixed-case and whitespace in the table name (Thomas)
Fix a couple of pg_dump bugs
Configure matches template/.similar entries better (Tom)
Change built-in function names from SPI_* to spi_*
OR WHERE clause fix (Vadim)
Fixes for mixed-case table names (Billy)
contrib/linux/postgres.init.csh/sh fix (Thomas)
libpq memory overrun fix
SunOS fixes (Tom)
Appendix E. Release Notes

Change exp() behavior to generate error on underflow (Thomas)
pg_dump fixes for memory leak, inheritance constraints, layout change
update pgaccess to 0.93
Fix prototype for 64-bit platforms
Multibyte fixes (Tatsuo)
New ecpg man page
Fix memory overruns (Tatsuo)
Fix for lo_import() crash (Bruce)
Better search for install program (Tom)
Timezone fixes (Tom)
HP-UX fixes (Tom)
Use implicit type coercion for matching DEFAULT values (Thomas)
Add routines to help with single-byte (internal) character type (Thomas)
Compilation of libpq for Windows fixes (Magnus)
Upgrade to PyGreSQL 2.2 (D’Arcy)

E.176. Release 6.4

Release date: 1998-10-30

There are many new features and improvements in this release. Thanks to our developers and maintainers, nearly every aspect of the system has received some attention since the previous release. Here is a brief, incomplete summary:

- Views and rules are now functional thanks to extensive new code in the rewrite rules system from Jan Wieck. He also wrote a chapter on it for the Programmers Guide.
- Jan also contributed a second procedural language, PL/pgSQL, to go with the original PL/pgTCL procedural language he contributed last release.
- We have optional multiple-byte character set support from Tatsuo Ishii to complement our existing locale support.
- Client/server communications has been cleaned up, with better support for asynchronous messages and interrupts thanks to Tom Lane.
- The parser will now perform automatic type coercion to match arguments to available operators and functions, and to match columns and expressions with target columns. This uses a generic mechanism which supports the type extensibility features of PostgreSQL. There is a new chapter in the User’s Guide which covers this topic.
- Three new data types have been added. Two types, inet and cidr, support various forms of IP network, subnet, and machine addressing. There is now an 8-byte integer type available on some platforms. See the chapter on data types in the User’s Guide for details. A fourth type, serial, is now supported by the parser as an amalgam of the int4 type, a sequence, and a unique index.
- Several more SQL92-compatible syntax features have been added, including INSERT DEFAULT VALUES
The automatic configuration and installation system has received some attention, and should be more robust for more platforms than it has ever been.

### E.176.1. Migration to Version 6.4

A dump/restore using `pg_dump` or `pg_dumpall` is required for those wishing to migrate data from any previous release of PostgreSQL.

### E.176.2. Changes

#### Bug Fixes

- Fix for a tiny memory leak in PQsetdb/PQfinish (Bryan)
- Remove char2-16 data types, use char/varchar (Darren)
- PQfn not handles a NOTICE message (Anders)
- Reduced busywaiting overhead for spinlocks with many backends (dg)
- Stuck spinlock detection (dg)
- Fix up "ISO-style" timespan decoding and encoding (Thomas)
- Fix problem with table drop after rollback of transaction (Vadim)
- Change error message and remove non-functional update message (Vadim)
- Fix for COPY array checking
- Fix for SELECT 1 UNION SELECT NULL
- Fix for buffer leaks in large object calls (Pascal)
- Change owner from oid to int4 type (Bruce)
- Fix a bug in the oracle compatibility functions btrim() ltrim() and rtrim() (Massimo)
- Prevent file descriptor leaks in failed COPY’s (Bruce)
- Fix memory leak in libpqtcl’s pg_select (Constantin)
- Fix problems with username/passwords over 8 characters (Tom)
- Fix problems with handling of asynchronous NOTIFY in backend (Tom)
- Fix of many bad system table entries (Tom)

#### Enhancements

- Upgrade ecpg and ecpglib, see src/interfaces/ecpc/ChangeLog (Michael)
- Show the index used in an EXPLAIN (Zeugswetter)
- EXPLAIN invokes rule system and shows plan(s) for rewritten queries (Jan)
- Multibyte awareness of many data types and functions, via configure (Tatsuo)
- New configure --with-mb option (Tatsuo)
- New initdb --pgencoding option (Tatsuo)
- New createdb -E multibyte option (Tatsuo)
- Select version(); now returns PostgreSQL version (Jeroen)
- libpq now allows asynchronous clients (Tom)
- Allow cancel from client of backend query (Tom)
- psql now cancels query with Control-C (Tom)
- libpq users need not issue dummy queries to get NOTIFY messages (Tom)
- NOTIFY now sends sender’s PID, so you can tell whether it was your own (Tom)
- PGresult struct now includes associated error message, if any (Tom)
- Define "tz_hour" and "tz_minute" arguments to date_part() (Thomas)
- Add routines to convert between varchar and bpchar (Thomas)
- Add routines to allow sizing of varchar and bpchar into target columns (Thomas)
- Add bit flags to support timezonehour and minute in data retrieval (Thomas)
Appendix E. Release Notes

Allow more variations on valid floating point numbers (e.g. ".1", "1e6") (Thomas)
Fixes for unary minus parsing with leading spaces (Thomas)
Implement TIMEZONE_HOUR, TIMEZONE_MINUTE per SQL92 specs (Thomas)
Check for and properly ignore FOREIGN KEY column constraints (Thomas)
Define USER as synonym for CURRENT_USER per SQL92 specs (Thomas)
Enable HAVING clause but no fixes elsewhere yet.
Make "char" type a synonym for "char(1)" (actually implemented as bpchar) (Thomas)
Save string type if specified for DEFAULT clause handling (Thomas)
Coerce operations involving different data types (Thomas)
Allow some index use for columns of different types (Thomas)
Cleanups for large objects, so file is truncated on open (Peter)
Readline cleanups (Tom)
Allow psql \f \ to make spaces as delimiter (Bruce)
Pass pg_attribute.atttypmod to the frontend for column field lengths (Tom, Bruce)
Msql compatibility library in /contrib (Aldrin)
Remove the requirement that ORDER/GROUP BY clause identifiers be included in the target list (David)
Convert columns to match columns in UNION clauses (Thomas)
Remove fork() / exec() and only do fork() (Bruce)
Jdbc cleanups (Peter)
Show backend status on ps command line (only works on some platforms) (Bruce)
Pg_hba.conf now has a sameuser option in the database field
Make lo_unlink take oid param, not int
New DISABLE_COMPLEX_MACRO for compilers that cannot handle our macros (Bruce)
 Libpgtcl now handles NOTIFY as a Tcl event, need not send dummy queries (Tom)
 Libpgtcl cleanups (Tom)
Add -error option to libpgtcl  s pg_result command (Tom)
New locale patch, see docs/README/locale (Oleg)
Fix for pg_dump so CONSTRAINT and CHECK syntax is correct (ccb)
New contrib/lo code for large object orphan removal (Peter)
New psqi command "SET CLIENT_ENCODING TO 'encoding'" for multibytes feature, see /doc/README.mb (Tatsuo)
contrib/noupdate code to revoke update permission on a column
libpq can now be compiled on Windows (Magnus)
Add PQsetdbLogin() in libpq
New 8-byte integer type, checked by configure for OS support (Thomas)
Better support for quoted table/column names (Thomas)
Surround table and column names with double-quotes in pg_dump (Thomas)
PQreset() now works with passwords (Tom)
Handle case of GROUP BY target list column number out of range (David)
Allow UNION in subselects
Add auto-size to screen to \d? commands (Bruce)
Use UNION to show all \d? results in one query (Bruce)
Add \d? field search feature (Bruce)
Pg_dump issues fewer \connect requests (Tom)
Make pg_dump -z flag work better, document it in manual page (Tom)
Add HAVING clause with full support for subselects and unions (Stephan)
Full text indexing routines in contrib/fulltextindex (Maarten)
Transaction ids now stored in shared memory (Vadim)
New PGCLIENTENCODING when issuing COPY command (Tatsuo)
Support for SQL92 syntax "SET NAMES" (Tatsuo)
Support for LATIN2-5 (Tatsuo)
Add UNICODE regression test case (Tatsuo)
Lock manager cleanup, new locking modes for LLL (Vadim)
Allow index use with OR clauses (Bruce)
Allows "SELECT NULL ORDER BY 1;"
Appendix E. Release Notes

Explain VERBOSE prints the plan, and now pretty-prints the plan to the postmaster log file (Bruce).
Add indexes display to \d command (Bruce).
Allow GROUP BY on functions (David).
New pg_class.relkind for large objects (Bruce).
New way to send libpq NOTICE messages to a different location (Tom).
New \w write command to psql (Bruce).
New /contrib/findoidjoins scans old columns to find join relationships (Bruce).
Allow binary-compatible indexes to be considered when checking for valid indexes for restriction clauses containing a constant (Thomas).
Allow NOT LIKE, IN, NOT IN, BETWEEN, and NOT BETWEEN constraint (Thomas).
New rewrite system fixes many problems with rules and views (Jan).
  * Rules on relations work
  * Event qualifications on insert/update/delete work
  * New OLD variable to reference CURRENT, CURRENT will be remove in future
  * Update rules can reference NEW and OLD in rule qualifications/actions
  * Insert/update/delete rules on views work
  * Multiple rule actions are now supported, surrounded by parentheses
  * Regular users can create views/rules on tables they have RULE permits
  * Rules and views inherit the privileges of the creator
  * No rules at the column level
  * No UPDATE NEW/OLD rules
  * New pg_tables, pg_indexes, pg_rules and pg_views system views
  * Only a single action on SELECT rules
  * Total rewrite overhaul, perhaps for 6.5
  * handle subselects
  * handle aggregates on views
  * handle insert into select from view works

System indexes are now multikey (Bruce).
Oldint2, oldint4, and oldname types are removed (Bruce).
Use system cache for more system table lookups (Bruce).
New backend programming language PL/pgSQL in backend/pl (Jan).
New SERIAL data type, auto-creates sequence/index (Thomas).
Enable assert checking without a recompile (Massimo).
User lock enhancements (Massimo).
New setval() command to set sequence value (Massimo).
Auto-remove unix socket file on start-up if no postmaster running (Massimo).
Conditional trace package (Massimo).
New UNLISTEN command (Massimo).
psql and libpq now compile under Windows using win32.mak (Magnus).
Lo_read no longer stores trailing NULL (Bruce).
Identifiers are now truncated to 31 characters internally (Bruce).
Createuser options now available on the command line.
Code for 64-bit integer supported added, configure tested, int8 type (Thomas).
Prevent file descriptor leaf from failed COPY (Bruce).
New pg_upgrade command (Bruce).
Updated /contrib directories (Massimo).
New CREATE TABLE DEFAULT VALUES statement available (Thomas).
New INSERT INTO TABLE DEFAULT VALUES statement available (Thomas).
New DECLARE and FETCH feature (Thomas).
libpq’s internal structures now not exported (Tom).
Allow up to 8 key indexes (Bruce).
Remove ARCHIVE key word, that is no longer used (Thomas).
pg_dump -n flag to suppress quotes around indentifiers.
disable system columns for views (Jan).
New INET and CIDR types for network addresses (TomH, Paul).
Appendix E. Release Notes

no more double quotes in psql output
pg_dump now dumps views(Terry)
new SET QUERY_LIMIT(Tatsuo,Jan)

Source Tree Changes
-------------------
/contrib cleanup(Jun)
Inline some small functions called for every row(Bruce)
Alpha/linux fixes
HP-UX cleanups(Tom)
Multibyte regression tests(Soonmyung.)
Remove --disabled options from configure
Define PGDOC to use POSTGRESDIR by default
Make regression optional
Remove extra braces code to pgindent(Bruce)
Add bsd shared library support(Bruce)
New --without-CXX support configure option(Brook)
New FAQ_CVS
Update backend flowchart in tools/backend(Bruce)
Change atttypmod from int16 to int32(Bruce, Tom)
Getrusage() fix for platforms that do not have it(Tom)
Add PQconnectdb, PGUSER, PGPASSWORD to libpq man page
NS32K platform fixes(Phil Nelson, John Buller)
SCO 7/UnixWare 2.x fixes(Billy,others)
Sparc/Solaris 2.5 fixes(Ryan)
Pgbuiltin.3 is obsolete, move to doc files(Thomas)
Even more documentation(Thomas)
Nextstep support(Jacek)
Aix support(David)
pgresql interface manual page(Bruce)
shared libraries all have version numbers
merged all OS-specific shared library defines into one file
smarter TCL/TK configuration checking(Billy)
smarter perl configuration(Brook)
configure uses supplied install-sh if no install script found(Tom)
new Makefile.shlib for shared library configuration(Tom)

E.177. Release 6.3.2

Release date: 1998-04-07

This is a bug-fix release for 6.3.x. Refer to the release notes for version 6.3 for a more complete summary of new features.

Summary:

• Repairs automatic configuration support for some platforms, including Linux, from breakage inadvertently introduced in version 6.3.1.

• Correctly handles function calls on the left side of BETWEEN and LIKE clauses.
A dump/restore is NOT required for those running 6.3 or 6.3.1. A make distclean, make, and make install is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use PostgreSQL libraries.

For upgrades from pre-6.3 installations, refer to the installation and migration instructions for version 6.3.

### E.177.1. Changes

Configure detection improvements for tcl/tk (Brook Milligan, Alvin)
Manual page improvements (Bruce)
BETWEEN and LIKE fix (Thomas)
fix for psql \connect used by pg_dump (Oliver Elphick)
New odbc driver
pgaccess, version 0.86
qsort removed, now uses libc version, cleanups (Jeroen)
fix for buffer over-runs detected (Maurice Gittens)
fix for buffer overrun in libpgtcl (Randy Kunkee)
fix for UNION with DISTINCT or ORDER BY (Bruce)
gettimeofday configure check (Doug Winterburn)
Fix "indexes not used" bug (Vadim)
docs additions (Thomas)
Fix for backend memory leak (Bruce)
libreadline cleanup (Erwan MAS)
Remove DISTDIR (Bruce)
Makefile dependency cleanup (Jeroen van Vianen)
ASSERT fixes (Bruce)

### E.178. Release 6.3.1

**Release date:** 1998-03-23

**Summary:**

- Additional support for multibyte character sets.
- Repair byte ordering for mixed-endian clients and servers.
- Minor updates to allowed SQL syntax.
- Improvements to the configuration autodetection for installation.
A dump/restore is NOT required for those running 6.3. A make distclean, make, and make install is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use PostgreSQL libraries.

For upgrades from pre-6.3 installations, refer to the installation and migration instructions for version 6.3.

E.178.1. Changes

ecpg cleanup/fixes, now version 1.1 (Michael Meskes)
pg_user cleanup (Bruce)
large object fix for pg_dump and tclsh (alvin)
LIKE fix for multiple adjacent underscores
fix for redefining builtin functions (Thomas)
ultrix4 cleanup
upgrade to pg_access 0.83
updated CLUSTER manual page
multibyte character set support, see doc/README.mb (Tatsuo)
configure --with-pgport fix
pg_ident fix
big-endian fix for backend communications (Kataoka)
SUBSTR() and substring() fix (Jan)
several jdbc fixes (Peter)
libpgtcl improvements, see libptcl/README (Randy Kunkee)
Fix for "Datasize = 0" error (Vadim)
Prevent \do from wrapping (Bruce)
Remove duplicate Russian character set entries
Sunos4 cleanup
Allow optional TABLE key word in LOCK and SELECT INTO (Thomas)
CREATE SEQUENCE options to allow a negative integer (Thomas)
Add "PASSWORD" as an allowed column identifier (Thomas)
Add checks for UNION target fields (Bruce)
Fix Alpha port (Dwayne Bailey)
Fix for text arrays containing quotes (Doug Gibson)
Solaris compile fix (Albert Chin-A-Young)
Better identify tcl and tk libs and includes (Bruce)

E.179. Release 6.3

Release date: 1998-03-01

There are many new features and improvements in this release. Here is a brief, incomplete summary:
Appendix E. Release Notes

- Many new SQL features, including full SQL92 subselect capability (everything is here but target-list subselects).
- Support for client-side environment variables to specify time zone and date style.
- Socket interface for client/server connection. This is the default now so you might need to start postmaster with the \-i flag.
- Better password authorization mechanisms. Default table privileges have changed.
- Old-style time travel has been removed. Performance has been improved.

Note: Bruce Momjian wrote the following notes to introduce the new release.

There are some general 6.3 issues that I want to mention. These are only the big items that cannot be described in one sentence. A review of the detailed changes list is still needed.

First, we now have subselects. Now that we have them, I would like to mention that without subselects, SQL is a very limited language. Subselects are a major feature, and you should review your code for places where subselects provide a better solution for your queries. I think you will find that there are more uses for subselects than you might think. Vadim has put us on the big SQL map with subselects, and fully functional ones too. The only thing you cannot do with subselects is to use them in the target list.

Second, 6.3 uses Unix domain sockets rather than TCP/IP by default. To enable connections from other machines, you have to use the new postmaster -i option, and of course edit pg_hba.conf. Also, for this reason, the format of pg_hba.conf has changed.

Third, char() fields will now allow faster access than varchar() or text. Specifically, the text and varchar() have a penalty for access to any columns after the first column of this type. char() used to also have this access penalty, but it no longer does. This might suggest that you redesign some of your tables, especially if you have short character columns that you have defined as varchar() or text. This and other changes make 6.3 even faster than earlier releases.

We now have passwords definable independent of any Unix file. There are new SQL USER commands. See the Administrator’s Guide for more information. There is a new table, pg_shadow, which is used to store user information and user passwords, and it by default only SELECT-able by the postgres super-user. pg_user is now a view of pg_shadow, and is SELECT-able by PUBLIC. You should keep using pg_user in your application without changes.

User-created tables now no longer have SELECT privilege to PUBLIC by default. This was done because the ANSI standard requires it. You can of course GRANT any privileges you want after the table is created. System tables continue to be SELECT-able by PUBLIC.

We also have real deadlock detection code. No more sixty-second timeouts. And the new locking code implements a FIFO better, so there should be less resource starvation during heavy use.

Many complaints have been made about inadequate documentation in previous releases. Thomas has put much effort into many new manuals for this release. Check out the doc/ directory.

For performance reasons, time travel is gone, but can be implemented using triggers (see \psql\contrib\spi\README). Please check out the new \d command for types, operators, etc. Also, views have their own privileges now, not based on the underlying tables, so privileges on them have to be set separately. Check /pgsql/interfaces for some new ways to talk to PostgreSQL.
This is the first release that really required an explanation for existing users. In many ways, this was necessary because the new release removes many limitations, and the work-arounds people were using are no longer needed.

E.179.1. Migration to Version 6.3

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of PostgreSQL.

E.179.2. Changes

Bug Fixes
--------
Fix binary cursors broken by MOVE implementation (Vadim)
Fix for tcl library crash (Jan)
Fix for array handling, from Gerhard Hintermayer
Fix acl error, and remove duplicate pgtrace (Bruce)
Fix psql \e for empty file (Bruce)
Fix for textcat on varchar() fields (Bruce)
Fix for DBT Sendproc (Zeugwetter Andres)
Fix vacuum analyze syntax problem (Bruce)
Fix for international identifiers (Tatsu)
Fix for duplicate pgtrace (Bruce)
Fix substr() for out-of-bounds data
Fix for select 1=1 or 2=2, select 1=1 and 2=2, and select sum(2+2) (Bruce)
Fix notty output to show status result. -q option still turns it off (Bruce)
Fix for count(\*), aggs with views and multiple tables and sum(3) (Bruce)
Fix cluster (Bruce)
Fix for PQtrace start/stop several times (Bruce)
Fix a variety of locking problems like never lock waiters getting lock before older waiters, and having readlock people not share locks if a writer is waiting for a lock, and waiting writers not getting priority over waiting readers (Bruce)
Fix crashes in psql when executing queries from external files (James)
Fix problem with multiple order by columns, with the first one having NULL values (Jeroen)
Use correct hash table support functions for float8 and int4 (Thomas)
Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
Change precedence for boolean operators to match expected behavior (Thomas)
Generate elog(ERROR) on over-large integer (Bruce)
Allow multiple-argument functions in constraint clauses (Thomas)
Check boolean input literals for 'true', 'false', 'yes', 'no', '1', '0'
and throw elog(ERROR) if unrecognized (Thomas)
Major large objects fix
Fix for GROUP BY showing duplicates (Vadim)
Fix for index scans in MergeJoin (Vadim)

Enhancements
------------
Subselects with EXISTS, IN, ALL, ANY key words (Vadim, Bruce, Thomas)
New User Manual (Thomas, others)
Speedup by inlining some frequently-called functions
Real deadlock detection, no more timeouts (Bruce)
Add SQL92 "constants" CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP,
Appendix E. Release Notes

CURRENT_USER(Thomas)
Modify constraint syntax to be SQL92-compliant(Thomas)
Implement SQL92 PRIMARY KEY and UNIQUE clauses using indexes(Thomas)
Recognize SQL92 syntax for FOREIGN KEY. Throw elog notice(Thomas)
Allow NOT NULL UNIQUE constraint clause (each allowed separately before)(Thomas)
Allow PostgreSQL-style casting ("::") of non-constants(Thomas)
Add support for SQL3 TRUE and FALSE boolean constants(Thomas)
Support SQL92 syntax for IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE(Thomas)
Allow shorter strings for boolean literals (e.g. "t", "tr", "tru") (Thomas)
Allow SQL92 delimited identifiers(Thomas)
Implement SQL92 binary and hexadecimal string decoding (b'10' and x'1F')(Thomas)
Support SQL92 syntax for type coercion of literal strings
(e.g. "DATETIME 'now'") (Thomas)
Add conversions for int2, int4, and OID types to and from text(Thomas)
Use shared lock when building indexes(Vadim)
Free memory allocated for an user query inside transaction block after
this query is done, was turned off in <= 6.2.1(Vadim)
New SQL statement CREATE PROCEDURAL LANGUAGE(Jan)
New PostgreSQL Procedural Language (PL) backend interface(Jan)
Rename pg_dump -H option to -h(Bruce)
Add Java support for passwords, European dates(Peter)
Use indexes for LIKE and ~, !~ operations(Bruce)
Add hash functions for datetime and timespan(Thomas)
Time Travel removed(Vadim, Bruce)
Add paging for \d and \z, and fix \i(Bruce)
Add Unix domain socket support to backend and to frontend library(Goran)
Implement CREATE DATABASE/WITH LOCATION and initlocation utility(Thomas)
Allow more SQL92 and/or PostgreSQL reserved words as column identifiers(Thomas)
Augment support for SQL92 SET TIME ZONE...(Thomas)
SET/SHOW/RESET TIME ZONE uses TZ backend environment variable(Thomas)
Implement SET keyword = DEFAULT and SET TIME ZONE DEFAULT(Thomas)
Enable SET TIME ZONE using TZ environment variable(Thomas)
Add PGDATESTYLE environment variable to frontend and backend initialization(Thomas)
Add PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGSEQ
frontend library initialization environment variables(Thomas)
Regression tests time zone automatically set with "setenv PGTZ PST8PDT"(Thomas)
Add pg_description table for info on tables, columns, operators, types, and
aggregates(Bruce)
Increase 16 char limit on system table/index names to 32 characters(Bruce)
Rename system indexes(Bruce)
Add 'GERMAN' option to SET DATESTYLE(Thomas)
Define an "ISO-style" timespan output format with "hh:mm:ss" fields(Thomas)
Allow fractional values for delta times (e.g. '2.5 days')(Thomas)
Validate numeric input more carefully for delta times(Thomas)
Implement day of year as possible input to date_part() (Thomas)
Define timespan_finite() and text_timespan() functions(Thomas)
Remove archive stuff(Bruce)
Allow for a pg_password authentication database that is separate from
the system password file(Todd)
Dump ACLs, GRANT, REVOKE privileges(Matt)
Define text, varchar, and bpchar string length functions(Thomas)
Fix Query handling for inheritance, and cost computations(Bruce)
Implement CREATE TABLE/AS SELECT (alternative to SELECT/INTO)(Thomas)
Allow NOT, IS NULL, IS NOT NULL in constraints(Thomas)
Implement UNIONs for SELECT(Bruce)
Add UNION, GROUP, DISTINCT to INSERT(Bruce)
varchar() stores only necessary bytes on disk(Bruce)
Appendix E. Release Notes

Fix for BLOBs (Peter)
Mega-Patch for JDBC...see README_6.3 for list of changes (Peter)
Remove unused "option" from PQconnectdb()
New LOCK command and lock manual page describing deadlocks (Bruce)
Add new psql \da, \dd, \df, \do, \ds, and \dT commands (Bruce)
Enhance psql \z to show sequences (Bruce)
Show NOT NULL and DEFAULT in psql \d table (Bruce)
New psql .psqlrc file start-up (Andrew)
Modify sample start-up script in contrib/linux to show syslog (Thomas)
New types for IP and MAC addresses in contrib/ip_and_mac (TomH)
Unix system time conversions with date/time types in contrib/unixdate (Thomas)
Update of contrib stuff (Massimo)
Add Unix socket support to DBD::Pg (Goran)
New python interface (PyGreSQL 2.0) (D’Arcy)
New frontend/backend protocol has a version number, network byte order (Phil)
Security features in pg_hba.conf enhanced and documented, many cleanups (Phil)
CHAR() now faster access than VARCHAR() or TEXT
ecpg embedded SQL preprocessor
Reduce system column overhead (Vadmin)
Remove pg_time table (Vadmin)
Add pg_type attribute to identify types that need length (bpchar, varchar)
Add report of offending line when COPY command fails
Allow VIEW privileges to be set separately from the underlying tables.
   For security, use GRANT/REVOKE on views as appropriate (Jan)
Tables now have no default GRANT SELECT TO PUBLIC. You must
   explicitly grant such privileges.
Clean up tutorial examples (Darren)

Source Tree Changes
-------------------
Add new html development tools, and flow chart in /tools/backend
Fix for SCO compiles
Stratus computer port Robert Gillies
Added support for shlib for BSD44_derived & i386_solaris
Make configure more automated (Brook)
Add script to check regression test results
Break parser functions into smaller files, group together (Bruce)
Rename heap_create to heap_create_and_catalog, rename heap_create
   to heap_create() (Bruce)
Sparc/Linux patch for locking (TomS)
Remove PORTNAME and reorganize port-specific stuff (Marc)
Add optimizer README file (Bruce)
Remove some recursion in optimizer and clean up some code there (Bruce)
Fix for NetBSD locking (Henry)
Fix for libptcl make (Tatsuo)
AIX patch (Darren)
Change IS TRUE, IS FALSE, ... to expressions using "=" rather than
   function calls to istrue() or isfalse() to allow optimization (Thomas)
Various fixes NetBSD/Sparc related (TomH)
Alpha linux locking (Travis, Ryan)
Change elog(WARN) to elog(ERROR) (Bruce)
FAQ for FreeBSD (Marc)
Bring in the PostODBC source tree as part of our standard distribution (Marc)
A minor patch for HP/UX 10 vs 9 (Stan)
New pg_attribute.atttypmod for type-specific info like varchar length (Bruce)
UnixWare patches (Billy)
New i386 ‘lock’ for spinlock asm (Billy)
Appendix E. Release Notes

Support for multiplexed backends is removed
Start an OpenBSD port
Start an AUX port
Start a Cygnus port
Add string functions to regression suite (Thomas)
Expand a few function names formerly truncated to 16 characters (Thomas)
Remove un-needed malloc() calls and replace with palloc() (Bruce)

E.180. Release 6.2.1

Release date: 1997-10-17

6.2.1 is a bug-fix and usability release on 6.2.
Summary:

• Allow strings to span lines, per SQL92.
• Include example trigger function for inserting user names on table updates.

This is a minor bug-fix release on 6.2. For upgrades from pre-6.2 systems, a full dump/reload is required. Refer to the 6.2 release notes for instructions.

E.180.1. Migration from version 6.2 to version 6.2.1

This is a minor bug-fix release. A dump/reload is not required from version 6.2, but is required from any release prior to 6.2.

In upgrading from version 6.2, if you choose to dump/reload you will find that \texttt{avg(money)} is now calculated correctly. All other bug fixes take effect upon updating the executables.

Another way to avoid dump/reload is to use the following SQL command from \texttt{psql} to update the existing system table:

\begin{verbatim}
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
where aggname = 'avg' and aggbasetype = 790;
\end{verbatim}

This will need to be done to every existing database, including template1.

E.180.2. Changes

Allow \texttt{TIME} and \texttt{TYPE} column names (Thomas)
Allow larger range of true/false as boolean values (Thomas)
Support output of "now" and "current" (Thomas)
Handle \texttt{DEFAULT} with \texttt{INSERT} of NULL properly (Vadim)
Appendix E. Release Notes

Fix for relation reference counts problem in buffer manager (Vadim)
Allow strings to span lines, like ANSI (Thomas)
Fix for backward cursor with ORDER BY (Vadim)
Fix avg(cash) computation (Thomas)
Fix for specifying a column twice in ORDER/GROUP BY (Vadim)
Documented new libpq function to return affected rows, PQcmdTuples (Bruce)
Trigger function for inserting user names for INSERT/UPDATE (Brook Milligan)

E.181. Release 6.2

Release date: 1997-10-02

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

E.181.1. Migration from version 6.1 to version 6.2

This migration requires a complete dump of the 6.1 database and a restore of the database in 6.2.
Note that the pg_dump and pg_dumpall utility from 6.2 should be used to dump the 6.1 database.

E.181.2. Migration from version 1.* to version 6.2

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

E.181.3. Changes

Bug Fixes
--------
Fix problems with pg_dump for inheritance, sequences, archive tables (Bruce)
Fix compile errors on overflow due to shifts, unsigned, and bad prototypes from Solaris (Diab Jerius)
Fix bugs in geometric line arithmetic (bad intersection calculations) (Thomas)
Check for geometric intersections at endpoints to avoid rounding ugliness (Thomas)
Catch non-functional delete attempts (Vadim)
Change time function names to be more consistent (Michael Reifenberg)
Check for zero divides (Michael Reifenberg)
Fix very old bug which made rows changed/inserted by a command visible to the command itself (so we had multiple update of updated rows, etc.) (Vadim)
Fix for SELECT null, ‘fail’ FROM pg_am (Patrick)
SELECT NULL as EMPTY_FIELD now allowed (Patrick)
Remove un-needed signal stuff from contrib/pginterface
Fix OR (where x != 1 or x isnull didn’t return rows with x NULL) (Vadim)
Appendix E. Release Notes

Fix time_cmp function (Vadim)
Fix handling of functions with non-attribute first argument in WHERE clauses (Vadim)
Fix GROUP BY when order of entries is different from order in target list (Vadim)
Fix pg_dump for aggregates without sfunc1 (Vadim)

Enhancements
------------
Default genetic optimizer GEQO parameter is now 8 (Bruce)
Allow use parameters in target list having aggregates in functions (Vadim)
Added JDBC driver as an interface (Adrian & Peter)
pg_password utility
Return number of rows inserted/affected by INSERT/UPDATE/DELETE etc. (Vadim)
Triggers implemented with CREATE TRIGGER (SQL3) (Vadim)
SPI (Server Programming Interface) allows execution of queries inside C-functions (Vadim)
NOT NULL implemented (SQL92) (Robson Paniago de Miranda)
Include reserved words for string handling, outer joins, and unions (Thomas)
Implement extended comments ("/* ... */") using exclusive states (Thomas)
Add "/" single-line comments (Bruce)
Remove some restrictions on characters in operator names (Thomas)
DEFAULT and CONSTRAINT for tables implemented (SQL92) (Vadim & Thomas)
Add text concatenation operator and function (SQL92) (Thomas)
Support WITH TIME ZONE syntax (SQL92) (Thomas)
Support INTERVAL unit TO unit syntax (SQL92) (Thomas)
Define types DOUBLE PRECISION, INTERVAL, CHARACTER, and CHARACTER VARYING (SQL92) (Thomas)
Define type FLOAT(p) and rudimentary DECIMAL(p,s), NUMERIC(p,s) (SQL92) (Thomas)
Define EXTRACT(), POSITION(), SUBSTRING(), and TRIM() (SQL92) (Thomas)
Define CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92) (Thomas)
Support UNION syntax and warnings for UNION, HAVING, INNER and OUTER JOIN (SQL92) (Thomas)
Allow more reserved words, mostly for SQL92 compliance (Thomas)
Add syntax for time entry for timespan/reltime types (Thomas)
Add center() routines for lseg, path, polygon (Thomas)
Add distance() routines for circle-polygon, polygon-polygon (Thomas)
Check explicitly for points and polygons contained within polygons using an axis-crossing algorithm (Thomas)
Add routine to convert circle-box (Thomas)
Merge conflicting operators for different geometric data types (Thomas)
Replace distance operator "<==" with "<->" (Thomas)
Replace "above" operator "!`^" with "^" and "below" operator "!|" with "<^" (Thomas)
Add routines for text trimming on both ends, substring, and string position (Thomas)
Added conversion routines circle(box) and poly(circle) (Thomas)
Allow internal sorts to be stored in memory rather than in files (Bruce & Vadim)
Allow functions and operators on internally-identical types to succeed (Bruce)
Speed up backend start-up after profiling analysis (Bruce)
Inline frequently called functions for performance (Bruce)
Reduce open() calls (Bruce)
psql: Add PAGER for \h and \?,\C fix
Fix for psql pager when no tty (Bruce)
New entab utility (Bruce)
General trigger functions for referential integrity (Vadim)
General trigger functions for time travel (Vadim)
General trigger functions for AUTOINCREMENT/IDENTITY feature (Vadim)
MOVE implementation (Vadim)

1781
Appendix E. Release Notes

Source Tree Changes
-------------------
HP-UX 10 patches (Vladimir Turin)
Added SCO support, (Daniel Harris)
MkLinux patches (Tatsuo Ishii)
Change geometric box terminology from "length" to "width"(Thomas)
Deprecate temporary unstored slope fields in geometric code(Thomas)
Remove restart instructions from INSTALL(Bruce)
Look in /usr/ucb first for install(Bruce)
Fix c++ copy example code(Thomas)
Add -o to psql manual page(Bruce)
Prevent rename unallocated string length from being copied into database(Bruce)
Cleanup for NAMEDATALEN use(Bruce)
Fix pg_proc names over 15 chars in output(Bruce)
Add strNcpy() function(Bruce)
remove some (void) casts that are unnecessary(Bruce)
new interfaces directory(Marc)
Replace fopen() calls with calls to fd.c functions(Bruce)
Make functions static where possible(Bruce)
enclose unused functions in #ifdef NOT_USED(Bruce)
Remove call to difftime() in timestamp support to fix SunOS(Bruce & Thomas)
Changes for Digital Unix
Portability fix for pg_dumpall(Bruce)
Rename pg_attribute.attnvals to attdispersion(Bruce)
"intro/unix" manual page now "pgintro"(Bruce)
"built-in" manual page now "pgbuiltin"(Bruce)
"drop" manual page now "drop_table"(Bruce)
Add "create_trigger", "drop_trigger" manual pages(Thomas)
Add constraints regression test(Vadim & Thomas)
Add comments syntax regression test(Thomas)
Add PGINDENT and support program(Bruce)
Massive commit to run PGINDENT on all *.c and *.h files(Bruce)
Files moved to /src/tools directory(Bruce)
SPI and Trigger programming guides (Vadim & D’Arcy)

E.182. Release 6.1.1

Release date: 1997-07-22

E.182.1. Migration from version 6.1 to version 6.1.1

This is a minor bug-fix release. A dump/reload is not required from version 6.1, but is required from any release prior to 6.1. Refer to the release notes for 6.1 for more details.
E.182.2. Changes

- fix for SET with options (Thomas)
- allow pg_dump/pg_dumpall to preserve ownership of all tables/objects (Bruce)
- new psql \connect option allows changing usernames without changing databases
- fix for initdb --debug option (Yoshihiko Ichikawa)
- littest cleanup (Bruce)
- hash fixes (Vadim)
- fix date/time month boundary arithmetic (Thomas)
- fix timezone daylight handling for some ports (Thomas, Bruce, Tatsuo)
- timestamp overhauled to use standard functions (Thomas)
- other code cleanup in date/time routines (Thomas)
- psql’s \d now case-insensitive (Bruce)
- psql’s backslash commands can now have trailing semicolon (Bruce)
- fix memory leak in psql when using \g (Bruce)
- major fix for endian handling of communication to server (Thomas, Tatsuo)
- Fix for Solaris assembler and include files (Yoshihiko Ichikawa)
- allow underscores in usernames (Bruce)
- pg_dumpall now returns proper status, portability fix (Bruce)

E.183. Release 6.1

Release date: 1997-06-08

The regression tests have been adapted and extensively modified for the 6.1 release of PostgreSQL.

Three new data types (datetime, timespan, and circle) have been added to the native set of PostgreSQL types. Points, boxes, paths, and polygons have had their output formats made consistent across the data types. The polygon output in misc.out has only been spot-checked for correctness relative to the original regression output.

PostgreSQL 6.1 introduces a new, alternate optimizer which uses genetic algorithms. These algorithms introduce a random behavior in the ordering of query results when the query contains multiple qualifiers or multiple tables (giving the optimizer a choice on order of evaluation). Several regression tests have been modified to explicitly order the results, and hence are insensitive to optimizer choices. A few regression tests are for data types which are inherently unordered (e.g., points and time intervals) and tests involving those types are explicitly bracketed with set geqo to 'off' and reset geqo.

The interpretation of array specifiers (the curly braces around atomic values) appears to have changed sometime after the original regression tests were generated. The current ./expected/*.out files reflect this new interpretation, which might not be correct!

The float8 regression test fails on at least some platforms. This is due to differences in implementations of pow() and exp() and the signaling mechanisms used for overflow and underflow conditions.

The “random” results in the random test should cause the “random” test to be “failed”, since the regression tests are evaluated using a simple diff. However, “random” does not seem to produce random results on my test machine (Linux/gcc/i686).
Appendix E. Release Notes

E.183.1. Migration to Version 6.1

This migration requires a complete dump of the 6.0 database and a restore of the database in 6.1. Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

E.183.2. Changes

Bug Fixes
---------
packet length checking in library routines
lock manager priority patch
check for under/over flow of float8(Bruce)
multitable join fix(Vadim)
SIGPIPE crash fix(Darren)
large object fixes(Sven)
allow btree indexes to handle NULLs(Vadim)
timezone fixes(D’Arcy)
select SUM(x) can return NULL on no rows(Thomas)
internal optimizer, executor bug fixes(Vadim)
fix problem where inner loop in < or <= has no rows(Vadim)
prevent re-commuting join index clauses(Vadim)
fix join clauses for multiple tables(Vadim)
fix hash, hashjoin for arrays(Vadim)
fix btree for abstime type(Vadim)
large object fixes(Raymond)
fix buffer leak in hash indexes (Vadim)
fix rtree for use in inner scan (Vadim)
fix gist for use in inner scan, cleanups (Vadim, Andrea)
avoid unnecessary local buffers allocation (Vadim, Massimo)
fix local buffers leak in transaction aborts (Vadim)
fix file manager memmory leaks, cleanups (Vadim, Massimo)
fix storage manager memmory leaks (Vadim)
fix btree duplicates handling (Vadim)
fix deleted rows reincarnation caused by vacuum (Vadim)
fix SELECT varchar()/char() INTO TABLE made zero-length fields(Bruce)
many psql, pg_dump, and libpq memory leaks fixed using Purify (Igor)

Enhancements
------------
attribute optimization statistics(Bruce)
much faster new btree bulk load code(Paul)
BTREE UNIQUE added to bulk load code(Vadim)
new lock debug code(Massimo)
massive changes to libpq++(Leo)
new GEQO optimizer speeds table multitable optimization(Martin)
new WARN message for non-unique insert into unique key(Marc)
update x=-3, no spaces, now valid(Bruce)
remove case-sensitive identifier handling(Bruce, Thomas, Dan)
debug backend now pretty-prints tree(Darren)
new Oracle character functions(Edmund)
new plaintext password functions(Dan)
no such class or insufficient privilege changed to distinct messages(Dan)
new ANSI timestamp function(Dan)
new ANSI Time and Date types (Thomas)
move large chunks of data in backend (Martin)
multicolumn btree indexes (Vadim)
new SET var TO value command (Martin)
update transaction status on reads (Dan)
new locale settings for character types (Oleg)
new SEQUENCE serial number generator (Vadim)
GROUP BY function now possible (Vadim)
re-organize regression test (Thomas, Marc)
new optimizer operation weights (Vadim)
new psql \z grant/permit option (Marc)
new MONEY data type (D’Arcy, Thomas)
tcp socket communication speed improved (Vadim)
new VACUUM option for attribute statistics, and for certain columns (Vadim)
many geometric type improvements (Thomas, Keith)
additional regression tests (Thomas)
new datestyle variable (Thomas, Vadim, Martin)
more comparison operators for sorting types (Thomas)
new conversion functions (Thomas)
new more compact btree format (Vadim)
allow pg_dumpall to preserve database ownership (Bruce)
new SET GEQO=# and R_PLANS variable (Vadim)
old (!GEQO) optimizer can use right-sided plans (Vadim)
typechecking improvement in SQL parser (Bruce)
new SET, SHOW, RESET commands (Thomas, Vadim)
new \connect database USER option
new destroydb -i option (Igor)
new \dt and \di psql commands (Darren)
SELECT "\n" now escapes newline (A. Duursma)
new geometry conversion functions from old format (Thomas)

Source tree changes
-------------------
new configuration script (Marc)
readline configuration option added (Marc)
OS-specific configuration options removed (Marc)
new OS-specific template files (Marc)
no more need to edit Makefile.global (Marc)
re-arrange include files (Marc)
nextstep patches (Gregor Hoffleit)
removed Windows-specific code (Bruce)
removed postmaster -e option, now only postgres -e option (Bruce)
merge duplicate library code in front/backends (Martin)
now works with eBones, international Kerberos (Jun)
more shared library support
C++ include file cleanup (Bruce)
warn about buggy flex (Bruce)
DG/UX, Ultrix, IRIX, AIX portability fixes
**E.184. Release 6.0**

**Release date:** 1997-01-29

A dump/restore is required for those wishing to migrate data from previous releases of PostgreSQL.

**E.184.1. Migration from version 1.09 to version 6.0**

This migration requires a complete dump of the 1.09 database and a restore of the database in 6.0.

**E.184.2. Migration from pre-1.09 to version 6.0**

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

**E.184.3. Changes**

**Bug Fixes**

---

ALTER TABLE bug - running postgres process needs to re-read table definition
Allow vacuum to be run on one table or entire database (Bruce)
Array fixes
Fix array over-runs of memory writes (Kurt)
Fix elusive btree range/non-range bug (Dan)
Fix for hash indexes on some types like time and date
Fix for pg_log size explosion
Fix permissions on lo_export() (Bruce)
Fix uninitialized reads of memory (Kurt)
Fixed ALTER TABLE ... char(3) bug (Bruce)
Fixed a few small memory leaks
Fixed EXPLAIN handling of options and changed full_path option name
Fixed output of group acl privileges
Memory leaks (hunt and destroy with tools like Purify) (Kurt)
Minor improvements to rules system
NOTIFY fixes
New asserts for run-checking
Overhauled parser/analyze code to properly report errors and increase speed
Pg_dump -d now handles NULL’s properly (Bruce)
Prevent SELECT NULL from crashing server (Bruce)
Properly report errors when INSERT ... SELECT columns did not match
Properly report errors when insert column names were not correct
psql \g filename now works (Bruce)
psql fixed problem with multiple statements on one line with multiple outputs
Removed duplicate system OIDs
SELECT * INTO TABLE . GROUP/ORDER BY gives unlink error if table exists (Bruce)
Several fixes for queries that crashed the backend
Starting quote in insert string errors (Bruce)
Submitting an empty query now returns empty status, not just " " query (Bruce)

Enhancements
Appendix E. Release Notes

---

Add EXPLAIN manual page (Bruce)
Add UNIQUE index capability (Dan)
Add hostname/user level access control rather than just hostname and user
Add synonym of != for <>(Bruce)
Allow "select oid,* from table"
Allow BY,ORDER BY to specify columns by number, or by non-alias table.column (Bruce)
Allow COPY from the frontend (Bryan)
Allow GROUP BY to use alias column name (Bruce)
Allow actual compression, not just reuse on the same page (Vadim)
Allow installation-configuration option to auto-add all local users (Bryan)
Allow libpq to distinguish between text value " and null (Bruce)
Allow non-postgres users with createdb privs to destroydb’s
Allow restriction on who can create C functions (Bryan)
Allow restriction on who can do backend COPY (Bryan)
Can shrink tables, pg_time and pg_log (Vadim & Erich)
Change debug level 2 to print queries only, changed debug heading layout (Bruce)
Change default decimal constant representation from float4 to float8 (Bruce)
European date format now set when postmaster is started
Execute lowercase function names if not found with exact case
Fixes for aggregate/GROUP processing, allow 'select sum(func(x),sum(x+y) from z’
Gist now included in the distribution (Marc)
Idend authentication of local users (Bryan)
Implement BETWEEN qualifier (Bruce)
Implement IN qualifier (Bruce)
libpq has PQgetisnull() (Bruce)
libpq++ improvements
New options to initdb (Bryan)
Pg_dump allow dump of OIDs (Bruce)
Pg_dump create indexes after tables are loaded for speed (Bruce)
Pg_dumpall dumps all databases, and the user table
Pginterface additions for NULL values (Bruce)
Prevent postmaster from being run as root
psql \h and \? is now readable (Bruce)
psql allow backslashed, semicolons anywhere on the line (Bruce)
psql changed command prompt for lines in query or in quotes (Bruce)
psql char(3) now displays as (bp)char in \d output (Bruce)
psql return code now more accurate (Bryan?)
psql updated help syntax (Bruce)
Re-visit and fix vacuum (Vadim)
Reduce size of regression diffs, remove timezone name difference (Bruce)
Remove compile-time parameters to enable binary distributions (Bryan)
Reverse meaning of HBA masks (Bryan)
Secure Authentication of local users (Bryan)
Speed up vacuum (Vadim)
Vacuum now had VERBOSE option (Bruce)

Source tree changes
---

All functions now have prototypes that are compared against the calls
Allow asserts to be disabled easily from Makefile.global (Bruce)
Change oid constants used in code to #define names
Decoupled sparc and solaris defines (Kurt)
Gcc -Wall compiles cleanly with warnings only from unfixable constructs
Major include file reorganization/reduction (Marc)
Make now stops on compile failure (Bryan)
Makefile restructuring (Bryan, Marc)
Appendix E. Release Notes

Merge bsdi_2_1 to bsdi (Bruce)
Monitor program removed
Name change from Postgres95 to PostgreSQL
New config.h file (Marc, Bryan)
PG_VERSION now set to 6.0 and used by postmaster
Portability additions, including Ultrix, DG/UX, AIX, and Solaris
Reduced the number of #define’s, centralized #define’s
Remove duplicate OIDS in system tables (Dan)
Remove duplicate system catalog info or report mismatches (Dan)
Removed many os-specific #define’s
Restructured object file generation/location (Bryan, Marc)
Restructured port-specific file locations (Bryan, Marc)
Unused/uninitialized variables corrected

E.185. Release 1.09

Release date: 1996-11-04

Sorry, we didn’t keep track of changes from 1.02 to 1.09. Some of the changes listed in 6.0 were actually included in the 1.02.1 to 1.09 releases.

E.186. Release 1.02

Release date: 1996-08-01

E.186.1. Migration from version 1.02 to version 1.02.1

Here is a new migration file for 1.02.1. It includes the ’copy’ change and a script to convert old ASCII files.

Note: The following notes are for the benefit of users who want to migrate databases from Postgres95 1.01 and 1.02 to Postgres95 1.02.1.

If you are starting afresh with Postgres95 1.02.1 and do not need to migrate old databases, you do not need to read any further.

In order to upgrade older Postgres95 version 1.01 or 1.02 databases to version 1.02.1, the following steps are required:

1. Start up a new 1.02.1 postmaster
2. Add the new built-in functions and operators of 1.02.1 to 1.01 or 1.02 databases. This is done by running the new 1.02.1 server against your own 1.01 or 1.02 database and applying the queries
attached at the end of the file. This can be done easily through psql. If your 1.01 or 1.02 database is named testdb and you have cut the commands from the end of this file and saved them in addfunc.sql:

```
% psql testdb -f addfunc.sql
```

Those upgrading 1.02 databases will get a warning when executing the last two statements in the file because they are already present in 1.02. This is not a cause for concern.

### E.186.2. Dump/Reload Procedure

If you are trying to reload a pg_dump or text-mode, copy tablename to stdout generated with a previous version, you will need to run the attached sed script on the ASCII file before loading it into the database. The old format used `.` as end-of-data, while `\` is now the end-of-data marker. Also, empty strings are now loaded in as `"` rather than NULL. See the copy manual page for full details.

```
sed 's/\./\./g' <in_file >out_file
```

If you are loading an older binary copy or non-stdout copy, there is no end-of-data character, and hence no conversion necessary.

```bash
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
cREATE OPERATOR ~* (leftarg = bpchar, rightarg = text, procedure = texticregexeq);
cREATE OPERATOR !~* (leftarg = bpchar, rightarg = text, procedure = texticregexne);
cREATE OPERATOR ~* (leftarg = varchar, rightarg = text, procedure = texticregexeq);
cREATE OPERATOR !~* (leftarg = varchar, rightarg = text, procedure = texticregexne);
```

### E.186.3. Changes

**Source code maintenance and development**
- worldwide team of volunteers
- the source tree now in CVS at ftp.ki.net

**Enhancements**
- psql (and underlying libpq library) now has many more options for formatting output, including HTML
- pg_dump now output the schema and/or the data, with many fixes to enhance completeness.
- psql used in place of monitor in administration shell scripts. monitor to be deprecated in next release.
- date/time functions enhanced
- NULL insert/update/comparison fixed/enhanced
- TCL/TK lib and shell fixed to work with both tck7.4/tk4.0 and tcl7.5/tk4.1

**Bug Fixes** (almost too numerous to mention)
- indexes
- storage management
- check for NULL pointer before dereferencing
- Makefile fixes
Appendix E. Release Notes

New Ports
* added SolarisX86 port
* added BSD/OS 2.1 port
* added DG/UX port

E.187. Release 1.01

Release date: 1996-02-23

E.187.1. Migration from version 1.0 to version 1.01

The following notes are for the benefit of users who want to migrate databases from Postgres95 1.0 to Postgres95 1.01.

If you are starting afresh with Postgres95 1.01 and do not need to migrate old databases, you do not need to read any further.

In order to Postgres95 version 1.01 with databases created with Postgres95 version 1.0, the following steps are required:

1. Set the definition of NAMEDATALEN in src/Makefile.global to 16 and OIDNAMELEN to 20.
2. Decide whether you want to use Host based authentication.
   a. If you do, you must create a file name pg_hba in your top-level data directory (typically the value of your $PGDATA). src/libpq/pg_hba shows an example syntax.
   b. If you do not want host-based authentication, you can comment out the line:
      HBA = 1
      in src/Makefile.global
      Note that host-based authentication is turned on by default, and if you do not take steps A or B above, the out-of-the-box 1.01 will not allow you to connect to 1.0 databases.
3. Compile and install 1.01, but DO NOT do the initdb step.
4. Before doing anything else, terminate your 1.0 postmaster, and backup your existing $PGDATA directory.
5. Set your $PGDATA environment variable to your 1.0 databases, but set up path up so that 1.01 binaries are being used.
6. Modify the file $PGDATA/PG_VERSION from 5.0 to 5.1
7. Start up a new 1.01 postmaster
8. Add the new built-in functions and operators of 1.01 to 1.0 databases. This is done by running the new 1.01 server against your own 1.0 database and applying the queries attached and saving in the file 1.0_to_1.01.sql. This can be done easily through psql. If your 1.0 database is name testdb:

   % psql testdb -f 1.0_to_1.01.sql
and then execute the following commands (cut and paste from here):

-- add builtin functions that are new to 1.01

create function int4eqoid (int4, oid) returns bool as 'foo'
    language 'internal';
create function oideqint4 (oid, int4) returns bool as 'foo'
    language 'internal';
create function char2icregexeq (char2, text) returns bool as 'foo'
    language 'internal';
create function char2icregexne (char2, text) returns bool as 'foo'
    language 'internal';
create function char4icregexeq (char4, text) returns bool as 'foo'
    language 'internal';
create function char4icregexne (char4, text) returns bool as 'foo'
    language 'internal';
create function char8icregexeq (char8, text) returns bool as 'foo'
    language 'internal';
create function char8icregexne (char8, text) returns bool as 'foo'
    language 'internal';
create function char16icregexeq (char16, text) returns bool as 'foo'
    language 'internal';
create function char16icregexne (char16, text) returns bool as 'foo'
    language 'internal';
create function texticregexeq (text, text) returns bool as 'foo'
    language 'internal';
create function texticregexne (text, text) returns bool as 'foo'
    language 'internal';

-- add builtin functions that are new to 1.01

create operator = (leftarg = int4, rightarg = oid, procedure = int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure = oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure = char2icregexeq);
create operator !~* (leftarg = char2, rightarg = text, procedure = char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure = char4icregexeq);
create operator !~* (leftarg = char4, rightarg = text, procedure = char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure = char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure = char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure = char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure = char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure = texticregexne);

E.187.2. Changes

Incompatibilities:
* 1.01 is backwards compatible with 1.0 database provided the user
   follow the steps outlined in the MIGRATION_from_1.0_to_1.01 file.
   If those steps are not taken, 1.01 is not compatible with 1.0 database.

Enhancements:
* added PQdisplayTuples() to libpq and changed monitor and psql to use it
* added NeXT port (requires SysVIPC implementation)
* added CAST .. AS ... syntax
Appendix E. Release Notes

• added ASC and DESC key words
• added 'internal' as a possible language for CREATE FUNCTION
  internal functions are C functions which have been statically linked
  into the postgres backend.
• a new type "name" has been added for system identifiers (table names,
  attribute names, etc.) This replaces the old char16 type. The
  of name is set by the NAMEDATALEN #define in src/Makefile.global
• a readable reference manual that describes the query language.
• added host-based access control. A configuration file ($PGDATA/pg_hba)
  is used to hold the configuration data. If host-based access control
  is not desired, comment out HBA=1 in src/Makefile.global.
• changed regex handling to be uniform use of Henry Spencer’s regex code
  regardless of platform. The regex code is included in the distribution
• added functions and operators for case-insensitive regular expressions.
  The operators are ~* and !~*.
• pg_dump uses COPY instead of SELECT loop for better performance

Bug fixes:
• fixed an optimizer bug that was causing core dumps when
  functions calls were used in comparisons in the WHERE clause
• changed all uses of getuid to geteuid so that effective uids are used
• psql now returns non-zero status on errors when using -c
• applied public patches 1-14

E.188. Release 1.0

Release date: 1995-09-05

E.188.1. Changes

Copyright change:
• The copyright of Postgres 1.0 has been loosened to be freely modifiable
  and modifiable for any purpose. Please read the COPYRIGHT file.
  Thanks to Professor Michael Stonebraker for making this possible.

Incompatibilities:
• date formats have to be MM-DD-YYYY (or DD-MM-YYYY if you’re using
  EUROPEAN STYLE). This follows SQL-92 specs.
• "delimiters" is now a key word

Enhancements:
• sql LIKE syntax has been added
• copy command now takes an optional USING DELIMITER specification.
  delimiters can be any single-character string.
• IRIX 5.3 port has been added.
  Thanks to Paul Walmsley and others.
• updated pg_dump to work with new libpq
• \d has been added psql
Appendix E. Release Notes

Thanks to Keith Parks
- regexp performance for architectures that use POSIX regex has been improved due to caching of precompiled patterns.
Thanks to Alistair Crooks
- a new version of libpq++
Thanks to William Wanders

Bug fixes:
- arbitrary userids can be specified in the createuser script
- \c to connect to other databases in psql now works.
- bad pg_proc entry for float4inc() is fixed
- users with usecreatedb field set can now create databases without having to be usesuper
- remove access control entries when the entry no longer has any privileges
- fixed non-portable datetimes implementation
- added kerberos flags to the src/backend/Makefile
- libpq now works with kerberos
- typographic errors in the user manual have been corrected.
- btrees with multiple index never worked, now we tell you they don’t work when you try to use them

E.189. Postgres95 Release 0.03

Release date: 1995-07-21

E.189.1. Changes

Incompatible changes:
- BETA-0.3 IS INCOMPATIBLE WITH DATABASES CREATED WITH PREVIOUS VERSIONS (due to system catalog changes and indexing structure changes).
- double-quote (") is deprecated as a quoting character for string literals; you need to convert them to single quotes (‘).
- name of aggregates (eg. int4sum) are renamed in accordance with the SQL standard (eg. sum).
- CHANGE ACL syntax is replaced by GRANT/REVOKE syntax.
- float literals (eg. 3.14) are now of type float4 (instead of float8 in previous releases); you might have to do typecasting if you depend on it being of type float8. If you neglect to do the typecasting and you assign a float literal to a field of type float8, you might get incorrect values stored!
- LIBPQ has been totally revamped so that frontend applications can connect to multiple backends
- the usesysid field in pg_user has been changed from int2 to int4 to allow wider range of Unix user ids.
- the netbsd/freebsd/bsd o/s ports have been consolidated into a single BSD44_derived port. (thanks to Alistair Crooks)
Appendix E. Release Notes

SQL standard-compliance (the following details changes that makes postgres95 more compliant to the SQL-92 standard):
  * the following SQL types are now built-in: smallint, int(eger), float, real, char(N), varchar(N), date and time.

The following are aliases to existing postgres types:
  smallint -> int2
  integer, int -> int4
  float, real -> float4

char(N) and varchar(N) are implemented as truncated text types. In addition, char(N) does blank-padding.
  * single-quote (') is used for quoting string literals; " (in addition to ") is supported as means of inserting a single quote in a string
  * SQL standard aggregate names (MAX, MIN, AVG, SUM, COUNT) are used (Also, aggregates can now be overloaded, i.e. you can define your own MAX aggregate to take in a user-defined type.)
  * CHANGE ACL removed. GRANT/REVOKE syntax added.
    - Privileges can be given to a group using the "GROUP" key word. For example:
      GRANT SELECT ON foobar TO GROUP my_group;
    The key word ‘PUBLIC’ is also supported to mean all users.
    Privileges can only be granted or revoked to one user or group at a time.
    "WITH GRANT OPTION" is not supported. Only class owners can change access control
    - The default access control is to grant users readonly access. You must explicitly grant insert/update access to users. To change this, modify the line in
      src/backend/utils/acl.h
    that defines ACL_WORLD_DEFAULT

Bug fixes:
  * the bug where aggregates of empty tables were not run has been fixed. Now, aggregates run on empty tables will return the initial conditions of the aggregates. Thus, COUNT of an empty table will now properly return 0. MAX/MIN of an empty table will return a row of value NULL.
  * allow the use of \; inside the monitor
  * the LISTEN/NOTIFY asynchronous notification mechanism now work
  * NOTIFY in rule action bodies now work
  * hash indexes work, and access methods in general should perform better. creation of large btree indexes should be much faster. (thanks to Paul Aoki)

Other changes and enhancements:
  * addition of an EXPLAIN statement used for explaining the query execution plan (eg. "EXPLAIN SELECT * FROM EMP" prints out the execution plan for the query).
  * WARN and NOTICE messages no longer have timestamps on them. To turn on timestamps of error messages, uncomment the line in
    src/backend/utils/elog.h:
    /* define ELOG_TIMESTAMPS */
  * On an access control violation, the message
    "Either no such class or insufficient privilege"
  will be given. This is the same message that is returned when a class is not found. This dissuades non-privileged users from
guessing the existence of privileged classes.
* some additional system catalog changes have been made that are not
  visible to the user.

libpgtcl changes:
* The -oid option has been added to the "pg_result" tcl command.
  pg_result -oid returns oid of the last row inserted. If the
  last command was not an INSERT, then pg_result -oid returns "."
* the large object interface is available as pg_lo* tcl commands:
  pg_lo_open, pg_lo_close, pg_lo_creat, etc.

Portability enhancements and New Ports:
* flex/lex problems have been cleared up. Now, you should be able to use
  flex instead of lex on any platforms. We no longer make assumptions of
  what lexer you use based on the platform you use.
* The Linux-ELF port is now supported. Various configuration have been
  tested: The following configuration is known to work:
    kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
    with everything in ELF format,

New utilities:
* ipc_notes added to the distribution
  ipc_notes usually does not need to be run, but if your backend crashes
  and leaves shared memory segments hanging around, ipc_notes will
  clean them up for you.

New documentation:
* the user manual has been revised and libpq documentation added.

---

**E.190. Postgres95 Release 0.02**

**Release date:** 1995-05-25

---

**E.190.1. Changes**

Incompatible changes:
* The SQL statement for creating a database is 'CREATE DATABASE' instead
  of 'CREATEDB'. Similarly, dropping a database is 'DROP DATABASE' instead
  of 'DESTROYDB'. However, the names of the executables 'createdb' and
  'destroydb' remain the same.

New tools:
* pgperl - a Perl (4.036) interface to Postgres95
* pg_dump - a utility for dumping out a postgres database into a
  script file containing query commands. The script files are in a ASCII
  format and can be used to reconstruct the database, even on other
  machines and other architectures. (Also good for converting
  a Postgres 4.2 database to Postgres95 database.)
The following ports have been incorporated into postgres95-beta-0.02:
- the NetBSD port by Alistair Crooks
- the AIX port by Mike Tung
- the Windows NT port by Jon Forrest (more stuff but not done yet)
- the Linux ELF port by Brian Gallew

The following bugs have been fixed in postgres95-beta-0.02:
- new lines not escaped in COPY OUT and problem with COPY OUT when first attribute is a ‘.’
- cannot type return to use the default user id in createuser
- SELECT DISTINCT on big tables crashes
- Linux installation problems
- monitor doesn’t allow use of ‘localhost’ as PGHOST
- psql core dumps when doing \c or \l
- the "pgtclsh" target missing from src/bin/pgtclsh/Makefile
- libpgtcl has a hard-wired default port number
- SELECT DISTINCT INTO TABLE hangs
- CREATE TYPE doesn’t accept ‘variable’ as the internallength
- wrong result using more than 1 aggregate in a SELECT

E.191. Postgres95 Release 0.01

Release date: 1995-05-01

Initial release.
Appendix F. Additional Supplied Modules

This appendix contains information regarding the modules that can be found in the contrib directory of the PostgreSQL distribution. These include porting tools, analysis utilities, and plug-in features that are not part of the core PostgreSQL system, mainly because they address a limited audience or are too experimental to be part of the main source tree. This does not preclude their usefulness.

When building from the source distribution, these modules are not built automatically. You can build and install all of them by running

```
gmake
gmake install
```

in the contrib directory of a configured source tree; or to build and install just one selected module, do the same in that module’s subdirectory. Many of the modules have regression tests, which can be executed by running

```
gmake installcheck
```

once you have a PostgreSQL server running. (Note that gmake check is not supported; you must have an operational database server to perform these tests, and you must have built and installed the module(s) to be tested.)

If you are using a pre-packaged version of PostgreSQL, these modules are typically made available as a separate subpackage, such as postgresql-contrib.

Many modules supply new user-defined functions, operators, or types. To make use of one of these modules, after you have installed the code you need to register the new objects in the database system by running the SQL commands in the .sql file supplied by the module. For example,

```
psql -d dbname -f SHAREDIR/contrib/module.sql
```

Here, SHAREDIR means the installation’s “share” directory (pg_config --sharedir will tell you what this is). In most cases the script must be run by a database superuser.

You need to run the .sql file in each database that you want the module’s facilities to be available in. Alternatively, run it in database template1 so that the module will be copied into subsequently-created databases by default.

You can modify the first command in the .sql file to determine which schema within the database the module’s objects will be created in. By default, they will be placed in public.

After a major-version upgrade of PostgreSQL, run the installation script again, even though the module’s objects might have been brought forward from the old installation by dump and restore. This ensures that any new functions will be available and any needed corrections will be applied.

F.1. adminpack

adminpack provides a number of support functions which pgAdmin and other administration and management tools can use to provide additional functionality, such as remote management of server log files.
Appendix F. Additional Supplied Modules

F.1.1. Functions implemented

The functions implemented by adminpack can only be run by a superuser. Here’s a list of these functions:

```sql
int8 pg_catalog.pg_file_write(fname text, data text, append bool)
bool pg_catalog.pg_file_rename(oldname text, newname text, archivename text)
bool pg_catalog.pg_file_rename(oldname text, newname text)
bool pg_catalog.pg_file_unlink(fname text)
setof record pg_catalog.pg_logdir_ls()
/* Renaming of existing backend functions for pgAdmin compatibility */
int8 pg_catalog.pg_file_read(fname text, data text, append bool)
bigint pg_catalog.pg_file_length(text)
int4 pg_catalog.pg_logfile_rotate()
```

F.2. btree_gist

btree_gist provides sample GiST operator classes that implement B-Tree equivalent behavior for the data types int2, int4, int8, float4, float8, numeric, timestamp with time zone, timestamp without time zone, time with time zone, time without time zone, date, interval, oid, money, char, varchar, text, bytea, bit, varbit, macaddr, inet, and cidr.

In general, these operator classes will not outperform the equivalent standard btree index methods, and they lack one major feature of the standard btree code: the ability to enforce uniqueness. However, they are useful for GiST testing and as a base for developing other GiST operator classes.

F.2.1. Example usage

```sql
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING gist (a);
-- query
SELECT * FROM test WHERE a < 10;
```

F.2.2. Authors

Teodor Sigaev (<teodor@stack.net>), Oleg Bartunov (<oleg@sai.msu.su>), and Janko Richter (<jankorichter@yahoo.de>). See http://www.sai.msu.su/~megera/postgres/gist for additional information.

F.3. chkpass

This module implements a data type chkpass that is designed for storing encrypted passwords. Each password is automatically converted to encrypted form upon entry, and is always stored encrypted.
To compare, simply compare against a clear text password and the comparison function will encrypt it before comparing.

There are provisions in the code to report an error if the password is determined to be easily crackable. However, this is currently just a stub that does nothing.

If you precede an input string with a colon, it is assumed to be an already-encrypted password, and is stored without further encryption. This allows entry of previously-encrypted passwords.

On output, a colon is prepended. This makes it possible to dump and reload passwords without re-encrypting them. If you want the encrypted password without the colon then use the raw() function. This allows you to use the type with things like Apache’s Auth_PostgreSQL module.

The encryption uses the standard Unix function crypt(), and so it suffers from all the usual limitations of that function; notably that only the first eight characters of a password are considered.

Note that the chkpass data type is not indexable.

Sample usage:

test=# create table test (p chkpass);
CREATE TABLE

test=# insert into test values ('hello');
INSERT 0 1

test=# select * from test;
p  
-------------  
:dVGkpXdorE3ko  
(1 row)

test=# select raw(p) from test;
raw  
-------------  
dVGkpXdorE3ko  
(1 row)

test=# select p = 'hello' from test;
?column?
----------  
t  
(1 row)

test=# select p = 'goodbye' from test;
?column?
----------  
f  
(1 row)

F.3.1. Author

D’Arcy J.M. Cain (<darcy@druid.net>)
F.4. cube

This module implements a data type `cube` for representing multi-dimensional cubes.

F.4.1. Syntax

The following are valid external representations for the `cube` type. `x`, `y`, etc denote floating-point numbers:

<table>
<thead>
<tr>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>A one-dimensional point (or, zero-length one-dimensional interval)</td>
</tr>
<tr>
<td><code>(x)</code></td>
<td>Same as above</td>
</tr>
<tr>
<td><code>x1, x2, ..., xn</code></td>
<td>A point in n-dimensional space, represented internally as a zero-volume cube</td>
</tr>
<tr>
<td><code>(x1, x2, ..., xn)</code></td>
<td>Same as above</td>
</tr>
<tr>
<td><code>(x), (y)</code></td>
<td>A one-dimensional interval starting at <code>x</code> and ending at <code>y</code> or vice versa; the order does not matter</td>
</tr>
<tr>
<td><code>[(x), (y)]</code></td>
<td>Same as above</td>
</tr>
<tr>
<td><code>(x1, ..., xn), (y1, ..., yn)</code></td>
<td>An n-dimensional cube represented by a pair of its diagonally opposite corners</td>
</tr>
<tr>
<td><code>[(x1, ..., xn), (y1, ..., yn)]</code></td>
<td>Same as above</td>
</tr>
</tbody>
</table>

It does not matter which order the opposite corners of a cube are entered in. The `cube` functions automatically swap values if needed to create a uniform “lower left — upper right” internal representation. White space is ignored, so `[(x), (y)]` is the same as `[(x), (y)]`.

F.4.2. Precision

Values are stored internally as 64-bit floating point numbers. This means that numbers with more than about 16 significant digits will be truncated.

F.4.3. Usage

The `cube` module includes a GiST index operator class for `cube` values. The operators supported by the GiST opclass include:

- `a = b` **Same as**
  - The cubes `a` and `b` are identical.
- `a && b` **Overlaps**
  - The cubes `a` and `b` overlap.
- `a @> b` **Contains**
  - The cube `a` contains the cube `b`. 
• a ⊊ b

The cube a is contained in the cube b.

(before PostgreSQL 8.2, the containment operators ⊊ ⊋ and ⊊ ⊋ were respectively called ⊊ and ⊊. These names are still available, but are deprecated and will eventually be retired. Notice that the old names are reversed from the convention formerly followed by the core geometric datatypes!)

The standard B-tree operators are also provided, for example

[a, b] < [c, d] Less than
[a, b] > [c, d] Greater than

These operators do not make a lot of sense for any practical purpose but sorting. These operators first compare (a) to (c), and if these are equal, compare (b) to (d). That results in reasonably good sorting in most cases, which is useful if you want to use ORDER BY with this type.

The following functions are available:

<table>
<thead>
<tr>
<th>Table F-2. Cube functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cube(float8) returns cube</strong></td>
</tr>
<tr>
<td><strong>cube(float8, float8) returns cube</strong></td>
</tr>
<tr>
<td><strong>cube(float8[]) returns cube</strong></td>
</tr>
<tr>
<td><strong>cube(float8[], float8[]) returns cube</strong></td>
</tr>
<tr>
<td><strong>cube(cube, float8) returns cube</strong></td>
</tr>
<tr>
<td><strong>cube(cube, float8, float8) returns cube</strong></td>
</tr>
<tr>
<td><strong>cube_dim(cube) returns int</strong></td>
</tr>
<tr>
<td><strong>cube_ll_coord(cube, int) returns double</strong></td>
</tr>
<tr>
<td><strong>cube_ur_coord(cube, int) returns double</strong></td>
</tr>
<tr>
<td><strong>cube_is_point(cube) returns bool</strong></td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cube_distance(cube, cube)</code> returns double</td>
<td>Returns the distance between two cubes. If both cubes are points, this is the normal distance function.</td>
</tr>
<tr>
<td><code>cube_subset(cube, int[])</code> returns cube</td>
<td>Makes a new cube from an existing cube, using a list of dimension indexes from an array. Can be used to find both the LL and UR coordinates of a single dimension, e.g. <code>cube_subset(cube('1,3,5),(6,7,8'), ARRAY[2]) = '(3),(7)'. Or can be used to drop dimensions, or reorder them as desired, e.g. </code>cube_subset(cube('1,3,5),(6,7,8'), ARRAY[3,2,1,1]) = '(5, 3, 1, 1),(8, 7, 6, 6)'.</td>
</tr>
<tr>
<td><code>cube_union(cube, cube)</code> returns cube</td>
<td>Produces the union of two cubes</td>
</tr>
<tr>
<td><code>cube_inter(cube, cube)</code> returns cube</td>
<td>Produces the intersection of two cubes</td>
</tr>
<tr>
<td><code>cube_enlarge(cube c, double r, int n)</code> returns cube</td>
<td>Increases the size of a cube by a specified radius in at least n dimensions. If the radius is negative the cube is shrunk instead. This is useful for creating bounding boxes around a point for searching for nearby points. All defined dimensions are changed by the radius r. LL coordinates are decreased by r and UR coordinates are increased by r. If a LL coordinate is increased to larger than the corresponding UR coordinate (this can only happen when r &lt; 0) than both coordinates are set to their average. If n is greater than the number of defined dimensions and the cube is being increased (r &gt;= 0) then 0 is used as the base for the extra coordinates.</td>
</tr>
</tbody>
</table>

F.4.4. Defaults

I believe this union:

```sql
select cube_union('(0,5,2),(2,3,1)', '0');
```

cube_union
-----------
(0, 0, 0), (2, 5, 2)
(1 row)

does not contradict common sense, neither does the intersection

```sql
select cube_inter('(0,-1),(1,1)', '(0),(2)');
```

cube_inter
-----------
(0, 0), (1, 0)
(1 row)
In all binary operations on differently-dimensioned cubes, I assume the lower-dimensional one to be a cartesian projection, i.e., having zeroes in place of coordinates omitted in the string representation. The above examples are equivalent to:

cube_union('(0,5,2),(2,3,1)', '(0,0,0),(0,0,0)');
cube_inter('(-1), (1,1)', '(-2,0),(2,0)');

The following containment predicate uses the point syntax, while in fact the second argument is internally represented by a box. This syntax makes it unnecessary to define a separate point type and functions for (box, point) predicates.

select cube_contains('(0,0),(1,1)', '0.5,0.5');

F.4.5. Notes

For examples of usage, see the regression test sql/cube.sql.

To make it harder for people to break things, there is a limit of 100 on the number of dimensions of cubes. This is set in cubedata.h if you need something bigger.

F.4.6. Credits

Original author: Gene Selkov, Jr. <selkovjr@mcs.anl.gov>, Mathematics and Computer Science Division, Argonne National Laboratory.

My thanks are primarily to Prof. Joe Hellerstein (http://db.cs.berkeley.edu/~jmh/) for elucidating the gist of the GiST (http://gist.cs.berkeley.edu/), and to his former student, Andy Dong (http://best.me.berkeley.edu/~adong/), for his example written for Illustra, http://garcia.me.berkeley.edu/~adong/rtree. I am also grateful to all Postgres developers, present and past, for enabling myself to create my own world and live undisturbed in it. And I would like to acknowledge my gratitude to Argonne Lab and to the U.S. Department of Energy for the years of faithful support of my database research.

Minor updates to this package were made by Bruno Wolff III <bruno@wolff.to> in August/September of 2002. These include changing the precision from single precision to double precision and adding some new functions.

Additional updates were made by Joshua Reich <josh@root.net> in July 2006. These include cube(float8[], float8[]) and cleaning up the code to use the V1 call protocol instead of the deprecated V0 protocol.

F.5. dblink

dblink is a module which supports connections to other PostgreSQL databases from within a database session.
dlmite_connect

Name
dmite_connect — opens a persistent connection to a remote database

Synopsis
dmite_connect(text connstr) returns text
dmite_connect(text conname, text connstr) returns text

Description
dmite_connect() establishes a connection to a remote PostgreSQL database. The server and
database to be contacted are identified through a standard libpq connection string. Optionally, a name
can be assigned to the connection. Multiple named connections can be open at once, but only one un-
named connection is permitted at a time. The connection will persist until closed or until the database
session is ended.

Arguments

conname

The name to use for this connection; if omitted, an unnamed connection is opened, replacing any
existing unnamed connection.

connstr

libpq-style connection info string, for example hostaddr=127.0.0.1 port=5432
dbnme=mydb user=postgres password=mypassword. For details see PQconnectdb in
Section 30.1.

Return Value

Returns status, which is always OK (since any error causes the function to throw an error instead of
returning).

Notes

Only superusers may use dmite_connect to create non-password-authenticated connections. If
non-superusers need this capability, use dmite_connect_u instead.

It is unwise to choose connection names that contain equal signs, as this opens a risk of confusion
with connection info strings in other dmite functions.
Example

```sql
select dblink_connect('dbname=postgres');
```
```
dblink_connect
----------------
  OK
(1 row)
```

```sql
select dblink_connect('myconn', 'dbname=postgres');
```
```
dblink_connect
----------------
  OK
(1 row)
```
**dblink_connect_u**

**Name**

dblink_connect_u — opens a persistent connection to a remote database, insecurely

**Synopsis**

dblink_connect_u(text connstr) returns text  
dblink_connect_u(text connname, text connstr) returns text

**Description**

dblink_connect_u() is identical to dblink_connect(), except that it will allow non-superusers to connect using any authentication method.

If the remote server selects an authentication method that does not involve a password, then impersonation and subsequent escalation of privileges can occur, because the session will appear to have originated from the user as which the local PostgreSQL server runs. Therefore, dblink_connect_u() is initially installed with all privileges revoked from PUBLIC, making it un-callable except by superusers. In some situations it may be appropriate to grant EXECUTE permission for dblink_connect_u() to specific users who are considered trustworthy, but this should be done with care.

For further details see dblink_connect().
**dblink_disconnect**

**Name**

`dblink_disconnect` — closes a persistent connection to a remote database

**Synopsis**

```
dblink_disconnect() returns text
dblink_disconnect(text conname) returns text
```

**Description**

`dblink_disconnect()` closes a connection previously opened by `dblink_connect()`. The form with no arguments closes an unnamed connection.

**Arguments**

`conname`

The name of a named connection to be closed.

**Return Value**

Returns status, which is always **OK** (since any error causes the function to throw an error instead of returning).

**Example**

```
test=# select dblink_disconnect();
        dblink_disconnect
-------------------
           OK         
(1 row)

select dblink_disconnect('myconn');
        dblink_disconnect
-------------------
           OK         
(1 row)
```
**dblink**

**Name**

dblink — executes a query in a remote database

**Synopsis**

```
dblink(text conname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

**Description**

dblink executes a query (usually a SELECT, but it can be any SQL statement that returns rows) in a remote database.

When two text arguments are given, the first one is first looked up as a persistent connection’s name; if found, the command is executed on that connection. If not found, the first argument is treated as a connection info string as for `dblink_connect`, and the indicated connection is made just for the duration of this command.

**Arguments**

- `conname`
  - Name of the connection to use; omit this parameter to use the unnamed connection.
- `connstr`
  - A connection info string, as previously described for `dblink_connect`.
- `sql`
  - The SQL query that you wish to execute in the remote database, for example `select * from foo`.
- `fail_on_error`
  - If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function returns no rows.

**Return Value**

The function returns the row(s) produced by the query. Since dblink can be used with any query, it is declared to return `record`, rather than specifying any particular set of columns. This means that you must specify the expected set of columns in the calling query — otherwise PostgreSQL would not know what to expect. Here is an example:

```
SELECT *
```
The "alias" part of the `FROM` clause must specify the column names and types that the function will return. (Specifying column names in an alias is actually standard SQL syntax, but specifying column types is a PostgreSQL extension.) This allows the system to understand what * `should` expand to, and what `proname` in the `WHERE` clause refers to, in advance of trying to execute the function. At runtime, an error will be thrown if the actual query result from the remote database does not have the same number of columns shown in the `FROM` clause. The column names need not match, however, and `dblink` does not insist on exact type matches either. It will succeed so long as the returned data strings are valid input for the column type declared in the `FROM` clause.

**Notes**

`dblink` fetches the entire remote query result before returning any of it to the local system. If the query is expected to return a large number of rows, it’s better to open it as a cursor with `dblink_open` and then fetch a manageable number of rows at a time.

A convenient way to use `dblink` with predetermined queries is to create a view. This allows the column type information to be buried in the view, instead of having to spell it out in every query. For example,

```sql
create view myremote_pg_proc as
    select *
    from dblink('dbname=postgres', 'select proname, prosrc from pg_proc')
    as t1(proname name, prosrc text);

select * from myremote_pg_proc where proname like 'bytea%';
```

**Example**

```sql
select * from dblink('dbname=postgres', 'select proname, prosrc from pg_proc')
    as t1(proname name, prosrc text) where proname like 'bytea%';

<table>
<thead>
<tr>
<th>proname</th>
<th>prosrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>byteacat</td>
<td>byteacat</td>
</tr>
<tr>
<td>byteaeq</td>
<td>byteaeq</td>
</tr>
<tr>
<td>bytealt</td>
<td>bytealt</td>
</tr>
<tr>
<td>byteale</td>
<td>byteale</td>
</tr>
<tr>
<td>byteagt</td>
<td>byteagt</td>
</tr>
<tr>
<td>byteage</td>
<td>byteage</td>
</tr>
<tr>
<td>byteane</td>
<td>byteane</td>
</tr>
<tr>
<td>byteacmp</td>
<td>byteacmp</td>
</tr>
<tr>
<td>bytealike</td>
<td>bytealike</td>
</tr>
<tr>
<td>byteanlike</td>
<td>byteanlike</td>
</tr>
<tr>
<td>byteain</td>
<td>byteain</td>
</tr>
<tr>
<td>byteaout</td>
<td>byteaout</td>
</tr>
</tbody>
</table>
(12 rows)

select dblink_connect('dbname=postgres');
```

1809
```sql
-- dblink

-- OK

select * from dblink('select proname, prosrc from pg_proc')
  as t1(proname name, prosrc text) where proname like 'bytea%';

proname | prosrc
---------+--------
byteacat | byteacat
byteaeq | byteaeq
bytealt | bytealt
byteale | byteale
byteagt | byteagt
byteage | byteage
byteane | byteane
byteacmp | byteacmp
bytealike | bytealike
byteanlike | byteanlike
byteain | byteain
byteaout | byteaout
(12 rows)

select dblink_connect('myconn', 'dbname=regression');
-- OK

-- (1 row)

select * from dblink('myconn', 'select proname, prosrc from pg_proc')
  as t1(proname name, prosrc text) where proname like 'bytea%';

proname | prosrc
---------+--------
bytearecv | bytearecv
byteasend | byteasend
byteale | byteale
byteagt | byteagt
byteage | byteage
byteane | byteane
byteacmp | byteacmp
bytealike | bytealike
byteanlike | byteanlike
byteacat | byteacat
byteaeq | byteaeq
bytealt | bytealt
byteain | byteain
byteaout | byteaout
(14 rows)
```
**dblink_exec**

**Name**

`dblink_exec` — executes a command in a remote database

**Synopsis**

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

**Description**

`dblink_exec` executes a command (that is, any SQL statement that doesn’t return rows) in a remote database.

When two `text` arguments are given, the first one is first looked up as a persistent connection’s name; if found, the command is executed on that connection. If not found, the first argument is treated as a connection info string as for `dblink_connect`, and the indicated connection is made just for the duration of this command.

**Arguments**

- **connname**
  
  Name of the connection to use; omit this parameter to use the unnamed connection.

- **connstr**
  
  A connection info string, as previously described for `dblink_connect`.

- **sql**
  
  The SQL command that you wish to execute in the remote database, for example `insert into foo values(0,’a’,”a0”,”b0”,”c0”).`

- **fail_on_error**
  
  If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function’s return value is set to ERROR.

**Return Value**

Returns status, either the command’s status string or ERROR.
Example

```sql
select dblink_connect('dbname=dblink_test_slave');
   dblink_connect
  ----------------
    OK
  (1 row)

select dblink_exec('insert into foo values(21,"z","{"a0","b0","c0"}"');
   dblink_exec
  -----------------
    INSERT 943366 1
  (1 row)

select dblink_connect('myconn', 'dbname=regression');
   dblink_connect
  ----------------
    OK
  (1 row)

select dblink_exec('myconn', 'insert into foo values(21,"z","{"a0","b0","c0"}"');
   dblink_exec
  ----------------
    INSERT 6432584 1
  (1 row)

select dblink_exec('myconn', 'insert into pg_class values ("foo")',false);
NOTICE: sql error
DETAIL: ERROR: null value in column "relnamespace" violates not-null constraint
   dblink_exec
  --------------
    ERROR
  (1 row)
```
**dblink_open**

**Name**

`dblink_open` — opens a cursor in a remote database

**Synopsis**

```sql
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
```

**Description**

`dblink_open()` opens a cursor in a remote database. The cursor can subsequently be manipulated with `dblink_fetch()` and `dblink_close()`.

**Arguments**

- **conname**
  
  Name of the connection to use; omit this parameter to use the unnamed connection.

- **cursorname**
  
  The name to assign to this cursor.

- **sql**
  
  The `SELECT` statement that you wish to execute in the remote database, for example `select * from pg_class`.

- **fail_on_error**
  
  If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function’s return value is set to `ERROR`.

**Return Value**

Returns status, either `OK` or `ERROR`.

**Notes**

Since a cursor can only persist within a transaction, `dblink_open` starts an explicit transaction block (`BEGIN`) on the remote side, if the remote side was not already within a transaction. This transaction will be closed again when the matching `dblink_close` is executed. Note that if you use `dblink_exec` to change data between `dblink_open` and `dblink_close`, and then an error occurs or you use `dblink_disconnect` before `dblink_close`, your change will be lost because the transaction will be aborted.
Example

test=# select dblink_connect('dbname=postgres');
dblink_connect
------------
OK
(1 row)

test=# select dblink_open('foo', 'select proname, prosrc from pg_proc');
dblink_open
-------------
OK
(1 row)
**dblink_fetch**

**Name**

dblink_fetch — returns rows from an open cursor in a remote database

**Synopsis**

dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record

dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error]) returns setof record

**Description**

dblink_fetch fetches rows from a cursor previously established by dblink_open.

**Arguments**

conname  
Name of the connection to use; omit this parameter to use the unnamed connection.

cursorname  
The name of the cursor to fetch from.

howmany  
The maximum number of rows to retrieve. The next howmany rows are fetched, starting at the current cursor position, moving forward. Once the cursor has reached its end, no more rows are produced.

fail_on_error  
If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function returns no rows.

**Return Value**

The function returns the row(s) fetched from the cursor. To use this function, you will need to specify the expected set of columns, as previously discussed for dblink.

**Notes**

On a mismatch between the number of return columns specified in the FROM clause, and the actual number of columns returned by the remote cursor, an error will be thrown. In this event, the remote cursor is still advanced by as many rows as it would have been if the error had not occurred. The same is true for any other error occurring in the local query after the remote FETCH has been done.
Example

dblink_connect

Example test=# select dblink_connect('dbname=postgres');
dblink_connect
----------------
OK
(1 row)

dblink_open

test=# select dblink_open('foo', 'select proname, prosrc from pg_proc where proname like 'bytea%');
dblink_open
-------------
OK
(1 row)

dblink_fetch

test=# select * from dblink_fetch('foo', 5) as (funcname name, source text);
funcname | source
----------+----------
byteacat | byteacat
byteacmp | byteacmp
byteaeq | byteaeq
byteage | byteage
byteagt | byteagt
(5 rows)

test=# select * from dblink_fetch('foo', 5) as (funcname name, source text);
funcname | source
----------+----------
byteain | byteain
byteale | byteale
bytealike | bytealike
bytealt | bytealt
byteane | byteane
(5 rows)

test=# select * from dblink_fetch('foo', 5) as (funcname name, source text);
funcname | source
----------+----------
byteanlike | byteanlike
byteaout | byteaout
(2 rows)

test=# select * from dblink_fetch('foo', 5) as (funcname name, source text);
funcname | source
----------+----------
(0 rows)
**dblink_close**

**Name**

`dblink_close` — closes a cursor in a remote database

**Synopsis**

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

**Description**

`dblink_close` closes a cursor previously opened with `dblink_open`.

**Arguments**

- **conname**
  Name of the connection to use; omit this parameter to use the unnamed connection.
- **cursorname**
  The name of the cursor to close.
- **fail_on_error**
  If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function’s return value is set to `ERROR`.

**Return Value**

Returns status, either `OK` or `ERROR`.

**Notes**

If `dblink_open` started an explicit transaction block, and this is the last remaining open cursor in this connection, `dblink_close` will issue the matching `COMMIT`.

**Example**

```
test=# select dblink_connect('dbname=postgres');
dblink_connect
-----------
OK
(1 row)
```
test=# select dblink_open('foo', 'select proname, prosrc from pg_proc');
   dblink_open
            -------------
               OK
(1 row)

test=# select dblink_close('foo');
   dblink_close
            -------------
               OK
(1 row)
dblink_get_connections

Name
dblink_get_connections — returns the names of all open named dblink connections

Synopsis
dblink_get_connections() returns text[]

Description
dblink_get_connections returns an array of the names of all open named dblink connections.

Return Value
Returns a text array of connection names, or NULL if none.

Example
SELECT dblink_get_connections();
**dblink_error_message**

**Name**

`dblink_error_message` — gets last error message on the named connection

**Synopsis**

`dblink_error_message(text connname) returns text`

**Description**

`dblink_error_message` fetches the most recent remote error message for a given connection.

**Arguments**

`connname`

Name of the connection to use.

**Return Value**

Returns last error message, or an empty string if there has been no error in this connection.

**Example**

```sql
SELECT dblink_error_message('dtest1');
```
**dblink_send_query**

**Name**

`dblink_send_query` — sends an async query to a remote database

**Synopsis**

```
dblink_send_query(text connname, text sql) returns int
```

**Description**

`dblink_send_query` sends a query to be executed asynchronously, that is, without immediately waiting for the result. There must not be an async query already in progress on the connection.

After successfully dispatching an async query, completion status can be checked with `dblink_is_busy`, and the results are ultimately collected with `dblink_get_result`. It is also possible to attempt to cancel an active async query using `dblink_cancel_query`.

**Arguments**

- **connname**
  
  Name of the connection to use.

- **sql**
  
  The SQL statement that you wish to execute in the remote database, for example `select * from pg_class`.

**Return Value**

Returns 1 if the query was successfully dispatched, 0 otherwise.

**Example**

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```
**dblink_is_busy**

**Name**

`dblink_is_busy` — checks if connection is busy with an async query

**Synopsis**

```
dblink_is_busy(text connname) returns int
```

**Description**

`dblink_is_busy` tests whether an async query is in progress.

**Arguments**

*conname*

Name of the connection to check.

**Return Value**

Returns 1 if connection is busy, 0 if it is not busy. If this function returns 0, it is guaranteed that `dblink_get_result` will not block.

**Example**

```
SELECT dblink_is_busy('dtest1');
```
**dblink\_get\_result**

**Name**

`dblink_get_result` — gets an async query result

**Synopsis**

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

**Description**

`dblink_get_result` collects the results of an asynchronous query previously sent with `dblink_send_query`. If the query is not already completed, `dblink_get_result` will wait until it is.

**Arguments**

`connname`

Name of the connection to use.

`fail_on_error`

If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function returns no rows.

**Return Value**

For an async query (that is, a SQL statement returning rows), the function returns the row(s) produced by the query. To use this function, you will need to specify the expected set of columns, as previously discussed for `dblink`.

For an async command (that is, a SQL statement not returning rows), the function returns a single row with a single text column containing the command’s status string. It is still necessary to specify that the result will have a single text column in the calling `FROM` clause.

**Notes**

This function must be called if `dblink_send_query` returned 1. It must be called once for each query sent, and one additional time to obtain an empty set result, before the connection can be used again.
Example

```
dlink_connect('dtest1', 'dbname=contrib_regression');
----------------
OK
(1 row)
```

```
contrib_regression=# SELECT dblink_send_query('dtest1', 'select * from foo where f1 < 3') as t1;
t1  ----
    1
(1 row)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
f1 | f2 | f3
----+----+------------
    0 | a  | {a0,b0,c0}
    1 | b  | {a1,b1,c1}
    2 | c  | {a2,b2,c2}
(3 rows)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        0 | a  | {a0, b0, c0}
        1 | b  | {a1, b1, c1}
        2 | c  | {a2, b2, c2}
(3 rows)
```

```
contrib_regression=# SELECT * from dblink_send_query('dtest1', 'select * from foo where f1 < 3; select * from foo where f1 > 6') as t1;
t1  ----
    1
(1 row)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        0 | a  | {a0, b0, c0}
        1 | b  | {a1, b1, c1}
        2 | c  | {a2, b2, c2}
(3 rows)
```

```
contrib_regression=# SELECT * from dblink_send_query('dtest1', 'select * from foo where f1 < 3; select * from foo where f1 > 6') as t1;
t1  ----
    1
(1 row)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        0 | a  | {a0, b0, c0}
        1 | b  | {a1, b1, c1}
        2 | c  | {a2, b2, c2}
(3 rows)
```

```
contrib_regression=# SELECT * from dblink_send_query('dtest1', 'select * from foo where f1 < 3; select * from foo where f1 > 6') as t1;
t1  ----
    1
(1 row)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        0 | a  | {a0, b0, c0}
        1 | b  | {a1, b1, c1}
        2 | c  | {a2, b2, c2}
(3 rows)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        7 | h  | {a7, b7, c7}
        8 | i  | {a8, b8, c8}
        9 | j  | {a9, b9, c9}
       10 | k  | {a10, b10, c10}
(4 rows)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        7 | h  | {a7, b7, c7}
        8 | i  | {a8, b8, c8}
        9 | j  | {a9, b9, c9}
       10 | k  | {a10, b10, c10}
(4 rows)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        7 | h  | {a7, b7, c7}
        8 | i  | {a8, b8, c8}
        9 | j  | {a9, b9, c9}
       10 | k  | {a10, b10, c10}
(4 rows)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        7 | h  | {a7, b7, c7}
        8 | i  | {a8, b8, c8}
        9 | j  | {a9, b9, c9}
       10 | k  | {a10, b10, c10}
(4 rows)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        7 | h  | {a7, b7, c7}
        8 | i  | {a8, b8, c8}
        9 | j  | {a9, b9, c9}
       10 | k  | {a10, b10, c10}
(4 rows)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        7 | h  | {a7, b7, c7}
        8 | i  | {a8, b8, c8}
        9 | j  | {a9, b9, c9}
       10 | k  | {a10, b10, c10}
(4 rows)
```

```
contrib_regression=# SELECT * from dblink_get_result('dtest1') as t1(f1 int, f2 text, f3 text[]);
        f1 | f2 | f3
 ---------------
        7 | h  | {a7, b7, c7}
        8 | i  | {a8, b8, c8}
        9 | j  | {a9, b9, c9}
       10 | k  | {a10, b10, c10}
(4 rows)
```
**dblink_cancel_query**

**Name**

`dblink_cancel_query` — cancels any active query on the named connection

**Synopsis**

```
dblink_cancel_query(text connname) returns text
```

**Description**

`dblink_cancel_query` attempts to cancel any query that is in progress on the named connection. Note that this is not certain to succeed (since, for example, the remote query might already have finished). A cancel request simply improves the odds that the query will fail soon. You must still complete the normal query protocol, for example by calling `dblink_get_result`.

**Arguments**

`connname`  
Name of the connection to use.

**Return Value**

Returns `OK` if the cancel request has been sent, or the text of an error message on failure.

**Example**

```
SELECT dblink_cancel_query('dtest1');
```
dblink_current_query

**Name**

dblink_current_query — returns the current query string

**Synopsis**

dblink_current_query() returns text

**Description**

Returns the currently executing interactive command string of the local database session, or NULL if it can’t be determined. Note that this function is not really related to dblink's other functionality. It is provided since it is sometimes useful in generating queries to be forwarded to remote databases.

**Return Value**

Returns a copy of the currently executing query string.

**Example**

test=# select dblink_current_query();
   dblink_current_query
   --------------------------------
   select dblink_current_query();
   (1 row)
**dblink_get_pkey**

**Name**
dblink_get_pkey — returns the positions and field names of a relation’s primary key fields

**Synopsis**
dblink_get_pkey(text relname) returns setof dblink_pkey_results

**Description**
dblink_get_pkey provides information about the primary key of a relation in the local database. This is sometimes useful in generating queries to be sent to remote databases.

**Arguments**

relname

Name of a local relation, for example foo or myschema.mytab. Include double quotes if the name is mixed-case or contains special characters, for example "FooBar"; without quotes, the string will be folded to lower case.

**Return Value**
Returns one row for each primary key field, or no rows if the relation has no primary key. The result rowtype is defined as

CREATE TYPE dblink_pkey_results AS (position int, colname text);

**Example**

test=# create table foobar(f1 int, f2 int, f3 int,
test(# primary key(f1,f2,f3));
CREATE TABLE
test=# select * from dblink_get_pkey('foobar');
| position | colname |
|----------+---------|
| 1 | f1 |
| 2 | f2 |
| 3 | f3 |

(3 rows)
**dblink_build_sql_insert**

**Name**

dblink_build_sql_insert — builds an INSERT statement using a local tuple, replacing the primary key field values with alternative supplied values

**Synopsis**

dblink_build_sql_insert(text relname, int2vector primary_key_attnums, int2 num_primary_key_atts, text[] src_pk_att_vals_array, text[] tgt_pk_att_vals_array) returns text

**Description**
dblink_build_sql_insert can be useful in doing selective replication of a local table to a remote database. It selects a row from the local table based on primary key, and then builds a SQL INSERT command that will duplicate that row, but with the primary key values replaced by the values in the last argument. (To make an exact copy of the row, just specify the same values for the last two arguments.)

**Arguments**

relname

Name of a local relation, for example foo or myschema.mytab. Include double quotes if the name is mixed-case or contains special characters, for example "FooBar"; without quotes, the string will be folded to lower case.

primary_key_attnums

Attribute numbers (1-based) of the primary key fields, for example 1 2.

num_primary_key_atts

The number of primary key fields.

src_pk_att_vals_array

Values of the primary key fields to be used to look up the local tuple. Each field is represented in text form. An error is thrown if there is no local row with these primary key values.

tgt_pk_att_vals_array

Values of the primary key fields to be placed in the resulting INSERT command. Each field is represented in text form.

**Return Value**

Returns the requested SQL statement as text.
Example

test=# select dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"a"}');

                                dblink_build_sql_insert
----------------------------------
    INSERT INTO foo(f1,f2,f3) VALUES('1','b"a','1')
(1 row)
**dblink_build_sql_delete**

**Name**

dblink_build_sql_delete — builds a DELETE statement using supplied values for primary key field values

**Synopsis**

dblink_build_sql_delete(text relname, int2vector primary_key_attnums, int2 num_primary_key_atts, text[] tgt_pk_att_vals_array) returns text

**Description**
dblink_build_sql_delete can be useful in doing selective replication of a local table to a remote database. It builds a SQL DELETE command that will delete the row with the given primary key values.

**Arguments**

rename

Name of a local relation, for example foo or myschema.mytab. Include double quotes if the name is mixed-case or contains special characters, for example "FooBar"; without quotes, the string will be folded to lower case.

primary_key_attnums

Attribute numbers (1-based) of the primary key fields, for example 1 2.

num_primary_key_atts

The number of primary key fields.

tgt_pk_att_vals_array

Values of the primary key fields to be used in the resulting DELETE command. Each field is represented in text form.

**Return Value**

Returns the requested SQL statement as text.

**Example**

test=# select dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
dblink_build_sql_delete

1830
DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
**dblink_build_sql_update**

**Name**

dblink_build_sql_update — builds an UPDATE statement using a local tuple, replacing the primary key field values with alternative supplied values

**Synopsis**

dblink_build_sql_update(text relname,
   int2vector primary_key_attnums,
   int2 num_primary_key_atts,
   text[] src_pk_att_vals_array,
   text[] tgt_pk_att_vals_array) returns text

**Description**

dblink_build_sql_update can be useful in doing selective replication of a local table to a remote database. It selects a row from the local table based on primary key, and then builds a SQL UPDATE command that will duplicate that row, but with the primary key values replaced by the values in the last argument. (To make an exact copy of the row, just specify the same values for the last two arguments.) The UPDATE command always assigns all fields of the row — the main difference between this and dblink_build_sql_insert is that it’s assumed that the target row already exists in the remote table.

**Arguments**

**relname**

Name of a local relation, for example *foo* or *myschema.mytab*. Include double quotes if the name is mixed-case or contains special characters, for example "FooBar"; without quotes, the string will be folded to lower case.

**primary_key_attnums**

Attribute numbers (1-based) of the primary key fields, for example 1 2.

**num_primary_key_atts**

The number of primary key fields.

**src_pk_att_vals_array**

Values of the primary key fields to be used to look up the local tuple. Each field is represented in text form. An error is thrown if there is no local row with these primary key values.

**tgt_pk_att_vals_array**

Values of the primary key fields to be placed in the resulting UPDATE command. Each field is represented in text form.
**Return Value**

Returns the requested SQL statement as text.

**Example**

```sql
test=# select dblink_build_sql_update('foo', '1 2', 2, '{{"1", "a"}}', '{{"1", "b"}}');

UPDATE foo SET f1='1', f2='b', f3='1' WHERE f1='1' AND f2='b'
(1 row)
```
Appendix F. Additional Supplied Modules

F.6. dict_int

dict_int is an example of an add-on dictionary template for full-text search. The motivation for this example dictionary is to control the indexing of integers (signed and unsigned), allowing such numbers to be indexed while preventing excessive growth in the number of unique words, which greatly affects the performance of searching.

F.6.1. Configuration

The dictionary accepts two options:

- The maxlen parameter specifies the maximum number of digits allowed in an integer word. The default value is 6.
- The rejectlong parameter specifies whether an overlength integer should be truncated or ignored. If rejectlong is false (the default), the dictionary returns the first maxlen digits of the integer. If rejectlong is true, the dictionary treats an overlength integer as a stop word, so that it will not be indexed. Note that this also means that such an integer cannot be searched for.

F.6.2. Usage

Running the installation script creates a text search template intdict_template and a dictionary intdict based on it, with the default parameters. You can alter the parameters, for example

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

or create new dictionaries based on the template.

To test the dictionary, you can try

```
mydb# select ts_lexize('intdict', '12345678');
  ts_lexize
            ----------
             {123456}
```

but real-world usage will involve including it in a text search configuration as described in Chapter 12. That might look like this:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.7. dict_xsyn

dict_xsyn (Extended Synonym Dictionary) is an example of an add-on dictionary template for full-text search. This dictionary type replaces words with groups of their synonyms, and so makes it possible to search for a word using any of its synonyms.
F.7.1. Configuration

A `dict_xsyn` dictionary accepts the following options:

- `keeporig` controls whether the original word is included (if `true`), or only its synonyms (if `false`). Default is `true`.
- `rules` is the base name of the file containing the list of synonyms. This file must be stored in `$SHAREDIR/tsearch_data/` (where `$SHAREDIR` means the PostgreSQL installation’s shared-data directory). Its name must end in `.rules` (which is not to be included in the `rules` parameter).

The rules file has the following format:

- Each line represents a group of synonyms for a single word, which is given first on the line. Synonyms are separated by whitespace, thus:
  
  ```
  word syn1 syn2 syn3
  ```

- The sharp (`#`) sign is a comment delimiter. It may appear at any position in a line. The rest of the line will be skipped.

  Look at `xsyn_sample.rules`, which is installed in `$SHAREDIR/tsearch_data/`, for an example.

F.7.2. Usage

Running the installation script creates a text search template `xsyn_template` and a dictionary `xsyn` based on it, with default parameters. You can alter the parameters, for example

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

or create new dictionaries based on the template.

To test the dictionary, you can try

```
mydb=# SELECT ts_lexize('xsyn', 'word');
```

```
  ts_lexize
  -----------------------
  {word, syn1, syn2, syn3}
```

but real-world usage will involve including it in a text search configuration as described in Chapter 12. That might look like this:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english.stem;
```

F.8. earthdistance

The `earthdistance` module provides two different approaches to calculating great circle distances on the surface of the Earth. The one described first depends on the `cube` package (which **must** be
installed before earthdistance can be installed). The second one is based on the built-in point datatype, using longitude and latitude for the coordinates.

In this module, the Earth is assumed to be perfectly spherical. (If that’s too inaccurate for you, you might want to look at the PostGIS\(^1\) project.)

### F.8.1. Cube-based earth distances

Data is stored in cubes that are points (both corners are the same) using 3 coordinates representing the x, y, and z distance from the center of the Earth. A domain earth over cube is provided, which includes constraint checks that the value meets these restrictions and is reasonably close to the actual surface of the Earth.

The radius of the Earth is obtained from the `earth()` function. It is given in meters. But by changing this one function you can change the module to use some other units, or to use a different value of the radius that you feel is more appropriate.

This package has applications to astronomical databases as well. Astronomers will probably want to change `earth()` to return a radius of \(\frac{180}{\pi}\) so that distances are in degrees.

Functions are provided to support input in latitude and longitude (in degrees), to support output of latitude and longitude, to calculate the great circle distance between two points and to easily specify a bounding box usable for index searches.

The following functions are provided:

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>earth()</code></td>
<td>float8</td>
<td>Returns the assumed radius of the Earth.</td>
</tr>
<tr>
<td><code>sec_to_gc(float8)</code></td>
<td>float8</td>
<td>Converts the normal straight line (secant) distance between two points on the surface of the Earth to the great circle distance between them.</td>
</tr>
<tr>
<td><code>gc_to_sec(float8)</code></td>
<td>float8</td>
<td>Converts the great circle distance between two points on the surface of the Earth to the normal straight line (secant) distance between them.</td>
</tr>
<tr>
<td><code>ll_to_earth(float8, float8)</code></td>
<td>earth</td>
<td>Returns the location of a point on the surface of the Earth given its latitude (argument 1) and longitude (argument 2) in degrees.</td>
</tr>
<tr>
<td><code>latitude(earth)</code></td>
<td>float8</td>
<td>Returns the latitude in degrees of a point on the surface of the Earth.</td>
</tr>
</tbody>
</table>

\(^1\) http://www.postgis.org/
### Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>longitude(earth)</code></td>
<td>float8</td>
<td>Returns the longitude in degrees of a point on the surface of the Earth.</td>
</tr>
<tr>
<td><code>earth_distance(earth, earth)</code></td>
<td>float8</td>
<td>Returns the great circle distance between two points on the surface of the Earth.</td>
</tr>
<tr>
<td><code>earth_box(earth, float8)</code></td>
<td>cube</td>
<td>Returns a box suitable for an indexed search using the <code>cube</code> <code>@&gt;</code> operator for points within a given great circle distance of a location. Some points in this box are further than the specified great circle distance from the location, so a second check using <code>earth_distance</code> should be included in the query.</td>
</tr>
</tbody>
</table>

### F.8.2. Point-based earth distances

The second part of the module relies on representing Earth locations as values of type `point`, in which the first component is taken to represent longitude in degrees, and the second component is taken to represent latitude in degrees. Points are taken as (longitude, latitude) and not vice versa because longitude is closer to the intuitive idea of x-axis and latitude to y-axis.

A single operator is provided:

**Table F-4. Point-based earthdistance operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>point &lt;@&gt; point</code></td>
<td>float8</td>
<td>Gives the distance in statute miles between two points on the Earth’s surface.</td>
</tr>
</tbody>
</table>

Note that unlike the `cube`-based part of the module, units are hardwired here; changing the `earth()` function will not affect the results of this operator.

One disadvantage of the longitude/latitude representation is that you need to be careful about the edge conditions near the poles and near +/- 180 degrees of longitude. The `cube`-based representation avoids these discontinuities.

### F.9. fuzzystrmatch

The `fuzzystrmatch` module provides several functions to determine similarities and distance between strings.
F.9.1. Soundex

The Soundex system is a method of matching similar-sounding names by converting them to the same code. It was initially used by the United States Census in 1880, 1900, and 1910. Note that Soundex is not very useful for non-English names.

The fuzzystrmatch module provides two functions for working with Soundex codes:

- `soundex(text)` returns `text`
- `difference(text, text)` returns `int`

The `soundex` function converts a string to its Soundex code. The `difference` function converts two strings to their Soundex codes and then reports the number of matching code positions. Since Soundex codes have four characters, the result ranges from zero to four, with zero being no match and four being an exact match. (Thus, the function is misnamed — similarity would have been a better name.)

Here are some usage examples:

```sql
SELECT soundex('hello world!');
SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');
```

CREATE TABLE s (nm text);

```sql
INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');
```

```sql
SELECT * FROM s WHERE soundex(nm) = soundex('john');
SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.9.2. Levenshtein

This function calculates the Levenshtein distance between two strings:

- `levenshtein(text source, text target)` returns `int`

Both `source` and `target` can be any non-null string, with a maximum of 255 characters.

Example:

```sql
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
levenshtein
----------
 2
(1 row)
```
F.9.3. Metaphone

Metaphone, like Soundex, is based on the idea of constructing a representative code for an input string. Two strings are then deemed similar if they have the same codes.

This function calculates the metaphone code of an input string:

```
metaphone(text source, int max_output_length) returns text
```

`source` has to be a non-null string with a maximum of 255 characters. `max_output_length` sets the maximum length of the output metaphone code; if longer, the output is truncated to this length.

Example:

```
test=# SELECT metaphone('GUMBO', 4);
metaphone
-----------
KM          
(1 row)
```

F.9.4. Double Metaphone

The Double Metaphone system computes two “sounds like” strings for a given input string — a “primary” and an “alternate”. In most cases they are the same, but for non-English names especially they can be a bit different, depending on pronunciation. These functions compute the primary and alternate codes:

```
dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text
```

There is no length limit on the input strings.

Example:

```
test=# select dmetaphone('gumbo');
dmetaphone
---------------------
KMP              
(1 row)
```

F.10. hstore

This module implements a data type `hstore` for storing sets of (key,value) pairs within a single PostgreSQL data field. This can be useful in various scenarios, such as rows with many attributes that are rarely examined, or semi-structured data.

In the current implementation, neither the key nor the value can exceed 65535 bytes in length; an error will be thrown if this limit is exceeded. These maximum lengths may change in future releases.
F.10.1. hstore External Representation

The text representation of an hstore value includes zero or more key => value items, separated by commas. For example:

```
k => v
  foo => bar, baz => whatever
  "1-a" => "anything at all"
```

The order of the items is not considered significant (and may not be reproduced on output). Whitespace between items or around the => sign is ignored. Use double quotes if a key or value includes whitespace, comma, = or >. To include a double quote or a backslash in a key or value, precede it with another backslash. (Keep in mind that depending on the setting of standard_conforming_strings, you may need to double backslashes in SQL literal strings.)

A value (but not a key) can be a SQL NULL. This is represented as

```
key => NULL
```

The NULL keyword is not case-sensitive. Again, use double quotes if you want the string null to be treated as an ordinary data value.

Currently, double quotes are always used to surround key and value strings on output, even when this is not strictly necessary.

F.10.2. hstore Operators and Functions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>hstore -&gt; text</td>
<td>get value for key (null if not present) `a=&gt;x, b=&gt;y':hstore -&gt; 'a'</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>text =&gt; text</td>
<td>make single-item hstore `'a' =&gt; 'b'</td>
<td>&quot;a&quot;=&gt;&quot;b&quot;</td>
<td></td>
</tr>
<tr>
<td>hstore</td>
<td></td>
<td>hstore</td>
<td>concatenation `a=&gt;b, c=&gt;d':hstore</td>
</tr>
<tr>
<td>hstore ? text</td>
<td>does hstore contain key? `a'=&gt;1':hstore ? 'a'</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>hstore @&gt; hstore</td>
<td>does left operand contain right? `a=&gt;b, b=&gt;1, c=&gt;NULL':hstore @&gt; 'b'=&gt;1'</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>hstore &lt;@ hstore</td>
<td>is left operand contained in right? `a'=&gt;c':hstore &lt;@ 'a'=&gt;b, b=&gt;1, c=&gt;NULL'</td>
<td>f</td>
<td></td>
</tr>
</tbody>
</table>

Before PostgreSQL 8.2, the containment operators @> and <@ were respectively called @ and ~. These names are still available, but are deprecated and will eventually be retired. Notice that the old
names are reversed from the convention formerly followed by the core geometric datatypes!)

Note: The -> operator is deprecated and may be removed in a future release. Use the hstore(text, text) function instead.

Table F-6. hstore Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>hstore(text,</td>
<td>hstore</td>
<td>make single-item</td>
<td>hstore('a',</td>
<td>&quot;a&quot;-&gt;&quot;b&quot;</td>
</tr>
<tr>
<td>text)</td>
<td></td>
<td>hstore</td>
<td>'b')</td>
<td></td>
</tr>
<tr>
<td>akeys(hstore)</td>
<td>text[]</td>
<td>get hstore’s keys as array</td>
<td>akeys('a-&gt;1,b=2')</td>
<td>{a,b}</td>
</tr>
<tr>
<td>skeys(hstore)</td>
<td>setof text</td>
<td>get hstore’s keys as set</td>
<td>skeys('a-&gt;1,b=2')</td>
<td>a,b</td>
</tr>
<tr>
<td>avals(hstore)</td>
<td>text[]</td>
<td>get hstore’s values as array</td>
<td>avals('a-&gt;1,b=2')</td>
<td>{1,2}</td>
</tr>
<tr>
<td>svals(hstore)</td>
<td>setof text</td>
<td>get hstore’s values as set</td>
<td>svals('a-&gt;1,b=2')</td>
<td>1,2</td>
</tr>
<tr>
<td>each(hstore)</td>
<td>setof (key</td>
<td>get hstore’s keys and values</td>
<td>select * from</td>
<td>key</td>
</tr>
<tr>
<td></td>
<td>text, value</td>
<td>and values as set</td>
<td>each('a-&gt;1,b=2')</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>text)</td>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>exist(hstore,</td>
<td>boolean</td>
<td>does hstore contain key?</td>
<td>exist('a-&gt;1',</td>
<td>True</td>
</tr>
<tr>
<td>text)</td>
<td></td>
<td></td>
<td>'a')</td>
<td></td>
</tr>
<tr>
<td>defined(hstore</td>
<td>boolean</td>
<td>does hstore contain non-null</td>
<td>defined('a-&gt;NULL',</td>
<td>False</td>
</tr>
<tr>
<td>text)</td>
<td></td>
<td>value for key?</td>
<td>'a')</td>
<td></td>
</tr>
<tr>
<td>delete(hstore,</td>
<td>hstore</td>
<td>delete any item matching key</td>
<td>delete('a-&gt;1,b=2')</td>
<td>&quot;b&quot;-&gt;&quot;3&quot;</td>
</tr>
</tbody>
</table>

F.10.3. Indexes

hstore has index support for @> and ? operators. You can use either GiST or GIN index types. For example:

CREATE INDEX hidx ON testhstore USING GIST(h);

CREATE INDEX hidx ON testhstore USING GIN(h);

F.10.4. Examples

Add a key, or update an existing key with a new value:

UPDATE tab SET h = h || ('c' => '3');
Delete a key:

```
UPDATE tab SET h = delete(h, 'k1');
```

### F.10.5. Statistics

The `hstore` type, because of its intrinsic liberality, could contain a lot of different keys. Checking for valid keys is the task of the application. Examples below demonstrate several techniques for checking keys and obtaining statistics.

**Simple example:**

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

**Using a table:**

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

**Online statistics:**

```
SELECT key, count(*) FROM 
    (SELECT (each(h)).key FROM testhstore) AS stat
GROUP BY key
ORDER BY count DESC, key;
```

<table>
<thead>
<tr>
<th>key</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>line</td>
<td>883</td>
</tr>
<tr>
<td>query</td>
<td>207</td>
</tr>
<tr>
<td>pos</td>
<td>203</td>
</tr>
<tr>
<td>node</td>
<td>202</td>
</tr>
<tr>
<td>space</td>
<td>197</td>
</tr>
<tr>
<td>status</td>
<td>195</td>
</tr>
<tr>
<td>public</td>
<td>194</td>
</tr>
<tr>
<td>title</td>
<td>190</td>
</tr>
<tr>
<td>org</td>
<td>189</td>
</tr>
</tbody>
</table>

### F.10.6. Authors

**Oleg Bartunov** (<oleg@sai.msu.su>, Moscow, Moscow University, Russia)

**Teodor Sigaev** (<teodor@sigaev.ru>, Moscow, Delta-Soft Ltd., Russia)

### F.11. intagg

The intagg module provides an integer aggregator and an enumerator.
F.11.1. Functions

The aggregator is an aggregate function `int_array_aggregate(integer)` that produces an integer array containing exactly the integers it is fed. Here is a not-tremendously-useful example:

```sql
test=# select int_array_aggregate(i) from
test-# generate_series(1,10,2) i;
```

```
<table>
<thead>
<tr>
<th>int_array_aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,3,5,7,9}</td>
</tr>
</tbody>
</table>
```

(1 row)

The enumerator is a function `int_array_enum(integer[])` that returns `setof integer`. It is essentially the reverse operation of the aggregator: given an array of integers, expand it into a set of rows. For example,

```sql
test=# select * from int_array_enum(array[1,3,5,7,9]);
```

```
<table>
<thead>
<tr>
<th>int_array_enum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
```

(5 rows)

F.11.2. Sample Uses

Many database systems have the notion of a one to many table. Such a table usually sits between two indexed tables, for example:

```sql
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

It is typically used like this:

```sql
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right) WHERE one_to_many.left = item;
```

This will return all the items in the right hand table for an entry in the left hand table. This is a very common construct in SQL.

Now, this methodology can be cumbersome with a very large number of entries in the `one_to_many` table. Often, a join like this would result in an index scan and a fetch for each right hand entry in the table for a particular left hand entry. If you have a very dynamic system, there is not much you can do. However, if you have some data which is fairly static, you can create a summary table with the aggregator.

```sql
CREATE TABLE summary as
SELECT left, int_array_aggregate(right) AS right FROM one_to_many
```
GROUP BY left;

This will create a table with one row per left item, and an array of right items. Now this is pretty useless without some way of using the array; that’s why there is an array enumerator. You can do

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

The above query using `int_array_enum` produces the same results as

```
SELECT left, right FROM one_to_many WHERE left = item;
```

The difference is that the query against the summary table has to get only one row from the table, whereas the direct query against `one_to_many` must index scan and fetch a row for each entry.

On one system, an `EXPLAIN` showed a query with a cost of 8488 was reduced to a cost of 329. The original query was a join involving the `one_to_many` table, which was replaced by:

```
SELECT right, count(right) FROM
(SELECT left, int_array_enum(right) AS right
FROM summary JOIN (SELECT left FROM left_table WHERE left = item) AS lefts
ON (summary.left = lefts.left)
) AS list
GROUP BY right
ORDER BY count DESC;
```

## F.12. `intarray`

The `intarray` module provides a number of useful functions and operators for manipulating one-dimensional arrays of integers. There is also support for indexed searches using some of the operators.

### F.12.1. `intarray` Functions and Operators

#### Table F-7. `intarray` Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>icount(int[])</code></td>
<td>int</td>
<td>number of elements in array</td>
<td><code>icount('1,2,3')</code></td>
<td>3 :: int[]</td>
</tr>
<tr>
<td><code>sort(int[], text dir)</code></td>
<td>int[]</td>
<td>sort array — dir must be asc or desc</td>
<td><code>sort('1,2,3','asc')</code></td>
<td>{1,2,3}</td>
</tr>
<tr>
<td><code>sort(int[])</code></td>
<td>int[]</td>
<td>sort in ascending order</td>
<td><code>sort(array[11,77,44])</code></td>
<td>{11,44,77}</td>
</tr>
<tr>
<td><code>sort_asc(int[])</code></td>
<td>int[]</td>
<td>sort in ascending order</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sort_desc(int[])</code></td>
<td>int[]</td>
<td>sort in descending order</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq(int[])</td>
<td>int[]</td>
<td>remove adjacent duplicates</td>
<td>uniq(sort('{1,2,3,2,1}'::int[]))</td>
<td></td>
</tr>
<tr>
<td>idx(int[], int item)</td>
<td>int</td>
<td>index of first element matching item (0 if none)</td>
<td>idx(array[11,22,33,22,11], 22)</td>
<td></td>
</tr>
<tr>
<td>subarray(int[], int start, int len)</td>
<td>int[]</td>
<td>portion of array starting at position start, len elements</td>
<td>subarray('{1,2,3,2,1}'::int[], 2, 3)</td>
<td>{2,3,2}</td>
</tr>
<tr>
<td>subarray(int[], int start)</td>
<td>int[]</td>
<td>portion of array starting at position start</td>
<td>subarray('{1,2,3,2,1}'::int[], 2)</td>
<td>{2,3,2,1}</td>
</tr>
<tr>
<td>intset(int)</td>
<td>int[]</td>
<td>make single-element array</td>
<td>intset(42)</td>
<td>{42}</td>
</tr>
</tbody>
</table>

Table F-8. intarray Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int[] &amp;@ int[]</td>
<td>boolean</td>
<td>overlap — true if arrays have at least one common element</td>
</tr>
<tr>
<td>int[] @&gt; int[]</td>
<td>boolean</td>
<td>contains — true if left array contains right array</td>
</tr>
<tr>
<td>int[] &lt;@ int[]</td>
<td>boolean</td>
<td>contained — true if left array is contained in right array</td>
</tr>
<tr>
<td># int[]</td>
<td>int</td>
<td>number of elements in array</td>
</tr>
<tr>
<td>int[] # int</td>
<td>int</td>
<td>index (same as idx function)</td>
</tr>
<tr>
<td>int[] + int</td>
<td>int[]</td>
<td>push element onto array (add it to end of array)</td>
</tr>
<tr>
<td>int[] &amp;+ int[]</td>
<td>int[]</td>
<td>array concatenation (right array added to the end of left one)</td>
</tr>
<tr>
<td>int[] - int</td>
<td>int[]</td>
<td>remove entries matching right argument from array</td>
</tr>
<tr>
<td>int[] - int[]</td>
<td>int[]</td>
<td>remove elements of right array from left</td>
</tr>
<tr>
<td>int[]</td>
<td>int</td>
<td>union of arguments</td>
</tr>
<tr>
<td>int[]</td>
<td>int</td>
<td>union of arrays</td>
</tr>
<tr>
<td>int[] &amp; int[]</td>
<td>int[]</td>
<td>intersection of arrays</td>
</tr>
<tr>
<td>int[] @@ query_int</td>
<td>boolean</td>
<td>true if array satisfies query (see below)</td>
</tr>
<tr>
<td>query_int ~~ int[]</td>
<td>boolean</td>
<td>true if array satisfies query (commutator of @@)</td>
</tr>
</tbody>
</table>

(Before PostgreSQL 8.2, the containment operators @> and <@ were respectively called @ and ~. These names are still available, but are deprecated and will eventually be retired. Notice that the old names are reversed from the convention formerly followed by the core geometric datatypes!)
Appendix F. Additional Supplied Modules

The `@` and `~~` operators test whether an array satisfies a query, which is expressed as a value of a specialized data type `query_int`. A query consists of integer values that are checked against the elements of the array, possibly combined using the operators `&` (AND), `|` (OR), and `!` (NOT). Parentheses can be used as needed. For example, the query `1 & (2 | 3)` matches arrays that contain 1 and also contain either 2 or 3.

F.12.2. Index Support

`intarray` provides index support for the `&&`, `<@`, `@>`, and `@@` operators, as well as regular array equality. The implementation uses an RD-tree data structure with built-in lossy compression.

Two GiST index operator classes are provided: `gist__int_ops` (used by default) is suitable for small and medium-size arrays, while `gist__intbig_ops` uses a larger signature and is more suitable for indexing large arrays.

There is also a non-default GIN operator class `gin__int_ops`.

The choice between GiST and GIN indexing depends on the relative performance characteristics of GiST and GIN, which are discussed elsewhere. As a rule of thumb, a GIN index is faster to search than a GiST index, but slower to build or update; so GIN is better suited for static data and GiST for often-updated data.

F.12.3. Example

```
-- a message can be in one or more "sections"
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- create specialized index
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops);

-- select messages in section 1 OR 2 - OVERLAP operator
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- select messages in sections 1 AND 2 - CONTAINS operator
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- the same, using QUERY operator
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.12.4. Benchmark

The source directory `contrib/intarray/bench` contains a benchmark test suite. To run:

```
cd .../bench
createdb TEST
psql TEST < ../_int.sql
./create_test.pl | psql TEST
./bench.pl
```

1846
The `bench.pl` script has numerous options, which are displayed when it is run without any arguments.

F.12.5. Authors

All work was done by Teodor Sigaev (<teodor@stack.net>) and Oleg Bartunov (<oleg@sai.msu.su>). See http://www.sai.msu.su/~megera/postgres/gist for additional information. Andrey Oktyabrski did a great work on adding new functions and operations.

F.13. isn

The `isn` module provides data types for the following international product numbering standards: EAN13, UPC, ISBN (books), ISMN (music), and ISSN (serials). Numbers are validated on input, and correctly hyphenated on output.

F.13.1. Data types

Table F-9. `isn` data types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAN13</td>
<td>European Article Numbers, always displayed in the EAN13 display format</td>
</tr>
<tr>
<td>ISBN13</td>
<td>International Standard Book Numbers to be displayed in the new EAN13 display format</td>
</tr>
<tr>
<td>ISMN13</td>
<td>International Standard Music Numbers to be displayed in the new EAN13 display format</td>
</tr>
<tr>
<td>ISSN13</td>
<td>International Standard Serial Numbers to be displayed in the new EAN13 display format</td>
</tr>
<tr>
<td>ISBN</td>
<td>International Standard Book Numbers to be displayed in the old short display format</td>
</tr>
<tr>
<td>ISMN</td>
<td>International Standard Music Numbers to be displayed in the old short display format</td>
</tr>
<tr>
<td>ISSN</td>
<td>International Standard Serial Numbers to be displayed in the old short display format</td>
</tr>
<tr>
<td>UPC</td>
<td>Universal Product Codes</td>
</tr>
</tbody>
</table>

Some notes:

1. ISBN13, ISMN13, ISSN13 numbers are all EAN13 numbers.
2. EAN13 numbers aren’t always ISBN13, ISMN13 or ISSN13 (some are).
3. Some ISBN13 numbers can be displayed as ISBN.
4. Some ISMN13 numbers can be displayed as ISMN.
5. Some ISSN13 numbers can be displayed as ISSN.
6. UPC numbers are a subset of the EAN13 numbers (they are basically EAN13 without the first 0 digit).

7. All UPC, ISBN, ISMN and ISSN numbers can be represented as EAN13 numbers.

Internally, all these types use the same representation (a 64-bit integer), and all are interchangeable. Multiple types are provided to control display formatting and to permit tighter validity checking of input that is supposed to denote one particular type of number.

The ISBN, ISMN, and ISSN types will display the short version of the number (ISxN 10) whenever it’s possible, and will show ISxN 13 format for numbers that do not fit in the short version. The EAN13, ISBN13, ISMN13 and ISSN13 types will always display the long version of the ISxN (EAN13).

F.13.2. Casts

The isn module provides the following pairs of type casts:

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

When casting from EAN13 to another type, there is a run-time check that the value is within the domain of the other type, and an error is thrown if not. The other casts are simply relabelings that will always succeed.

F.13.3. Functions and Operators

The isn module provides the standard comparison operators, plus btree and hash indexing support for all these datatypes. In addition there are several specialized functions. In this table, isn means any one of the module’s data types:

Table F-10. isn functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isn_weak(boolean)</td>
<td>boolean</td>
<td>Sets the weak input mode (returns new setting)</td>
</tr>
<tr>
<td>isn_weak()</td>
<td>boolean</td>
<td>Gets the current status of the weak mode</td>
</tr>
<tr>
<td>make_valid(isn)</td>
<td>isn</td>
<td>Validates an invalid number (clears the invalid flag)</td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_valid(isn)</td>
<td>boolean</td>
<td>Checks for the presence of the invalid flag</td>
</tr>
</tbody>
</table>

Weak mode is used to be able to insert invalid data into a table. Invalid means the check digit is wrong, not that there are missing numbers.

Why would you want to use the weak mode? Well, it could be that you have a huge collection of ISBN numbers, and that there are so many of them that for weird reasons some have the wrong check digit (perhaps the numbers were scanned from a printed list and the OCR got the numbers wrong, perhaps the numbers were manually captured... who knows). Anyway, the point is you might want to clean the mess up, but you still want to be able to have all the numbers in your database and maybe use an external tool to locate the invalid numbers in the database so you can verify the information and validate it more easily; so for example you’d want to select all the invalid numbers in the table.

When you insert invalid numbers in a table using the weak mode, the number will be inserted with the corrected check digit, but it will be displayed with an exclamation mark (!) at the end, for example 0-11-000322-5!. This invalid marker can be checked with the is_valid function and cleared with the make_valid function.

You can also force the insertion of invalid numbers even when not in the weak mode, by appending the ! character at the end of the number.

Another special feature is that during input, you can write ? in place of the check digit, and the correct check digit will be inserted automatically.

F.13.4. Examples

--Using the types directly:
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

--Casting types:
-- note that you can only cast from ean13 to another type when the
-- number would be valid in the realm of the target type;
-- thus, the following will NOT work: select isbn(ean13('0220356483481'));
-- but these will:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

--Create a table with a single column to hold ISBN numbers:
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

--Automatically calculate check digits (observe the '??'):
INSERT INTO test VALUES('220500896??');
INSERT INTO test VALUES('978055215372??');

SELECT issn('3251231??');
SELECT ismn('979047213542??');

--Using the weak mode:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
Appendix F. Additional Supplied Modules

INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X';

SELECT * FROM test;
SELECT isbn13(id) FROM test;

F.13.5. Bibliography

The information to implement this module was collected from several sites, including:

http://www.isbn-international.org/
http://www.issn.org/
http://www.ismn-international.org/
http://www.wikipedia.org/

The prefixes used for hyphenation were also compiled from:

http://www.gs1.org/productssolutions/idkeys/support/prefix_list.html
http://www.isbn-international.org/en/identifiers.html
http://www.ismn-international.org/ranges.html

Care was taken during the creation of the algorithms and they were meticulously verified against the suggested algorithms in the official ISBN, ISMN, ISSN User Manuals.

F.13.6. Author

Germán Méndez Bravo (Kronuz), 2004 - 2006

This module was inspired by Garrett A. Wollman’s isbn_issn code.

F.14. lo

The lo module provides support for managing Large Objects (also called LOs or BLOBs). This includes a data type lo and a trigger lo_manage.

F.14.1. Rationale

One of the problems with the JDBC driver (and this affects the ODBC driver also), is that the specification assumes that references to BLOBs (Binary Large OBjects) are stored within a table, and if that entry is changed, the associated BLOB is deleted from the database.
As PostgreSQL stands, this doesn’t occur. Large objects are treated as objects in their own right; a table entry can reference a large object by OID, but there can be multiple table entries referencing the same large object OID, so the system doesn’t delete the large object just because you change or remove one such entry.

Now this is fine for PostgreSQL-specific applications, but standard code using JDBC or ODBC won’t delete the objects, resulting in orphan objects — objects that are not referenced by anything, and simply occupy disk space.

The lo module allows fixing this by attaching a trigger to tables that contain LO reference columns. The trigger essentially just does a lo_unlink whenever you delete or modify a value referencing a large object. When you use this trigger, you are assuming that there is only one database reference to any large object that is referenced in a trigger-controlled column!

The module also provides a data type lo, which is really just a domain of the oid type. This is useful for differentiating database columns that hold large object references from those that are OIDs of other things. You don’t have to use the lo type to use the trigger, but it may be convenient to use it to keep track of which columns in your database represent large objects that you are managing with the trigger. It is also rumored that the ODBC driver gets confused if you don’t use lo for BLOB columns.

**F.14.2. How to Use It**

Here’s a simple example of usage:

```sql
CREATE TABLE image (title TEXT, raster lo);
CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE PROCEDURE lo_manage(raster);
```

For each column that will contain unique references to large objects, create a BEFORE UPDATE OR DELETE trigger, and give the column name as the sole trigger argument. If you need multiple lo columns in the same table, create a separate trigger for each one, remembering to give a different name to each trigger on the same table.

**F.14.3. Limitations**

- Dropping a table will still orphan any objects it contains, as the trigger is not executed. You can avoid this by preceding the DROP TABLE with DELETE FROM table. TRUNCATE has the same hazard.

If you already have, or suspect you have, orphaned large objects, see the contrib/vaccumlo module (Section F.33) to help you clean them up. It’s a good idea to run vacuumlo occasionally as a back-stop to the lo_manage trigger.

- Some frontends may create their own tables, and will not create the associated trigger(s). Also, users may not remember (or know) to create the triggers.

**F.14.4. Author**

Peter Mount <peter@retep.org.uk>
Appendix F. Additional Supplied Modules

F.15. ltree

This module implements a data type ltree for representing labels of data stored in a hierarchical tree-like structure. Extensive facilities for searching through label trees are provided.

F.15.1. Definitions

A **label** is a sequence of alphanumeric characters and underscores (for example, in C locale the characters A-Za-z0-9_ are allowed). Labels must be less than 256 bytes long.

Examples: 42, Personal_Services

A **label path** is a sequence of zero or more labels separated by dots, for example L1.L2.L3, representing a path from the root of a hierarchical tree to a particular node. The length of a label path must be less than 65Kb, but keeping it under 2Kb is preferable. In practice this is not a major limitation; for example, the longest label path in the DMOZ catalogue (http://www.dmoz.org) is about 240 bytes.

Example: Top.Countries.Europe.Russia

The ltree module provides several datatypes:

- **ltree** stores a label path.
- **lquery** represents a regular-expression-like pattern for matching ltree values. A simple word matches that label within a path. A star symbol (*) matches zero or more labels. For example:

  ```
  foo                Match the exact label path foo
  *.foo.*            Match any label path containing the label foo
  **.foo**           Match any label path whose last label is foo
  ```

Star symbols can also be quantified to restrict how many labels they can match:

- `*{n}` Match exactly n labels
- `*{n,}` Match at least n labels
- `*{n,m}` Match at least n but not more than m labels
- `*{,m}` Match at most m labels — same as `*{0,m}`

There are several modifiers that can be put at the end of a non-star label in lquery to make it match more than just the exact match:

- `@` Match case-insensitively, for example a@ matches A
- `*` Match any label with this prefix, for example foo* matches foobar
- `%` Match initial underscore-separated words

The behavior of `%` is a bit complicated. It tries to match words rather than the entire label. For example foo_bar% matches foo_bar baz but not foo_bar baz. If combined with *, prefix matching applies to each word separately, for example foo_bar%* matches fool_bar2_baz but not fool_br2_baz.

Also, you can write several possibly-modified labels separated with `|` (OR) to match any of those labels, and you can put `!` (NOT) at the start to match any label that doesn’t match any of the alternatives.

Here’s an annotated example of lquery:

```
Top.*{0,2}.sport@!.football|tennis.Russ*|Spain
a. b. c. d. e.
```

This query will match any label path that:
Appendix F. Additional Supplied Modules

a. begins with the label Top
b. and next has zero to two labels before
c. a label beginning with the case-insensitive prefix sport
d. then a label not matching football nor tennis
e. and then ends with a label beginning with Russ or exactly matching Spain.

• ltxtquery represents a full-text-search-like pattern for matching ltree values. An ltxtquery value contains words, possibly with the modifiers @, *, % at the end; the modifiers have the same meanings as in lquery. Words can be combined with & (AND), | (OR), ! (NOT), and parentheses. The key difference from lquery is that ltxtquery matches words without regard to their position in the label path.

Here’s an example ltxtquery:

Europe & Russia* & !Transportation

This will match paths that contain the label Europe and any label beginning with Russia (case-insensitive), but not paths containing the label Transportation. The location of these words within the path is not important. Also, when % is used, the word can be matched to any underscore-separated word within a label, regardless of position.

Note: ltxtquery allows whitespace between symbols, but ltree and lquery do not.

F.15.2. Operators and Functions

Type ltree has the usual comparison operators =, <>, <, <=, >=. Comparison sorts in the order of a tree traversal, with the children of a node sorted by label text. In addition, there are the following specialized operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ltree @&gt; ltree</td>
<td>boolean</td>
<td>is left argument an ancestor of right (or equal)?</td>
</tr>
<tr>
<td>ltree &lt;@ ltree</td>
<td>boolean</td>
<td>is left argument a descendant of right (or equal)?</td>
</tr>
<tr>
<td>ltree ~ lquery</td>
<td>boolean</td>
<td>does ltree match lquery?</td>
</tr>
<tr>
<td>lquery ~ ltree</td>
<td>boolean</td>
<td>does ltree match lquery?</td>
</tr>
<tr>
<td>ltree ? lquery[]</td>
<td>boolean</td>
<td>does ltree match any lquery in array?</td>
</tr>
<tr>
<td>lquery[] ? ltree</td>
<td>boolean</td>
<td>does ltree match any lquery in array?</td>
</tr>
<tr>
<td>ltree @ ltxtquery</td>
<td>boolean</td>
<td>does ltree match ltxtquery?</td>
</tr>
<tr>
<td>ltxtquery @ ltree</td>
<td>boolean</td>
<td>does ltree match ltxtquery?</td>
</tr>
<tr>
<td>ltree</td>
<td></td>
<td>ltree</td>
</tr>
</tbody>
</table>
The operators `<@, @>` have analogues `^@, ^@`, which are the same except they do not use indexes. These are useful only for testing purposes.

The following functions are available:

**Table F-12. ltree Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>subltree(ltree, int start, int end)</td>
<td>ltree</td>
<td>subpath of ltree from position start to position end-1 (counting from 0)</td>
<td>subltree('Top.Child1.Child2',1,2)</td>
<td></td>
</tr>
</tbody>
</table>

1854
### Appendix F: Additional Supplied Modules

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>subpath(ltree, int offset, int len)</td>
<td>ltree</td>
<td>subpath of ltree starting at position offset, length len. If offset is negative, subpath starts that far from the end of the path. If len is negative, leaves that many labels off the end of the path.</td>
<td>subpath('Top.Child1.Child2',0,2)</td>
<td></td>
</tr>
<tr>
<td>subpath(ltree, int offset)</td>
<td>ltree</td>
<td>subpath of ltree starting at position offset, extending to end of path. If offset is negative, subpath starts that far from the end of the path.</td>
<td>subpath('Top.Child1.Child2')</td>
<td></td>
</tr>
<tr>
<td>nlevel(ltree)</td>
<td>integer</td>
<td>number of labels in path</td>
<td>nlevel('Top.Child1.Child2')</td>
<td></td>
</tr>
<tr>
<td>index(ltree a, ltree b)</td>
<td>integer</td>
<td>position of first occurrence of b in a: -1 if not found</td>
<td>index('0.1.2.3.5.4.5.6.8.5.6.8','5.6')</td>
<td>index('0.1.2.3.5.4.5.6.8.5.6.8','5.6','-4')</td>
</tr>
<tr>
<td>index(ltree a, ltree b, int offset)</td>
<td>integer</td>
<td>position of first occurrence of b in a, searching starting at offset; negative offset means start -offset labels from the end of the path</td>
<td>index('0.1.2.3.5.4.5.6.8.5.6.8','5.6','-4')</td>
<td></td>
</tr>
<tr>
<td>text2ltree(text)</td>
<td>ltree</td>
<td>cast text to ltree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ltree2text(ltree)</td>
<td>text</td>
<td>cast ltree to text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lca(ltree, ltree, ...)</td>
<td>ltree</td>
<td>lowest common ancestor, i.e., longest common prefix of paths (up to 8 arguments supported)</td>
<td>lca('1.2.2.3','1.2.3.4.5.6')</td>
<td></td>
</tr>
</tbody>
</table>
## Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>lca(ltree[])</td>
<td>ltree</td>
<td>lowest common ancestor, i.e., longest common prefix of paths</td>
<td>lca(array['1.2', '1.2':ltree,'1' 2.3'])</td>
<td></td>
</tr>
</tbody>
</table>

### F.15.3. Indexes

ltree supports several types of indexes that can speed up the indicated operators:

- **B-tree index over ltree:** <, <=, =, >
- **GiST index over ltree:** <, <=, =, >, @, <@, @, ~, ?

Example of creating such an index:

```
CREATE INDEX path_gist_idx ON test USING GIST (path);
```

- **GiST index over ltree[]:** ltree[] <@ ltree, ltree @ ltree[], @, ~, ?

Example of creating such an index:

```
CREATE INDEX path_gist_idx ON test USING GIST (array_path);
```

Note: This index type is lossy.

### F.15.4. Example

This example uses the following data (also available in file contrib/ltree/ltreetest.sql in the source distribution):

```sql
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING gist(path);
CREATE INDEX path_idx ON test USING btree(path);
```

Now, we have a table test populated with data describing the hierarchy shown below:

```
Top
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
|   Science Hobbies Collections
```
Appendix F: Additional Supplied Modules

We can do inheritance:

```
ltreetest=# select path from test where path <@ 'Top.Science';
path
------------------------------------
Top.Science
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
(4 rows)
```

Here are some examples of path matching:

```
ltreetest=# select path from test where path ~ '*.Astronomy.*';
path
-----------------------------------------------
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
Top.Collections.Pictures.Astronomy
(7 rows)
```

```
ltreetest=# select path from test where path ~ '*.!pictures@.*.Astronomy.*';
path
------------------------------------
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
(3 rows)
```

Here are some examples of full text search:

```
ltreetest=# select path from test where path @ 'Astro*% & !pictures@';
path
------------------------------------
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
Top.Hobbies.Amateurs_Astronomy
(4 rows)
```

```
ltreetest=# select path from test where path @ 'Astro* & !pictures@';
path
------------------------------------
Top.Science.Astronomy
```

1857
Appendix F. Additional Supplied Modules

Top.Science.Astronomy.Astrophysics

(3 rows)

Path construction using functions:

ltreetest=# select subpath(path,0,2)||'Space'||subpath(path,2) from test where path <@ 'Top.Science.Astronomy';
?column?
------------------------------------------
Top.Science.Space.Astronomy
(3 rows)

We could simplify this by creating a SQL function that inserts a label at a specified position in a path:

CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
LANGUAGE SQL IMMUTABLE;

ltreetest=# select ins_label(path,2,'Space') from test where path <@ 'Top.Science.Astronomy';
ins_label
------------------------------------------
Top.Science.Space.Astronomy
(3 rows)

F.15.5. Authors

All work was done by Teodor Sigaev (<teodor@stack.net>) and Oleg Bartunov (<oleg@sai.msu.su>). See http://www.sai.msu.su/~megera/postgres/gist for additional information. Authors would like to thank Eugeny Rodichev for helpful discussions. Comments and bug reports are welcome.

F.16. oid2name

oid2name is a utility program that helps administrators to examine the file structure used by PostgreSQL. To make use of it, you need to be familiar with the database file structure, which is described in Chapter 53.

Note: The name "oid2name" is historical, and is actually rather misleading, since most of the time when you use it, you will really be concerned with tables’ filenode numbers (which are the file names visible in the database directories). Be sure you understand the difference between table OIDs and table filenodes!
F.16.1. Overview

oid2name connects to a target database and extracts OID, filenode, and/or table name information. You can also have it show database OIDs or tablespace OIDs. The program is controlled by a large number of command-line switches, as shown in Table F-13.

Table F-13. oid2name switches

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o oid</td>
<td>show info for table with OID oid</td>
</tr>
<tr>
<td>-f filenode</td>
<td>show info for table with filenode filenode</td>
</tr>
<tr>
<td>-t tablename_pattern</td>
<td>show info for table(s) matching tablename_pattern</td>
</tr>
<tr>
<td>-s</td>
<td>show tablespace OIDs</td>
</tr>
<tr>
<td>-S</td>
<td>include system objects (those in information_schema, pg_toast and pg_catalog schemas)</td>
</tr>
<tr>
<td>-i</td>
<td>include indexes and sequences in the listing</td>
</tr>
<tr>
<td>-x</td>
<td>display more information about each object shown: tablespace name, schema name, and OID</td>
</tr>
<tr>
<td>-q</td>
<td>omit headers (useful for scripting)</td>
</tr>
<tr>
<td>-d database</td>
<td>database to connect to</td>
</tr>
<tr>
<td>-H host</td>
<td>database server’s host</td>
</tr>
<tr>
<td>-p port</td>
<td>database server’s port</td>
</tr>
<tr>
<td>-U username</td>
<td>username to connect as</td>
</tr>
</tbody>
</table>

To display specific tables, select which tables to show by using -o, -f and/or -t. -o takes an OID, -f takes a filenode, and -t takes a tablename (actually, it’s a LIKE pattern, so you can use things like foo%). You can use as many of these switches as you like, and the listing will include all objects matched by any of the switches. But note that these switches can only show objects in the database given by -d.

If you don’t give any of -o, -f or -t, but do give -d, it will list all tables in the database named by -d. In this mode, the -S and -i switches control what gets listed.

If you don’t give -d either, it will show a listing of database OIDs. Alternatively you can give -s to get a tablespace listing.

F.16.2. Examples

$ # what’s in this database server, anyway?
$ oid2name
All databases:

<table>
<thead>
<tr>
<th>Oid</th>
<th>Database Name</th>
<th>Tablespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>17228</td>
<td>alvherre</td>
<td>pg_default</td>
</tr>
<tr>
<td>17255</td>
<td>regression</td>
<td>pg_default</td>
</tr>
<tr>
<td>17227</td>
<td>template0</td>
<td>pg_default</td>
</tr>
<tr>
<td>1</td>
<td>template1</td>
<td>pg_default</td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

$ oid2name -s
All tablespaces:

<table>
<thead>
<tr>
<th>Oid</th>
<th>Tablespace Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1663</td>
<td>pg_default</td>
</tr>
<tr>
<td>1664</td>
<td>pg_global</td>
</tr>
<tr>
<td>155151</td>
<td>fastdisk</td>
</tr>
<tr>
<td>155152</td>
<td>bigdisk</td>
</tr>
</tbody>
</table>

$ # OK, let's look into database alvherre
$ cd $PGDATA/base/17228

$ # get top 10 db objects in the default tablespace, ordered by size
$ ls -lS * | head -10

```
-rw------- 1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw------- 1 alvherre alvherre 17965056 sep 14 09:51 1155291
-rw------- 1 alvherre alvherre 1204224 sep 14 09:51 116717
-rw------- 1 alvherre alvherre 581632 sep  6 17:51 1255
-rw------- 1 alvherre alvherre 237568 sep 14 09:50 16674
-rw------- 1 alvherre alvherre 212992 sep 14 09:51 1249
-rw------- 1 alvherre alvherre 204800 sep 14 09:51 16684
-rw------- 1 alvherre alvherre 196608 sep 14 09:50 16700
-rw------- 1 alvherre alvherre 163840 sep 14 09:50 16699
-rw------- 1 alvherre alvherre 122880 sep  6 17:51 16751
```

$ # I wonder what file 155173 is ...
$ oid2name -d alvherre -f 155173
From database "alvherre":

<table>
<thead>
<tr>
<th>Filenode</th>
<th>Table Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>155173</td>
<td>accounts</td>
</tr>
</tbody>
</table>

$ # you can ask for more than one object
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":

<table>
<thead>
<tr>
<th>Filenode</th>
<th>Table Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>155173</td>
<td>accounts</td>
</tr>
<tr>
<td>1155291</td>
<td>accounts_pkey</td>
</tr>
</tbody>
</table>

$ # you can mix the options, and get more details with -x
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":

<table>
<thead>
<tr>
<th>Filenode</th>
<th>Table Name</th>
<th>Oid</th>
<th>Schema</th>
<th>Tablespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>155173</td>
<td>accounts</td>
<td>155173</td>
<td>public</td>
<td>pg_default</td>
</tr>
<tr>
<td>1155291</td>
<td>accounts_pkey</td>
<td>1155291</td>
<td>public</td>
<td>pg_default</td>
</tr>
</tbody>
</table>

$ # show disk space for every db object
$ du [0-9]* | 
> while read SIZE FILENODE
> do
>   echo "$SIZE 'oid2name -q -d alvherre -i -f "$FILENODE"'"
> done

```
16 1155287 branches_pkey
16 1155289 tellers_pkey
17561 1155291 accounts_pkey
```
Appendix F: Additional Supplied Modules

...$ # same, but sort by size
$ du [0-9]* | sort -rn | while read SIZE FN
> do
> echo "$SIZE 'oid2name -q -d alvherre -f $FN''
> done
133466 155173 accounts
17561 1155291 accounts_pkey
1177 16717 pg_proc_proname_args_nsp_index
...

$ # If you want to see what’s in tablespaces, use the pg_tblspc directory
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:

<table>
<thead>
<tr>
<th>Oid</th>
<th>Tablespace Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1663</td>
<td>pg_default</td>
</tr>
<tr>
<td>1664</td>
<td>pg_global</td>
</tr>
<tr>
<td>155151</td>
<td>fastdisk</td>
</tr>
<tr>
<td>155152</td>
<td>bigdisk</td>
</tr>
</tbody>
</table>

$ # what databases have objects in tablespace "fastdisk"?
$ ls -d 155151/*
155151/17228/ 155151/PG_VERSION

$ # Oh, what was database 17228 again?
$ oid2name
All databases:

<table>
<thead>
<tr>
<th>Oid</th>
<th>Database Name</th>
<th>Tablespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>17228</td>
<td>alvherre</td>
<td>pg_default</td>
</tr>
<tr>
<td>17255</td>
<td>regression</td>
<td>pg_default</td>
</tr>
<tr>
<td>17227</td>
<td>template0</td>
<td>pg_default</td>
</tr>
<tr>
<td>1</td>
<td>template1</td>
<td>pg_default</td>
</tr>
</tbody>
</table>

$ # Let’s see what objects does this database have in the tablespace.
$ cd 155151/17228
$ ls -l
total 0
-rw------- 1 postgres postgres 0 sep 13 23:20 155156

$ # OK, this is a pretty small table ... but which one is it?
$ oid2name -d alvherre -f 155156
From database "alvherre":

<table>
<thead>
<tr>
<th>Filenode</th>
<th>Table Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>155156</td>
<td>foo</td>
</tr>
</tbody>
</table>
F.16.3. Limitations

oid2name requires a running database server with non-corrupt system catalogs. It is therefore of only limited use for recovering from catastrophic database corruption situations.

F.16.4. Author

B. Palmer <bpalmer@crimelabs.net>

F.17. pageinspect

The pageinspect module provides functions that allow you to inspect the contents of database pages at a low level, which is useful for debugging purposes. All of these functions may be used only by superusers.

F.17.1. Functions

get_raw_page(text, int) returns bytea

get_raw_page reads the specified block of the named table and returns a copy as a bytea value. This allows a single time-consistent copy of the block to be obtained.

page_header(bytea) returns record

page_header shows fields that are common to all PostgreSQL heap and index pages. A page image obtained with get_raw_page should be passed as argument. For example:

test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));

lsn | tli | flags | lower | upper | special | pagesize | version | prune_xid
---------------------------------+-------+-------+--------+--------+----------+----------+---------+-----------
0/24A1B50 | 1 | 1 | 232 | 368 | 8192 | 8192 | 4 | 0

The returned columns correspond to the fields in the PageHeaderData struct. See src/include/storage/bufpage.h for details.

heap_page_items(bytea) returns setof record

heap_page_items shows all line pointers on a heap page. For those line pointers that are in use, tuple headers are also shown. All tuples are shown, whether or not the tuples were visible to an MVCC snapshot at the time the raw page was copied.

A heap page image obtained with get_raw_page should be passed as argument. For example:

test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));  

See src/include/storage/itemid.h and src/include/access/htup.h for explanations of the fields returned.

bt_metap(text) returns record

bt_metap returns information about a btree index’s metapage. For example:

test=# SELECT * FROM bt_metap('pg_cast_oid_index');

- [ RECORD 1 ]-----
magic | 340322
version | 2
root | 1
level | 0
fastroot | 1
fastlevel | 0

bt_page_stats(text, int) returns record

bt_page_stats returns summary information about single pages of btree indexes. For example:

test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
- [RECORD 1]-----
| blkno | 1
| type | l
| live_items | 256
| dead_items | 0
| avg_item_size | 12
| page_size | 8192
| free_size | 4056
| btpo_prev | 0
| btpo_next | 0
| btpo | 0
| btpo_flags | 3

bt_page_items(text, int) returns setof record

bt_page_items returns detailed information about all of the items on a btree index page. For example:

test=# SELECT * FROM bt_page_items('pg_cast_oid_index', 1);

<table>
<thead>
<tr>
<th>itemoffset</th>
<th>ctid</th>
<th>itemlen</th>
<th>nulls</th>
<th>vars</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0,1)</td>
<td>12</td>
<td>f</td>
<td>f</td>
<td>23 27 00 00</td>
</tr>
<tr>
<td>2</td>
<td>(0,2)</td>
<td>12</td>
<td>f</td>
<td>f</td>
<td>24 27 00 00</td>
</tr>
<tr>
<td>3</td>
<td>(0,3)</td>
<td>12</td>
<td>f</td>
<td>f</td>
<td>25 27 00 00</td>
</tr>
<tr>
<td>4</td>
<td>(0,4)</td>
<td>12</td>
<td>f</td>
<td>f</td>
<td>26 27 00 00</td>
</tr>
<tr>
<td>5</td>
<td>(0,5)</td>
<td>12</td>
<td>f</td>
<td>f</td>
<td>27 27 00 00</td>
</tr>
<tr>
<td>6</td>
<td>(0,6)</td>
<td>12</td>
<td>f</td>
<td>f</td>
<td>28 27 00 00</td>
</tr>
<tr>
<td>7</td>
<td>(0,7)</td>
<td>12</td>
<td>f</td>
<td>f</td>
<td>29 27 00 00</td>
</tr>
<tr>
<td>8</td>
<td>(0,8)</td>
<td>12</td>
<td>f</td>
<td>f</td>
<td>2a 27 00 00</td>
</tr>
</tbody>
</table>

F.18. pgbench

pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction. However, it is easy to test other cases by writing your own transaction script files.

Typical output from pgbench looks like:

transaction type: TPC-B (sort of)
scaling factor: 10
number of clients: 10
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)

The first four lines just report some of the most important parameter settings. The next line reports the number of transactions completed and intended (the latter being just the product of number of clients and number of transactions); these will be equal unless the run failed before completion. The last two lines report the TPS rate, figured with and without counting the time to start database sessions.

### F.18.1. Overview

The default TPC-B-like transaction test requires specific tables to be set up beforehand. pgbench should be invoked with the \(-i\) (initialize) option to create and populate these tables. (When you are testing a custom script, you don’t need this step, but will instead need to do whatever setup your test needs.) Initialization looks like:

```
pgbench -i [ other-options ] dbname
```

where `dbname` is the name of the already-created database to test in. (You may also need \(-h\), \(-p\), and/or \(-U\) options to specify how to connect to the database server.)

**Caution**

`pgbench -i` creates four tables `accounts`, `branches`, `history`, and `tellers`, destroying any existing tables of these names. Be very careful to use another database if you have tables having these names!

At the default “scale factor” of 1, the tables initially contain this many rows:

<table>
<thead>
<tr>
<th>Table</th>
<th># of rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>branches</td>
<td>1</td>
</tr>
<tr>
<td>tellers</td>
<td>10</td>
</tr>
<tr>
<td>accounts</td>
<td>100000</td>
</tr>
<tr>
<td>history</td>
<td>0</td>
</tr>
</tbody>
</table>

You can (and, for most purposes, probably should) increase the number of rows by using the \(-s\) (scale factor) option. The \(-F\) (fillfactor) option might also be used at this point.

Once you have done the necessary setup, you can run your benchmark with a command that doesn’t include \(-i\), that is

```
pgbench [ options ] dbname
```

In nearly all cases, you’ll need some options to make a useful test. The most important options are \(-c\) (number of clients), \(-t\) (number of transactions), and \(-f\) (specify a custom script file). See below for a full list.
Table F-14 shows options that are used during database initialization, while Table F-15 shows options that are used while running benchmarks, and Table F-16 shows options that are useful in both cases.

Table F-14. pgbench initialization options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-i</td>
<td>Required to invoke initialization mode.</td>
</tr>
<tr>
<td>-s scale_factor</td>
<td>Multiply the number of rows generated by the scale factor. For example, <code>-s 100</code> will imply 10,000,000 rows in the accounts table. Default is 1.</td>
</tr>
<tr>
<td>-F fillfactor</td>
<td>Create the accounts, tellers and branches tables with the given fillfactor. Default is 100.</td>
</tr>
</tbody>
</table>

Table F-15. pgbench benchmarking options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c clients</td>
<td>Number of clients simulated, that is, number of concurrent database sessions. Default is 1.</td>
</tr>
<tr>
<td>-t transactions</td>
<td>Number of transactions each client runs. Default is 10.</td>
</tr>
<tr>
<td>-N</td>
<td>Do not update tellers and branches. This will avoid update contention on these tables, but it makes the test case even less like TPC-B.</td>
</tr>
<tr>
<td>-S</td>
<td>Perform select-only transactions instead of TPC-B-like test.</td>
</tr>
<tr>
<td>-f filename</td>
<td>Read transaction script from filename. See below for details. <code>-N</code>, <code>-S</code>, and <code>-f</code> are mutually exclusive.</td>
</tr>
<tr>
<td>-n</td>
<td>No vacuuming is performed before running the test. This option is necessary if you are running a custom test scenario that does not include the standard tables accounts, branches, history, and tellers.</td>
</tr>
<tr>
<td>-v</td>
<td>Vacuum all four standard tables before running the test. With neither <code>-n</code> nor <code>-v</code>, pgbench will vacuum tellers and branches tables, and will remove all entries in history.</td>
</tr>
<tr>
<td>-D varname=value</td>
<td>Define a variable for use by a custom script (see below). Multiple <code>-D</code> options are allowed.</td>
</tr>
<tr>
<td>-C</td>
<td>Establish a new connection for each transaction, rather than doing it just once per client thread. This is useful to measure the connection overhead.</td>
</tr>
<tr>
<td>-l</td>
<td>Write the time taken by each transaction to a logfile. See below for details.</td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-s scale_factor</code></td>
<td>Report the specified scale factor in pgbench’s output. With the built-in tests, this is not necessary; the correct scale factor will be detected by counting the number of rows in the branches table. However, when testing custom benchmarks (-f option), the scale factor will be reported as 1 unless this option is used.</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>Print debugging output.</td>
</tr>
</tbody>
</table>

Table F-16. pgbench common options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h hostname</code></td>
<td>database server’s host</td>
</tr>
<tr>
<td><code>-p port</code></td>
<td>database server’s port</td>
</tr>
<tr>
<td><code>-U login</code></td>
<td>username to connect as</td>
</tr>
</tbody>
</table>

F.18.2. What is the “transaction” actually performed in pgbench?

The default transaction script issues seven commands per transaction:

1. BEGIN;
2. UPDATE accounts SET abalance = abalance + :delta WHERE aid = :aid;
3. SELECT abalance FROM accounts WHERE aid = :aid;
4. UPDATE tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
5. UPDATE branches SET bbalance = bbalance + :delta WHERE bid = :bid;
6. INSERT INTO history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
7. END;

If you specify -N, steps 4 and 5 aren’t included in the transaction. If you specify -S, only the SELECT is issued.

F.18.3. Custom Scripts

pgbench has support for running custom benchmark scenarios by replacing the default transaction script (described above) with a transaction script read from a file (-f option). In this case a “transaction” counts as one execution of a script file. You can even specify multiple scripts (multiple -f options), in which case a random one of the scripts is chosen each time a client session starts a new transaction.

The format of a script file is one SQL command per line; multi-line SQL commands are not supported. Empty lines and lines beginning with -- are ignored. Script file lines can also be “meta commands”, which are interpreted by pgbench itself, as described below.
Appendix F. Additional Supplied Modules

There is a simple variable-substitution facility for script files. Variables can be set by the command-line -D option, explained above, or by the meta commands explained below. In addition to any variables preset by -D command-line options, the variable scale is preset to the current scale factor. Once set, a variable’s value can be inserted into a SQL command by writing :variablename. When running more than one client session, each session has its own set of variables.

Script file meta commands begin with a backslash (\). Arguments to a meta command are separated by white space. These meta commands are supported:

\set varname operand1 [ operator operand2 ]

Sets variable varname to a calculated integer value. Each operand is either an integer constant or a :variablename reference to a variable having an integer value. The operator can be +, -, *, or /.

Example:
\set ntellers 10 * :scale

\setrandom varname min max

Sets variable varname to a random integer value between the limits min and max. Each limit can be either an integer constant or a :variablename reference to a variable having an integer value.

Example:
\setrandom aid 1 :naccounts

\sleep number [ us | ms | s ]

Causes script execution to sleep for the specified duration in microseconds (us), milliseconds (ms) or seconds (s). If the unit is omitted then seconds are the default. number can be either an integer constant or a :variablename reference to a variable having an integer value.

Example:
\sleep 10 ms

As an example, the full definition of the built-in TPC-B-like transaction is:

\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000
BEGIN;
    UPDATE accounts SET abalance = abalance + :delta WHERE aid = :aid;
    SELECT abalance FROM accounts WHERE aid = :aid;
    UPDATE tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
    UPDATE branches SET bbalance = bbalance + :delta WHERE bid = :bid;
    INSERT INTO history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
This script allows each iteration of the transaction to reference different, randomly-chosen rows. (This example also shows why it’s important for each client session to have its own variables — otherwise they’d not be independently touching different rows.)

### F.18.4. Per-transaction logging

With the `-l` option, pgbench writes the time taken by each transaction to a logfile. The logfile will be named `pgbench_log.nnn`, where `nnn` is the PID of the pgbench process. The format of the log is:

```
client_id transaction_no time file_no time_epoch time_us
```

where `time` is the elapsed transaction time in microseconds, `file_no` identifies which script file was used (useful when multiple scripts were specified with `-f`), and `time_epoch/time_us` are a UNIX epoch format timestamp and an offset in microseconds (suitable for creating a ISO 8601 timestamp with fractional seconds) showing when the transaction completed.

Here are example outputs:

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

### F.18.5. Good Practices

It is very easy to use pgbench to produce completely meaningless numbers. Here are some guidelines to help you get useful results.

In the first place, never believe any test that runs for only a few seconds. Increase the `-t` setting enough to make the run last at least a few minutes, so as to average out noise. In some cases you could need hours to get numbers that are reproducible. It’s a good idea to try the test run a few times, to find out if your numbers are reproducible or not.

For the default TPC-B-like test scenario, the initialization scale factor (`-s`) should be at least as large as the largest number of clients you intend to test (`-c`); else you’ll mostly be measuring update contention. There are only `-s` rows in the `branches` table, and every transaction wants to update one of them, so `-c` values in excess of `-s` will undoubtedly result in lots of transactions blocked waiting for other transactions.

The default test scenario is also quite sensitive to how long it’s been since the tables were initialized: accumulation of dead rows and dead space in the tables changes the results. To understand the results you must keep track of the total number of updates and when vacuuming happens. If autovacuum is enabled it can result in unpredictable changes in measured performance.

A limitation of pgbench is that it can itself become the bottleneck when trying to test a large number of client sessions. This can be alleviated by running pgbench on a different machine from the database server, although low network latency will be essential. It might even be useful to run several pgbench instances concurrently, on several client machines, against the same database server.
F.19. pg_buffercache

The pg_buffercache module provides a means for examining what’s happening in the shared buffer cache in real time.

The module provides a C function pg_buffercache_pages that returns a set of records, plus a view pg_buffercache that wraps the function for convenient use.

By default public access is revoked from both of these, just in case there are security issues lurking.

F.19.1. The pg_buffercache view

The definitions of the columns exposed by the view are:

Table F-17. pg_buffercache Columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufferid</td>
<td>integer</td>
<td></td>
<td>ID, in the range 1..shared_buffers</td>
</tr>
<tr>
<td>relfilenode</td>
<td>oid</td>
<td>pg_class.relfilenode</td>
<td>Relfilenode of the relation</td>
</tr>
<tr>
<td>reltablespace</td>
<td>oid</td>
<td>pg_tablespace.oid</td>
<td>Tablespace OID of the relation</td>
</tr>
<tr>
<td>reldatabase</td>
<td>oid</td>
<td>pg_database.oid</td>
<td>Database OID of the relation</td>
</tr>
<tr>
<td>relblocknumber</td>
<td>bigint</td>
<td></td>
<td>Page number within the relation</td>
</tr>
<tr>
<td>isdirty</td>
<td>boolean</td>
<td></td>
<td>Is the page dirty?</td>
</tr>
<tr>
<td>usagecount</td>
<td>smallint</td>
<td></td>
<td>Page LRU count</td>
</tr>
</tbody>
</table>

There is one row for each buffer in the shared cache. Unused buffers are shown with all fields null except bufferid. Shared system catalogs are shown as belonging to database zero.

Because the cache is shared by all the databases, there will normally be pages from relations not belonging to the current database. This means that there may not be matching join rows in pg_class for some rows, or that there could even be incorrect joins. If you are trying to join against pg_class, it’s a good idea to restrict the join to rows having reldatabase equal to the current database’s OID or zero.

When the pg_buffercache view is accessed, internal buffer manager locks are taken for long enough to copy all the buffer state data that the view will display. This ensures that the view produces a consistent set of results, while not blocking normal buffer activity longer than necessary. Nonetheless there could be some impact on database performance if this view is read often.

F.19.2. Sample output

```
regression=# SELECT c.relname, count(*) AS buffers 
FROM pg_buffercache b INNER JOIN pg_class c 
ON b.relfilenode = c.relfilenode AND 
b.reldatabase IN (0, (SELECT oid FROM pg_database WHERE datname = current_database())) 
GROUP BY c.relname
```

1869
F.19.3. Authors
Mark Kirkwood <markir@paradise.net.nz>
Design suggestions: Neil Conway <neilc@samurai.com>
Debugging advice: Tom Lane <tgl@sss.pgh.pa.us>

F.20. pgcrypto

The pgcrypto module provides cryptographic functions for PostgreSQL.

F.20.1. General hashing functions

F.20.1.1. digest()


digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea

Computes a binary hash of the given data. type is the algorithm to use. Standard algorithms are md5, sha1, sha224, sha256, sha384 and sha512. If pgcrypto was built with OpenSSL, more algorithms are available, as detailed in Table F-21.

If you want the digest as a hexadecimal string, use encode() on the result. For example:

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
SELECT encode(digest($1, 'sha1'), 'hex')$$ LANGUAGE SQL STRICT IMMUTABLE;
```
F.20.1.2. hmac()

hmac(data text, key text, type text) returns bytea
hmac(data bytea, key text, type text) returns bytea

Calculates hashed MAC for data with key. key, type is the same as in digest().

This is similar to digest() but the hash can only be recalculated knowing the key. This prevents the scenario of someone altering data and also changing the hash to match.

If the key is larger than the hash block size it will first be hashed and the result will be used as key.

F.20.2. Password hashing functions

The functions crypt() and gen_salt() are specifically designed for hashing passwords. crypt() does the hashing and gen_salt() prepares algorithm parameters for it.

The algorithms in crypt() differ from usual hashing algorithms like MD5 or SHA1 in the following respects:

1. They are slow. As the amount of data is so small, this is the only way to make brute-forcing passwords hard.
2. They use a random value, called the salt, so that users having the same password will have different encrypted passwords. This is also an additional defense against reversing the algorithm.
3. They include the algorithm type in the result, so passwords hashed with different algorithms can co-exist.
4. Some of them are adaptive — that means when computers get faster, you can tune the algorithm to be slower, without introducing incompatibility with existing passwords.

Table F-18. Supported algorithms for crypt()

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Max password length</th>
<th>Adaptive?</th>
<th>Salt bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bf</td>
<td>72</td>
<td>yes</td>
<td>128</td>
<td>Blowfish-based, variant 2a</td>
</tr>
<tr>
<td>md5</td>
<td>unlimited</td>
<td>no</td>
<td>48</td>
<td>MD5-based crypt</td>
</tr>
<tr>
<td>xdes</td>
<td>8</td>
<td>yes</td>
<td>24</td>
<td>Extended DES</td>
</tr>
<tr>
<td>des</td>
<td>8</td>
<td>no</td>
<td>12</td>
<td>Original UNIX crypt</td>
</tr>
</tbody>
</table>

F.20.2.1. crypt()

crypt(password text, salt text) returns text

Calculates a crypt(3)-style hash of password. When storing a new password, you need to use gen_salt() to generate a new salt value. To check a password, pass the stored hash value as salt, and test whether the result matches the stored value.

Example of setting a new password:
UPDATE ... SET pshash = crypt(‘new password’, gen_salt(‘md5’));

Example of authentication:

SELECT pshash = crypt(‘entered password’, pshash) FROM ... ;

This returns true if the entered password is correct.

**F.20.2.2. gen_salt()**

```
gen_salt(type text [, iter_count integer ]) returns text
```

Generates a new random salt string for use in crypt(). The salt string also tells crypt() which algorithm to use.

The type parameter specifies the hashing algorithm. The accepted types are: des, xdes, md5 and bf.

The iter_count parameter lets the user specify the iteration count, for algorithms that have one. The higher the count, the more time it takes to hash the password and therefore the more time to break it. Although with too high a count the time to calculate a hash may be several years — which is somewhat impractical. If the iter_count parameter is omitted, the default iteration count is used.

Allowed values for iter_count depend on the algorithm:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Default</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdes</td>
<td>725</td>
<td>1</td>
<td>16777215</td>
</tr>
<tr>
<td>bf</td>
<td>6</td>
<td>4</td>
<td>31</td>
</tr>
</tbody>
</table>

For xdes there is an additional limitation that the iteration count must be an odd number.

To pick an appropriate iteration count, consider that the original DES crypt was designed to have the speed of 4 hashes per second on the hardware of that time. Slower than 4 hashes per second would probably dampen usability. Faster than 100 hashes per second is probably too fast.

Here is a table that gives an overview of the relative slowness of different hashing algorithms. The table shows how much time it would take to try all combinations of characters in an 8-character password, assuming that the password contains either only lowercase letters, or upper- and lower-case letters and numbers. In the crypt-bf entries, the number after a slash is the iter_count parameter of gen_salt.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Hashes/sec</th>
<th>For [a-z]</th>
<th>For [A-Za-z0-9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypt-bf/8</td>
<td>28</td>
<td>246 years</td>
<td>251322 years</td>
</tr>
<tr>
<td>crypt-bf/7</td>
<td>57</td>
<td>121 years</td>
<td>123457 years</td>
</tr>
<tr>
<td>crypt-bf/6</td>
<td>112</td>
<td>62 years</td>
<td>62831 years</td>
</tr>
<tr>
<td>crypt-bf/5</td>
<td>211</td>
<td>33 years</td>
<td>33351 years</td>
</tr>
<tr>
<td>crypt-md5</td>
<td>2681</td>
<td>2.6 years</td>
<td>2625 years</td>
</tr>
<tr>
<td>crypt-des</td>
<td>362837</td>
<td>7 days</td>
<td>19 years</td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Hashes/sec</th>
<th>For [a-z]</th>
<th>For [A-Za-z0-9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>sha1</td>
<td>590223</td>
<td>4 days</td>
<td>12 years</td>
</tr>
<tr>
<td>md5</td>
<td>2345086</td>
<td>1 day</td>
<td>3 years</td>
</tr>
</tbody>
</table>

Notes:

- The machine used is a 1.5GHz Pentium 4.
- crypt-des and crypt-md5 algorithm numbers are taken from John the Ripper v1.6.38 -test output.
- md5 numbers are from mdcrack 1.2.
- sha1 numbers are from lcrack-20031130-beta.
- crypt-bf numbers are taken using a simple program that loops over 1000 8-character passwords. That way I can show the speed with different numbers of iterations. For reference: john -test shows 213 loops/sec for crypt-bf/5. (The very small difference in results is in accordance with the fact that the crypt-bf implementation in pgcrypto is the same one used in John the Ripper.)

Note that “try all combinations” is not a realistic exercise. Usually password cracking is done with the help of dictionaries, which contain both regular words and various mutations of them. So, even somewhat word-like passwords could be cracked much faster than the above numbers suggest, while a 6-character non-word-like password may escape cracking. Or not.

F.20.3. PGP encryption functions

The functions here implement the encryption part of the OpenPGP (RFC 4880) standard. Supported are both symmetric-key and public-key encryption.

An encrypted PGP message consists of 2 parts, or packets:

- Packet containing a session key — either symmetric-key or public-key encrypted.
- Packet containing data encrypted with the session key.

When encrypting with a symmetric key (i.e., a password):

1. The given password is hashed using a String2Key (S2K) algorithm. This is rather similar to crypt() algorithms — purposefully slow and with random salt — but it produces a full-length binary key.
2. If a separate session key is requested, a new random key will be generated. Otherwise the S2K key will be used directly as the session key.
3. If the S2K key is to be used directly, then only S2K settings will be put into the session key packet. Otherwise the session key will be encrypted with the S2K key and put into the session key packet.

When encrypting with a public key:

1. A new random session key is generated.
2. It is encrypted using the public key and put into the session key packet.

In either case the data to be encrypted is processed as follows:
Appendix F. Additional Supplied Modules

1. Optional data-manipulation: compression, conversion to UTF-8, and/or conversion of line-endings.
2. The data is prefixed with a block of random bytes. This is equivalent to using a random IV.
3. An SHA1 hash of the random prefix and data is appended.
4. All this is encrypted with the session key and placed in the data packet.

F.20.3.1. `pgp_sym_encrypt()`

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

Encrypt `data` with a symmetric PGP key `psw`. The `options` parameter can contain option settings, as described below.

F.20.3.2. `pgp_sym_decrypt()`

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

Decrypt a symmetric-key-encrypted PGP message.

Decrypting bytea data with `pgp_sym_decrypt` is disallowed. This is to avoid outputting invalid character data. Decrypting originally textual data with `pgp_sym_decrypt_bytea` is fine.

The `options` parameter can contain option settings, as described below.

F.20.3.3. `pgp_pub_encrypt()`

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

Encrypt `data` with a public PGP key `key`. Giving this function a secret key will produce a error.

The `options` parameter can contain option settings, as described below.

F.20.3.4. `pgp_pub_decrypt()`

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns bytea
```

Decrypt a public-key-encrypted message. `key` must be the secret key corresponding to the public key that was used to encrypt. If the secret key is password-protected, you must give the password in `psw`. If there is no password, but you want to specify options, you need to give an empty password.

Decrypting bytea data with `pgp_pub_decrypt` is disallowed. This is to avoid outputting invalid character data. Decrypting originally textual data with `pgp_pub_decrypt_bytea` is fine.

The `options` parameter can contain option settings, as described below.
F.20.3.5. pgp_key_id()

pgp_key_id(bytea) returns text

pgp_key_id extracts the key ID of a PGP public or secret key. Or it gives the key ID that was used for encrypting the data, if given an encrypted message.

It can return 2 special key IDs:

- **SYMKEY**
  The message is encrypted with a symmetric key.

- **ANYKEY**
  The message is public-key encrypted, but the key ID has been removed. That means you will need to try all your secret keys on it to see which one decrypts it. pgcrypto itself does not produce such messages.

Note that different keys may have the same ID. This is rare but a normal event. The client application should then try to decrypt with each one, to see which fits — like handling ANYKEY.

F.20.3.6. armor(), dearmor()

armor(data bytea) returns text
dearmor(data text) returns bytea

These functions wrap/unwrap binary data into PGP Ascii Armor format, which is basically Base64 with CRC and additional formatting.

F.20.3.7. Options for PGP functions

Options are named to be similar to GnuPG. An option's value should be given after an equal sign; separate options from each other with commas. For example:

```sql
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

All of the options except convert-crlf apply only to encrypt functions. Decrypt functions get the parameters from the PGP data.

The most interesting options are probably compress-algo and unicode-mode. The rest should have reasonable defaults.

F.20.3.7.1. cipher-algo

Which cipher algorithm to use.

Values: bf, aes128, aes192, aes256 (OpenSSL-only: 3des, cast5)
Default: aes128
Applies to: pgp_sym_encrypt, pgp_pub_encrypt
Appendix F. Additional Supplied Modules

F.20.3.7.2. compress-algo
Which compression algorithm to use. Only available if PostgreSQL was built with zlib.

Values:
0 - no compression
1 - ZIP compression
2 - ZLIB compression (= ZIP plus meta-data and block CRCs)
Default: 0
Applies to: pgp_sym_encrypt, pgp_pub_encrypt

F.20.3.7.3. compress-level
How much to compress. Higher levels compress smaller but are slower. 0 disables compression.

Values: 0, 1-9
Default: 6
Applies to: pgp_sym_encrypt, pgp_pub_encrypt

F.20.3.7.4. convert-crlf
Whether to convert \n into \r\n when encrypting and \r\n to \n when decrypting. RFC 4880 specifies that text data should be stored using \r\n line-feeds. Use this to get fully RFC-compliant behavior.

Values: 0, 1
Default: 0
Applies to: pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.20.3.7.5. disable-mdc
Do not protect data with SHA-1. The only good reason to use this option is to achieve compatibility with ancient PGP products, predating the addition of SHA-1 protected packets to RFC 4880. Recent gnupg.org and pgp.com software supports it fine.

Values: 0, 1
Default: 0
Applies to: pgp_sym_encrypt, pgp_pub_encrypt

F.20.3.7.6. enable-session-key
Use separate session key. Public-key encryption always uses a separate session key; this is for symmetric-key encryption, which by default uses the S2K key directly.

Values: 0, 1
Default: 0
Applies to: pgp_sym_encrypt
**F.20.3.7.7. s2k-mode**

Which S2K algorithm to use.

Values:
- 0 - Without salt. Dangerous!
- 1 - With salt but with fixed iteration count.
- 3 - Variable iteration count.

Default: 3

Applies to: pgp_sym_encrypt

**F.20.3.7.8. s2k-digest-algo**

Which digest algorithm to use in S2K calculation.

Values: md5, sha1

Default: sha1

Applies to: pgp_sym_encrypt

**F.20.3.7.9. s2k-cipher-algo**

Which cipher to use for encrypting separate session key.

Values: bf, aes, aes128, aes192, aes256

Default: use cipher-algo

Applies to: pgp_sym_encrypt

**F.20.3.7.10. unicode-mode**

Whether to convert textual data from database internal encoding to UTF-8 and back. If your database already is UTF-8, no conversion will be done, but the message will be tagged as UTF-8. Without this option it will not be.

Values: 0, 1

Default: 0

Applies to: pgp_sym_encrypt, pgp_pub_encrypt

**F.20.3.8. Generating PGP keys with GnuPG**

To generate a new key:

```bash
gpg --gen-key
```

The preferred key type is “DSA and Elgamal”.

For RSA encryption you must create either DSA or RSA sign-only key as master and then add an RSA encryption subkey with `gpg --edit-key`.
Appendix F. Additional Supplied Modules

To list keys:

```
gpg --list-secret-keys
```

To export a public key in ascii-armor format:

```
gpg -a --export KEYID > public.key
```

To export a secret key in ascii-armor format:

```
gpg -a --export-secret-keys KEYID > secret.key
```

You need to use `dearmor()` on these keys before giving them to the PGP functions. Or if you can handle binary data, you can drop `-a` from the command.


F.20.3.9. Limitations of PGP code

- No support for signing. That also means that it is not checked whether the encryption subkey belongs to the master key.
- No support for encryption key as master key. As such practice is generally discouraged, this should not be a problem.
- No support for several subkeys. This may seem like a problem, as this is common practice. On the other hand, you should not use your regular GPG/PGP keys with `pgcrypto`, but create new ones, as the usage scenario is rather different.

F.20.4. Raw encryption functions

These functions only run a cipher over data; they don’t have any advanced features of PGP encryption. Therefore they have some major problems:

1. They use user key directly as cipher key.
2. They don’t provide any integrity checking, to see if the encrypted data was modified.
3. They expect that users manage all encryption parameters themselves, even IV.
4. They don’t handle text.

So, with the introduction of PGP encryption, usage of raw encryption functions is discouraged.

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

Encrypt/decrypt data using the cipher method specified by `type`. The syntax of the `type` string is:

```
algorithm [ - mode ] [ /pad: padding ]
```

where `algorithm` is one of:

- `bf` — Blowfish
- `aes` — AES (Rijndael-128)

and `mode` is one of:

- `cbc` — next block depends on previous (default)
- `ecb` — each block is encrypted separately (for testing only)

and `padding` is one of:

- `pkcs` — data may be any length (default)
- `none` — data must be multiple of cipher block size

So, for example, these are equivalent:

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

In `encrypt_iv` and `decrypt_iv`, the `iv` parameter is the initial value for the CBC mode; it is ignored for ECB. It is clipped or padded with zeroes if not exactly block size. It defaults to all zeroes in the functions without this parameter.

**F.20.5. Random-data functions**

```
gen_random_bytes(count integer) returns bytea
```

Returns `count` cryptographically strong random bytes. At most 1024 bytes can be extracted at a time. This is to avoid draining the randomness generator pool.

**F.20.6. Notes**

**F.20.6.1. Configuration**

`pgcrypto` configures itself according to the findings of the main PostgreSQL `configure` script. The options that affect it are `--with-zlib` and `--with-openssl`.

When compiled with zlib, PGP encryption functions are able to compress data before encrypting.

When compiled with OpenSSL, there will be more algorithms available. Also public-key encryption functions will be faster as OpenSSL has more optimized BIGNUM functions.

**Table F-21. Summary of functionality with and without OpenSSL**
### Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Built-in</th>
<th>With OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SHA1</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SHA224/256/384/512</td>
<td>yes</td>
<td>yes (Note 1)</td>
</tr>
<tr>
<td>Other digest algorithms</td>
<td>no</td>
<td>yes (Note 2)</td>
</tr>
<tr>
<td>Blowfish</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>AES</td>
<td>yes</td>
<td>yes (Note 3)</td>
</tr>
<tr>
<td>DES/3DES/CAST5</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Raw encryption</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>PGP Symmetric encryption</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>PGP Public-Key encryption</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. SHA2 algorithms were added to OpenSSL in version 0.9.8. For older versions, `pgcrypto` will use built-in code.
2. Any digest algorithm OpenSSL supports is automatically picked up. This is not possible with ciphers, which need to be supported explicitly.
3. AES is included in OpenSSL since version 0.9.7. For older versions, `pgcrypto` will use built-in code.

### F.20.6.2. NULL handling

As is standard in SQL, all functions return NULL, if any of the arguments are NULL. This may create security risks on careless usage.

### F.20.6.3. Security limitations

All `pgcrypto` functions run inside the database server. That means that all the data and passwords move between `pgcrypto` and client applications in clear text. Thus you must:

1. Connect locally or use SSL connections.
2. Trust both system and database administrator.

If you cannot, then better do crypto inside client application.

### F.20.6.4. Useful reading

  The GNU Privacy Handbook.
  Describes the crypt-blowfish algorithm.
  How to choose a good password.
Appendix F. Additional Supplied Modules

  Interesting idea for picking passwords.
- [http://www.interhack.net/people/cmcurtin/snake-oil-faq.html](http://www.interhack.net/people/cmcurtin/snake-oil-faq.html)
  Describes good and bad cryptography.

F.20.6.5. Technical references

  OpenPGP message format.
  The MD5 Message-Digest Algorithm.
  HMAC: Keyed-Hashing for Message Authentication.
  Comparison of crypt-des, crypt-md5 and bcrypt algorithms.
  Standards for DES, 3DES and AES.
  Description of Fortuna CSPRNG.
- [http://jlcooke.ca/random/](http://jlcooke.ca/random/)
  Jean-Luc Cooke Fortuna-based /dev/random driver for Linux.
  Collection of cryptology pointers.

F.20.7. Author

Marko Kreen <markokr@gmail.com>

pgcrypto uses code from the following sources:

Table F-22. Credits

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Author</th>
<th>Source origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES crypt</td>
<td>David Burren and others</td>
<td>FreeBSD libcrypt</td>
</tr>
<tr>
<td>MD5 crypt</td>
<td>Poul-Henning Kamp</td>
<td>FreeBSD libcrypt</td>
</tr>
<tr>
<td>Blowfish crypt</td>
<td>Solar Designer</td>
<td><a href="http://www.openwall.com">www.openwall.com</a></td>
</tr>
<tr>
<td>Blowfish cipher</td>
<td>Simon Tatham</td>
<td>PuTTY</td>
</tr>
<tr>
<td>Rijndael cipher</td>
<td>Brian Gladman</td>
<td>OpenBSD sys/crypto</td>
</tr>
<tr>
<td>MD5 and SHA1</td>
<td>WIDE Project</td>
<td>KAME kame/sys/crypto</td>
</tr>
<tr>
<td>SHA256/384/512</td>
<td>Aaron D. Gifford</td>
<td>OpenBSD sys/crypto</td>
</tr>
</tbody>
</table>
F.21. pg_freespacemap

The pg_freespacemap module provides a means for examining the free space map (FSM). It provides two C functions: `pg_freespacemap_relations` and `pg_freespacemap_pages` that each return a set of records, plus two views `pg_freespacemap_relations` and `pg_freespacemap_pages` that wrap the functions for convenient use.

By default public access is revoked from the functions and views, just in case there are security issues lurking.

F.21.1. The pg_freespacemap views

The definitions of the columns exposed by the views are:

**Table F-23. pg_freespacemap_relations Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reltablespace</td>
<td>oid</td>
<td>pg_tablespace.oid</td>
<td>Tablespace OID of the relation</td>
</tr>
<tr>
<td>reldatabase</td>
<td>oid</td>
<td>pg_database.oid</td>
<td>Database OID of the relation</td>
</tr>
<tr>
<td>relfilenode</td>
<td>oid</td>
<td>pg_class.relfilenode</td>
<td>Relfilenode of the relation</td>
</tr>
<tr>
<td>avgrequest</td>
<td>integer</td>
<td></td>
<td>Moving average of free space requests (NULL for indexes)</td>
</tr>
<tr>
<td>interestingpages</td>
<td>integer</td>
<td></td>
<td>Count of pages last reported as containing useful free space</td>
</tr>
<tr>
<td>storedpages</td>
<td>integer</td>
<td></td>
<td>Count of pages actually stored in free space map</td>
</tr>
<tr>
<td>nextpage</td>
<td>integer</td>
<td></td>
<td>Page index (from 0) to start next search at</td>
</tr>
</tbody>
</table>

**Table F-24. pg_freespacemap_pages Columns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reltablespace</td>
<td>oid</td>
<td>pg_tablespace.oid</td>
<td>Tablespace OID of the relation</td>
</tr>
<tr>
<td>reldatabase</td>
<td>oid</td>
<td>pg_database.oid</td>
<td>Database OID of the relation</td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relfilenode</td>
<td>oid</td>
<td>pg_class.relfilenode</td>
<td>Relfilenode of the relation</td>
</tr>
<tr>
<td>relblocknumber</td>
<td>bigint</td>
<td></td>
<td>Page number within the relation</td>
</tr>
<tr>
<td>bytes</td>
<td>integer</td>
<td></td>
<td>Free bytes in the page, or NULL for an index page (see below)</td>
</tr>
</tbody>
</table>

For `pg_freespacemap_relations`, there is one row for each relation in the free space map. `storedpages` is the number of pages actually stored in the map, while `interestingpages` is the number of pages the last VACUUM thought had useful amounts of free space.

If `storedpages` is consistently less than `interestingpages` then it’d be a good idea to increase `max_fsm_pages`. Also, if the number of rows in `pg_freespacemap_relations` is close to `max_fsm_relations`, then you should consider increasing `max_fsm_relations`.

For `pg_freespacemap_pages`, there is one row for each page in the free space map. The number of rows for a relation will match the `storedpages` column in `pg_freespacemap_relations`.

For indexes, what is tracked is entirely-unused pages, rather than free space within pages. Therefore, the average request size and free bytes within a page are not meaningful, and are shown as NULL.

Because the map is shared by all the databases, there will normally be entries for relations not belonging to the current database. This means that there may not be matching join rows in `pg_class` for some rows, or that there could even be incorrect joins. If you are trying to join against `pg_class`, it’s a good idea to restrict the join to rows having `reldatabase` equal to the current database’s OID or zero.

When either of the views is accessed, internal free space map locks are taken for long enough to copy all the state data that the view will display. This ensures that the views produce a consistent set of results, while not blocking normal activity longer than necessary. Nonetheless there could be some impact on database performance if they are read often.

F.21.2. Sample output

```
regression=# SELECT c.relname, r.avgrequest, r.interestingpages, r.storedpages
FROM pg_freespacemap_relations r INNER JOIN pg_class c
ON r.relfilenode = c.relfilenode AND
r.reldatabase IN (0, (SELECT oid FROM pg_database
WHERE datname = current_database()))
ORDER BY r.storedpages DESC LIMIT 10;
```

<table>
<thead>
<tr>
<th>relname</th>
<th>avgrequest</th>
<th>interestingpages</th>
<th>storedpages</th>
</tr>
</thead>
<tbody>
<tr>
<td>onek</td>
<td>256</td>
<td>109</td>
<td>109</td>
</tr>
<tr>
<td>pg_attribute</td>
<td>167</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>pg_class</td>
<td>191</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td>pg_attribute_relid_attnam_index</td>
<td></td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>onek2</td>
<td>256</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>pg_depend</td>
<td>95</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>pg_type</td>
<td>199</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>pg_rewrite</td>
<td>1011</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>pg_class_relname_nsp_index</td>
<td></td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>pg_proc</td>
<td>302</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

(10 rows)
Appendix F. Additional Supplied Modules

regression=# SELECT c.relname, p.relblocknumber, p.bytes
FROM pg_freespacemap_pages p INNER JOIN pg_class c
ON p.relfilenode = c.relfilenode AND
p.reldatabase IN (0, (SELECT oid FROM pg_database
WHERE datname = current_database()))
ORDER BY c.relname LIMIT 10;

<table>
<thead>
<tr>
<th>relname</th>
<th>relblocknumber</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_star</td>
<td>0</td>
<td>8040</td>
</tr>
<tr>
<td>abstime_tbl</td>
<td>0</td>
<td>7908</td>
</tr>
<tr>
<td>aggtest</td>
<td>0</td>
<td>8008</td>
</tr>
<tr>
<td>altinhoid</td>
<td>0</td>
<td>8128</td>
</tr>
<tr>
<td>altstartwith</td>
<td>0</td>
<td>8128</td>
</tr>
<tr>
<td>arrtest</td>
<td>0</td>
<td>7172</td>
</tr>
<tr>
<td>b_star</td>
<td>0</td>
<td>7976</td>
</tr>
<tr>
<td>box_tbl</td>
<td>0</td>
<td>7912</td>
</tr>
<tr>
<td>bt_f8_heap</td>
<td>54</td>
<td>7728</td>
</tr>
<tr>
<td>bt_i4_heap</td>
<td>49</td>
<td>8008</td>
</tr>
</tbody>
</table>

(10 rows)

F.21.3. Author

Mark Kirkwood <markir@paradise.net.nz>

F.22. pgrowlocks

The pgrowlocks module provides a function to show row locking information for a specified table.

F.22.1. Overview

pgrowlocks(text) returns setof record

The parameter is the name of a table. The result is a set of records, with one row for each locked row within the table. The output columns are:

Table F-25. pgrowlocks output columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locked_row</td>
<td>tid</td>
<td>Tuple ID (TID) of locked row</td>
</tr>
<tr>
<td>lock_type</td>
<td>text</td>
<td>Shared for shared lock, or Exclusive for exclusive lock</td>
</tr>
<tr>
<td>locker</td>
<td>xid</td>
<td>Transaction ID of locker, or multixact ID if multi-transaction</td>
</tr>
</tbody>
</table>
### Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi</td>
<td>boolean</td>
<td>True if locker is a multi-transaction</td>
</tr>
<tr>
<td>xids</td>
<td>xid[]</td>
<td>Transaction IDs of lockers (more than one if multi-transaction)</td>
</tr>
<tr>
<td>pids</td>
<td>integer[]</td>
<td>Process IDs of locking backends (more than one if multi-transaction)</td>
</tr>
</tbody>
</table>

`pgrowlocks` takes `AccessShareLock` for the target table and reads each row one by one to collect the row locking information. This is not very speedy for a large table. Note that:

1. If the table as a whole is exclusive-locked by someone else, `pgrowlocks` will be blocked.
2. `pgrowlocks` is not guaranteed to produce a self-consistent snapshot. It is possible that a new row lock is taken, or an old lock is freed, during its execution.

`pgrowlocks` does not show the contents of locked rows. If you want to take a look at the row contents at the same time, you could do something like this:

```sql
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
WHERE p.locked_row = a.ctid;
```

Be aware however that (as of PostgreSQL 8.3) such a query will be very inefficient.

### F.22.2. Sample output

```
test=# SELECT * FROM pgrowlocks('t1');
 locked_row | lock_type | locker | multi | xids | pids  
-----------------+-----------+--------+-------+-----------+---------------
(0,1) | Shared | 19 | t | {804,805} | {29066,29068}  
(0,2) | Shared | 19 | t | {804,805} | {29066,29068}  
(0,3) | Exclusive | 804 | f | {804} | {29066}  
(0,4) | Exclusive | 804 | f | {804} | {29066}  
(4 rows)
```

### F.22.3. Author

Tatsuo Ishii

### F.23. pg_standby

`pg_standby` supports creation of a “warm standby” database server. It is designed to be a production-ready program, as well as a customizable template should you require specific modifications.
pg_standby is designed to be a waiting restore_command, which is needed to turn a standard archive recovery into a warm standby operation. Other configuration is required as well, all of which is described in the main server manual (see Section 24.4).

pg_standby features include:

- Supports copy or link for restoring WAL files
- Written in C, so very portable and easy to install
- Easy-to-modify source code, with specifically designated sections to modify for your own needs
- Already tested on Linux and Windows

F.23.1. Usage

To configure a standby server to use pg_standby, put this into its recovery.conf configuration file:

```bash
restore_command = 'pg_standby archiveDir %f %p %r'
```

where `archiveDir` is the directory from which WAL segment files should be restored.

The full syntax of pg_standby’s command line is

```bash
pg_standby [ option ... ] archivelocation nextwalfile xlogfilepath [ restartwalfile ]
```

When used within `restore_command`, the `%f` and `%p` macros should be specified for `nextwalfile` and `xlogfilepath` respectively, to provide the actual file and path required for the restore.

If `restartwalfile` is specified, normally by using the `%r` macro, then all WAL files logically preceding this file will be removed from `archivelocation`. This minimizes the number of files that need to be retained, while preserving crash-restart capability. Use of this parameter is appropriate if the `archivelocation` is a transient staging area for this particular standby server, but not when the `archivelocation` is intended as a long-term WAL archive area.

pg_standby assumes that `archivelocation` is a directory readable by the server-owning user. If `restartwalfile` (or `-k`) is specified, the `archivelocation` directory must be writable too.

Table F-26. pg_standby options

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-c</code></td>
<td>yes</td>
<td>Use <code>cp</code> or <code>copy</code> command to restore WAL files from archive.</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>no</td>
<td>Print lots of debug logging output on stderr.</td>
</tr>
<tr>
<td>Option</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-k numfiles</td>
<td>0</td>
<td>Remove files from <code>archivelocation</code> so that no more than this many WAL files before the current one are kept in the archive. Zero (the default) means not to remove any files from <code>archivelocation</code>. This parameter will be silently ignored if <code>restartwalfile</code> is specified, since that specification method is more accurate in determining the correct archive cut-off point. Use of this parameter is deprecated as of PostgreSQL 8.3; it is safer and more efficient to specify a <code>restartwalfile</code> parameter. A too small setting could result in removal of files that are still needed for a restart of the standby server, while a too large setting wastes archive space.</td>
</tr>
<tr>
<td>-l</td>
<td>no</td>
<td>Use <code>ln</code> command to restore WAL files from archive. Link is more efficient than copy, but the default is copy since link will not work in all scenarios. On Windows, this option uses the <code>mklink</code> command to provide a file-to-file symbolic link. <code>-l</code> will not work on versions of Windows prior to Vista.</td>
</tr>
<tr>
<td>-r maxretries</td>
<td>3</td>
<td>Set the maximum number of times to retry the copy or link command if it fails. After each failure, we wait for <code>sleeptime * num_retries</code> so that the wait time increases progressively. So by default, we will wait 5 secs, 10 secs, then 15 secs before reporting the failure back to the standby server. This will be interpreted as end of recovery and the standby will come up fully as a result.</td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-s sleeptime</td>
<td>5</td>
<td>Set the number of seconds (up to 60) to sleep between tests to see if the WAL file to be restored is available in the archive yet. The default setting is not necessarily recommended; consult Section 24.4 for discussion.</td>
</tr>
<tr>
<td>-t triggerfile</td>
<td>none</td>
<td>Specify a trigger file whose presence should cause recovery to end whether or not the next WAL file is available. It is recommended that you use a structured filename to avoid confusion as to which server is being triggered when multiple servers exist on the same system; for example /tmp/pgsql.trigger.5432.</td>
</tr>
<tr>
<td>-w maxwaittime</td>
<td>0</td>
<td>Set the maximum number of seconds to wait for the next WAL file, after which recovery will end and the standby will come up. A setting of zero (the default) means wait forever. The default setting is not necessarily recommended; consult Section 24.4 for discussion.</td>
</tr>
</tbody>
</table>

**Caution**

It is critical the trigger file be created with permissions allowing the postgres process to remove the file. Generally this is best done by creating the file from the postgres user account. Failure to do so will prevent completion of WAL file recovery and the server from coming back online successfully.

**Note:** `--help` is not supported since `pg_standby` is not intended for interactive use, except during development and testing.

**F.23.2. Examples**

On Linux or Unix systems, you might use:

```bash
archive_command = 'cp %p .../archive/%f'
```
restore_command = 'pg_standby -l -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f %p %r 2>>standby.log'

where the archive directory is physically located on the standby server, so that the archive_command is accessing it across NFS, but the files are local to the standby (enabling use of ln). This will:

• use the ln command to restore WAL files from archive
• produce debugging output in standby.log
• sleep for 2 seconds between checks for next WAL file availability
• stop waiting only when a trigger file called /tmp/pgsql.trigger.5442 appears
• remove no-longer-needed files from the archive directory

On Windows, you might use:
archive_command = 'copy %p ...\archive\%f'

restore_command = 'pg_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p %r 2>>standby.log'

Note that backslashes need to be doubled in the archive_command, but not in the restore_command. This will:

• use the copy command to restore WAL files from archive
• produce debugging output in standby.log
• sleep for 5 seconds between checks for next WAL file availability
• stop waiting only when a trigger file called C:\pgsql.trigger.5442 appears
• remove no-longer-needed files from the archive directory

Since the Windows example uses copy at both ends, either or both servers might be accessing the archive directory across the network.

F.23.3. Supported server versions

pg_standby is designed to work with PostgreSQL 8.2 and later.

PostgreSQL 8.3 provides the %r macro, which is designed to let pg_standby know the last file it needs to keep. With PostgreSQL 8.2, the -k option must be used if archive cleanup is required. This option remains available in 8.3, but its use is deprecated.

F.23.4. Author

Simon Riggs <simon@2ndquadrant.com>

F.24. pgstattpuple

The pgstattpuple module provides various functions to obtain tuple-level statistics.
F.24.1. Functions

pgstattuple(text) returns record

pgstattuple returns a relation’s physical length, percentage of “dead” tuples, and other info. This may help users to determine whether vacuum is necessary or not. The argument is the target relation’s name (optionally schema-qualified). For example:

test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
- [ RECORD 1 ]-------
table_len | 458752
tuple_count | 1470
tuple_len | 438896
tuple_percent | 95.67
dead_tuple_count | 11
dead_tuple_len | 3157
dead_tuple_percent | 0.69
free_space | 8932
free_percent | 1.95

The output columns are:

Table F-27. pgstattuple output columns

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_len</td>
<td>bigint</td>
<td>Physical relation length in bytes</td>
</tr>
<tr>
<td>tuple_count</td>
<td>bigint</td>
<td>Number of live tuples</td>
</tr>
<tr>
<td>tuple_len</td>
<td>bigint</td>
<td>Total length of live tuples in bytes</td>
</tr>
<tr>
<td>tuple_percent</td>
<td>float8</td>
<td>Percentage of live tuples</td>
</tr>
<tr>
<td>dead_tuple_count</td>
<td>bigint</td>
<td>Number of dead tuples</td>
</tr>
<tr>
<td>dead_tuple_len</td>
<td>bigint</td>
<td>Total length of dead tuples in bytes</td>
</tr>
<tr>
<td>dead_tuple_percent</td>
<td>float8</td>
<td>Percentage of dead tuples</td>
</tr>
<tr>
<td>free_space</td>
<td>bigint</td>
<td>Total free space in bytes</td>
</tr>
<tr>
<td>free_percent</td>
<td>float8</td>
<td>Percentage of free space</td>
</tr>
</tbody>
</table>

pgstattuple acquires only a read lock on the relation. So the results do not reflect an instantaneous snapshot; concurrent updates will affect them.

pgstattuple judges a tuple is “dead” if HeapTupleSatisfiesNow returns false.

pgstattuple(oid) returns record

This is the same as pgstattuple(text), except that the target relation is specified by OID.

pgstatindex(text) returns record

pgstatindex returns a record showing information about a btree index. For example:

test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
- [ RECORD 1 ]--------
version | 2
tree_level | 0
index_size | 8192
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>integer</td>
<td>Btree version number</td>
</tr>
<tr>
<td>tree_level</td>
<td>integer</td>
<td>Tree level of the root page</td>
</tr>
<tr>
<td>index_size</td>
<td>integer</td>
<td>Total number of pages in index</td>
</tr>
<tr>
<td>root_block_no</td>
<td>integer</td>
<td>Location of root block</td>
</tr>
<tr>
<td>internal_pages</td>
<td>integer</td>
<td>Number of “internal” (upper-level) pages</td>
</tr>
<tr>
<td>leaf_pages</td>
<td>integer</td>
<td>Number of leaf pages</td>
</tr>
<tr>
<td>empty_pages</td>
<td>integer</td>
<td>Number of empty pages</td>
</tr>
<tr>
<td>deleted_pages</td>
<td>integer</td>
<td>Number of deleted pages</td>
</tr>
<tr>
<td>avg_leaf_density</td>
<td>float8</td>
<td>Average density of leaf pages</td>
</tr>
<tr>
<td>leaf_fragmentation</td>
<td>float8</td>
<td>Leaf page fragmentation</td>
</tr>
</tbody>
</table>

As with pgstattuple, the results are accumulated page-by-page, and should not be expected to represent an instantaneous snapshot of the whole index.

```
pg_relpages(text) returns integer
```

`pg_relpages` returns the number of pages in the relation.

### F.24.2. Author

Tatsuo Ishii

### F.25. pg_trgm

The `pg_trgm` module provides functions and operators for determining the similarity of text based on trigram matching, as well as index operator classes that support fast searching for similar strings.

#### F.25.1. Trigram (or Trigraph) Concepts

A trigram is a group of three consecutive characters taken from a string. We can measure the similarity of two strings by counting the number of trigrams they share. This simple idea turns out to be very effective for measuring the similarity of words in many natural languages.
Appendix F. Additional Supplied Modules

Note: A string is considered to have two spaces prefixed and one space suffixed when determining the set of trigrams contained in the string. For example, the set of trigrams in the string “cat” is “c”, “ca”, “cat”, and “at”.

F.25.2. Functions and Operators

Table F-29. pg_trgm functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>similarity(text, text)</td>
<td>real</td>
<td>Returns a number that indicates how similar the two arguments are. The range of the result is zero (indicating that the two strings are completely dissimilar) to one (indicating that the two strings are identical).</td>
</tr>
<tr>
<td>show_trgm(text)</td>
<td>text[]</td>
<td>Returns an array of all the trigrams in the given string. (In practice this is seldom useful except for debugging.)</td>
</tr>
<tr>
<td>show_limit()</td>
<td>real</td>
<td>Returns the current similarity threshold used by the % operator. This sets the minimum similarity between two words for them to be considered similar enough to be misspellings of each other, for example.</td>
</tr>
<tr>
<td>set_limit(real)</td>
<td>real</td>
<td>Sets the current similarity threshold that is used by the % operator. The threshold must be between 0 and 1 (default is 0.3). Returns the same value passed in.</td>
</tr>
</tbody>
</table>

Table F-30. pg_trgm operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text % text</td>
<td>boolean</td>
<td>Returns true if its arguments have a similarity that is greater than the current similarity threshold set by set_limit.</td>
</tr>
</tbody>
</table>
F.25.3. Index Support

The `pg_trgm` module provides GiST and GIN index operator classes that allow you to create an index over a text column for the purpose of very fast similarity searches. These index types support the `%` similarity operator (and no other operators, so you may want a regular btree index too).

Example:

```sql
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING gist (t gist_trgm_ops);
```

or

```sql
CREATE INDEX trgm_idx ON test_trgm USING gin (t gin_trgm_ops);
```

At this point, you will have an index on the `t` column that you can use for similarity searching. A typical query is

```sql
SELECT t, similarity(t, 'word') AS sml
FROM test_trgm
WHERE t % 'word'
ORDER BY sml DESC, t;
```

This will return all values in the text column that are sufficiently similar to `word`, sorted from best match to worst. The index will be used to make this a fast operation even over very large data sets.

The choice between GiST and GIN indexing depends on the relative performance characteristics of GiST and GIN, which are discussed elsewhere. As a rule of thumb, a GIN index is faster to search than a GiST index, but slower to build or update; so GIN is better suited for static data and GiST for often-updated data.

F.25.4. Text Search Integration

Trigram matching is a very useful tool when used in conjunction with a full text index. In particular it can help to recognize misspelled input words that will not be matched directly by the full text search mechanism.

The first step is to generate an auxiliary table containing all the unique words in the documents:

```sql
CREATE TABLE words AS SELECT word FROM
  ts_stat('SELECT to_tsvector("simple", bodytext) FROM documents');
```

where `documents` is a table that has a text field `bodytext` that we wish to search. The reason for using the `simple` configuration with the `to_tsvector` function, instead of using a language-specific configuration, is that we want a list of the original (unstemmed) words.

Next, create a trigram index on the word column:

```sql
CREATE INDEX words_idx ON words USING gin(word gin_trgm_ops);
```
Appendix F. Additional Supplied Modules

Now, a SELECT query similar to the previous example can be used to suggest spellings for misspelled words in user search terms. A useful extra test is to require that the selected words are also of similar length to the misspelled word.

Note: Since the words table has been generated as a separate, static table, it will need to be periodically regenerated so that it remains reasonably up-to-date with the document collection. Keeping it exactly current is usually unnecessary.

F.25.5. References

GiST Development Site http://www.sai.msu.su/~megera/postgres/gist/
Tsearch2 Development Site http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.25.6. Authors

Oleg Bartunov <oleg@sai.msu.su>, Moscow, Moscow University, Russia
Teodor Sigaev <teodor@sigaev.ru>, Moscow, Delta-Soft Ltd., Russia
Documentation: Christopher Kings-Lynne
This module is sponsored by Delta-Soft Ltd., Moscow, Russia.

F.26. seg

This module implements a data type seg for representing line segments, or floating point intervals. seg can represent uncertainty in the interval endpoints, making it especially useful for representing laboratory measurements.

F.26.1. Rationale

The geometry of measurements is usually more complex than that of a point in a numeric continuum. A measurement is usually a segment of that continuum with somewhat fuzzy limits. The measurements come out as intervals because of uncertainty and randomness, as well as because the value being measured may naturally be an interval indicating some condition, such as the temperature range of stability of a protein.

Using just common sense, it appears more convenient to store such data as intervals, rather than pairs of numbers. In practice, it even turns out more efficient in most applications.

Further along the line of common sense, the fuzziness of the limits suggests that the use of traditional numeric data types leads to a certain loss of information. Consider this: your instrument reads 6.50, and you input this reading into the database. What do you get when you fetch it? Watch:

test=> select 6.50 :: float8 as "pH";
pH
---
6.5
(1 row)
In the world of measurements, 6.50 is not the same as 6.5. It may sometimes be critically different. The experimenters usually write down (and publish) the digits they trust. 6.50 is actually a fuzzy interval contained within a bigger and even fuzzier interval, 6.5, with their center points being (probably) the only common feature they share. We definitely do not want such different data items to appear the same.

Conclusion? It is nice to have a special data type that can record the limits of an interval with arbitrarily variable precision. Variable in the sense that each data element records its own precision.

Check this out:

```sql
test=> select '6.25 .. 6.50'::seg as "pH";
pH
-------------
   6.25 .. 6.50
(1 row)
```

### F.26.2. Syntax

The external representation of an interval is formed using one or two floating point numbers joined by the range operator (.. or ...). Alternatively, it can be specified as a center point plus or minus a deviation. Optional certainty indicators (<, > and ~) can be stored as well. (Certainty indicators are ignored by all the built-in operators, however.)

In the following table, \( x, y \), and \( \text{delta} \) denote floating-point numbers. \( x \) and \( y \), but not \( \text{delta} \), can be preceded by a certainty indicator:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( \text{delta} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table F-31. seg external representations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x .. y )</td>
</tr>
<tr>
<td>( x \text{~}(+)\text{delta} )</td>
</tr>
<tr>
<td>( x .. )</td>
</tr>
<tr>
<td>.. ( x )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table F-32. Examples of valid seg input</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 5.0 )</td>
</tr>
<tr>
<td>( \sim 5.0 )</td>
</tr>
<tr>
<td>( &lt; 5.0 )</td>
</tr>
<tr>
<td>( &gt; 5.0 )</td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5(+-)0.3</td>
<td>Creates an interval 4.7 .. 5.3. Note that the (+-) notation isn’t preserved.</td>
</tr>
<tr>
<td>50 ..</td>
<td>Everything that is greater than or equal to 50</td>
</tr>
<tr>
<td>.. 0</td>
<td>Everything that is less than or equal to 0</td>
</tr>
<tr>
<td>1.5e-2 .. 2E-2</td>
<td>Creates an interval 0.015 .. 0.02</td>
</tr>
<tr>
<td>1 ... 2</td>
<td>The same as 1...2, or 1 .. 2, or 1..2 (spaces around the range operator are ignored)</td>
</tr>
</tbody>
</table>

Because ... is widely used in data sources, it is allowed as an alternative spelling of ... Unfortunately, this creates a parsing ambiguity: it is not clear whether the upper bound in 0...23 is meant to be 23 or 0.23. This is resolved by requiring at least one digit before the decimal point in all numbers in seg input.

As a sanity check, seg rejects intervals with the lower bound greater than the upper, for example 5 .. 2.

F.26.3. Precision

seg values are stored internally as pairs of 32-bit floating point numbers. This means that numbers with more than 7 significant digits will be truncated.

Numbers with 7 or fewer significant digits retain their original precision. That is, if your query returns 0.00, you will be sure that the trailing zeroes are not the artifacts of formatting: they reflect the precision of the original data. The number of leading zeroes does not affect precision: the value 0.0067 is considered to have just 2 significant digits.

F.26.4. Usage

The seg module includes a GiST index operator class for seg values. The operators supported by the GiST opclass include:

- [a, b] << [c, d] Is left of [a, b] is entirely to the left of [c, d]. That is, [a, b] << [c, d] is true if b < c and false otherwise
- [a, b] >> [c, d] Is right of [a, b] is entirely to the right of [c, d]. That is, [a, b] >> [c, d] is true if a > d and false otherwise
- [a, b] &< [c, d] Overlaps or is left of [a, b] is entirely to the right of [c, d]. That is, [a, b] &< [c, d] is true if b < c and false otherwise
- [a, b] &> [c, d] Overlaps or is right of [a, b] is entirely to the left of [c, d]. That is, [a, b] &> [c, d] is true if a > d and false otherwise
- [a, b] = [c, d] Same as The segments [a, b] and [c, d] are identical, that is, a = c and b = d
- [a, b] & [c, d] Overlaps The segments [a, b] and [c, d] overlap.
- [a, b] @> [c, d] Contains The segment [a, b] contains the segment [c, d], that is, a <= c and b >= d
Appendix F. Additional Supplied Modules

- \([a, b] <@ [c, d]\) Contained in

  The segment \([a, b]\) is contained in \([c, d]\), that is, \(a >= c\) and \(b <= d\)

  (Before PostgreSQL 8.2, the containment operators @> and @@ were respectively called @ and ~. These names are still available, but are deprecated and will eventually be retired. Notice that the old names are reversed from the convention formerly followed by the core geometric datatypes!)

  The standard B-tree operators are also provided, for example

  \[
  \begin{align*}
  [a, b] < [c, d] & \quad \text{Less than} \\
  [a, b] > [c, d] & \quad \text{Greater than}
  \end{align*}
  \]

  These operators do not make a lot of sense for any practical purpose but sorting. These operators first compare \((a)\) to \((c)\), and if these are equal, compare \((b)\) to \((d)\). That results in reasonably good sorting in most cases, which is useful if you want to use ORDER BY with this type.

F.26.5. Notes

For examples of usage, see the regression test sql/seg.sql.

The mechanism that converts \((+-)\) to regular ranges isn’t completely accurate in determining the number of significant digits for the boundaries. For example, it adds an extra digit to the lower boundary if the resulting interval includes a power of ten:

```sql
postgres=> select '10(+-)1'::seg as seg;
    seg
----------
 9.0 .. 11  -- should be: 9 .. 11
```

The performance of an R-tree index can largely depend on the initial order of input values. It may be very helpful to sort the input table on the seg column; see the script sort-segments.pl for an example.

F.26.6. Credits

Original author: Gene Selkov, Jr. <selkovjr@mcs.anl.gov>, Mathematics and Computer Science Division, Argonne National Laboratory.

My thanks are primarily to Prof. Joe Hellerstein (http://db.cs.berkeley.edu/~jmh/) for elucidating the gist of the GiST (http://gist.cs.berkeley.edu/). I am also grateful to all Postgres developers, present and past, for enabling myself to create my own world and live undisturbed in it. And I would like to acknowledge my gratitude to Argonne Lab and to the U.S. Department of Energy for the years of faithful support of my database research.

F.27. spi

The contrib/spi module provides several workable examples of using SPI and triggers. While these functions are of some value in their own right, they are even more useful as examples to modify
for your own purposes. The functions are general enough to be used with any table, but you have to specify table and field names (as described below) while creating a trigger.

**F.27.1. refint.c — functions for implementing referential integrity**

*check_primary_key()* and *check_foreign_key()* are used to check foreign key constraints. (This functionality is long since superseded by the built-in foreign key mechanism, of course, but the module is still useful as an example.)

*check_primary_key()* checks the referencing table. To use, create a *BEFORE INSERT OR UPDATE* trigger using this function on a table referencing another table. Specify as the trigger arguments: the referencing table’s column name(s) which form the foreign key, the referenced table name, and the column names in the referenced table which form the primary/unique key. To handle multiple foreign keys, create a trigger for each reference.

*check_foreign_key()* checks the referenced table. To use, create a *BEFORE DELETE OR UPDATE* trigger using this function on a table referenced by other table(s). Specify as the trigger arguments: the number of referencing tables for which the function has to perform checking, the action if a referencing key is found (cascade — to delete the referencing row, restrict — to abort transaction if referencing keys exist, setnull — to set referencing key fields to null), the triggered table’s column names which form the primary/unique key, then the referencing table name and column names (repeated for as many referencing tables as were specified by first argument). Note that the primary/unique key columns should be marked NOT NULL and should have a unique index.

There are examples in `refint.example`.

**F.27.2. timetravel.c — functions for implementing time travel**

Long ago, PostgreSQL had a built-in time travel feature that kept the insert and delete times for each tuple. This can be emulated using these functions. To use these functions, you must add to a table two columns of *abstime* type to store the date when a tuple was inserted (start_date) and changed/deleted (stop_date):

```
CREATE TABLE mytab (  ...
    start_date  abstime,
    stop_date   abstime
    ...          ...
 ) ;
```

The columns can be named whatever you like, but in this discussion we’ll call them *start_date* and *stop_date*.

When a new row is inserted, *start_date* should normally be set to current time, and *stop_date* to *infinity*. The trigger will automatically substitute these values if the inserted data contains nulls in these columns. Generally, inserting explicit non-null data in these columns should only be done when re-loading dumped data.

Tuples with *stop_date* equal to *infinity* are “valid now”, and can be modified. Tuples with a finite *stop_date* cannot be modified anymore — the trigger will prevent it. (If you need to do that, you can turn off time travel as shown below.)
For a modifiable row, on update only the stop_date in the tuple being updated will be changed (to current time) and a new tuple with the modified data will be inserted. Start_date in this new tuple will be set to current time and stop_date to infinity.

A delete does not actually remove the tuple but only sets its stop_date to current time.

To query for tuples “valid now”, include stop_date = ‘infinity’ in the query’s WHERE condition. (You might wish to incorporate that in a view.) Similarily, you can query for tuples valid at any past time with suitable conditions on start_date and stop_date.

timetravel() is the general trigger function that supports this behavior. Create a BEFORE INSERT OR UPDATE OR DELETE trigger using this function on each time-traveled table. Specify two trigger arguments: the actual names of the start_date and stop_date columns. Optionally, you can specify one to three more arguments, which must refer to columns of type text. The trigger will store the name of the current user into the first of these columns during INSERT, the second column during UPDATE, and the third during DELETE.

set_timetravel() allows you to turn time-travel on or off for a table. set_timetravel(‘mytab’, 1) will turn TT ON for table mytab. set_timetravel(‘mytab’, 0) will turn TT OFF for table mytab. In both cases the old status is reported. While TT is off, you can modify the start_date and stop_date columns freely. Note that the on/off status is local to the current database session — fresh sessions will always start out with TT ON for all tables.

get_timetravel() returns the TT state for a table without changing it.

There is an example in timetravel.example.

**F.27.3. autoinc.c — functions for autoincrementing fields**

autoinc() is a trigger that stores the next value of a sequence into an integer field. This has some overlap with the built-in “serial column” feature, but it is not the same: autoinc() will override attempts to substitute a different field value during inserts, and optionally it can be used to increment the field during updates, too.

To use, create a BEFORE INSERT (or optionally BEFORE INSERT OR UPDATE) trigger using this function. Specify two trigger arguments: the name of the integer column to be modified, and the name of the sequence object that will supply values. (Actually, you can specify any number of pairs of such names, if you’d like to update more than one autoincrementing column.)

There is an example in autoinc.example.

**F.27.4. insert_username.c — functions for tracking who changed a table**

insert_username() is a trigger that stores the current user’s name into a text field. This can be useful for tracking who last modified a particular row within a table.

To use, create a BEFORE INSERT and/or UPDATE trigger using this function. Specify a single trigger argument: the name of the text column to be modified.

There is an example in insert_username.example.
F.27.5. moddatetime.c — functions for tracking last modification time

moddatetime() is a trigger that stores the current time into a timestamp field. This can be useful for tracking the last modification time of a particular row within a table.

To use, create a BEFORE UPDATE trigger using this function. Specify a single trigger argument: the name of the timestamp column to be modified.

There is an example in moddatetime.example.

F.28. sslinfo

The sslinfo module provides information about the SSL certificate that the current client provided when connecting to PostgreSQL. The module is useless (most functions will return NULL) if the current connection does not use SSL.

This extension won’t build at all unless the installation was configured with --with-openssl.

F.28.1. Functions Provided

ssl_is_used() returns boolean

Returns TRUE if current connection to server uses SSL, and FALSE otherwise.

ssl_client_cert_present() returns boolean

Returns TRUE if current client has presented a valid SSL client certificate to the server, and FALSE otherwise. (The server might or might not be configured to require a client certificate.)

ssl_client_serial() returns numeric

Returns serial number of current client certificate. The combination of certificate serial number and certificate issuer is guaranteed to uniquely identify a certificate (but not its owner — the owner ought to regularly change his keys, and get new certificates from the issuer).

So, if you run your own CA and allow only certificates from this CA to be accepted by the server, the serial number is the most reliable (albeit not very mnemonic) means to identify a user.

ssl_client_dn() returns text

Returns the full subject of the current client certificate, converting character data into the current database encoding. It is assumed that if you use non-ASCII characters in the certificate names, your database is able to represent these characters, too. If your database uses the SQL_ASCII encoding, non-ASCII characters in the name will be represented as UTF-8 sequences.

The result looks like /CN=Somebody /C=Some country/O=Some organization.

ssl_issuer_dn() returns text

Returns the full issuer name of the current client certificate, converting character data into the current database encoding. Encoding conversions are handled the same as for ssl_client_dn.

The combination of the return value of this function with the certificate serial number uniquely identifies the certificate.
This function is really useful only if you have more than one trusted CA certificate in your server’s root.crt file, or if this CA has issued some intermediate certificate authority certificates.

ssl_client_dn_field(fieldname text) returns text

This function returns the value of the specified field in the certificate subject, or NULL if the field is not present. Field names are string constants that are converted into ASN1 object identifiers using the OpenSSL object database. The following values are acceptable:

- commonName (alias CN)
- surname (alias SN)
- givenName (alias GN)
- countryName (alias C)
- localityName (alias L)
- stateOrProvinceName (alias ST)
- organizationName (alias O)
- organizationUnitName (alias OU)
- title
- description
- initials
- postalCode
- streetAddress
- generationQualifier
- description
- dnQualifier
- x500UniqueIdentifier
- pseudonym
- role
- emailAddress

All of these fields are optional, except commonName. It depends entirely on your CA’s policy which of them would be included and which wouldn’t. The meaning of these fields, however, is strictly defined by the X.500 and X.509 standards, so you cannot just assign arbitrary meaning to them.

ssl_issuer_field(fieldname text) returns text

Same as ssl_client_dn_field, but for the certificate issuer rather than the certificate subject.

F.28.2. Author

Victor Wagner <vitus@cryptocom.ru>, Cryptocom LTD

E-Mail of Cryptocom OpenSSL development group: <openssl@cryptocom.ru>

F.29. tablefunc

The tablefunc module includes various functions that return tables (that is, multiple rows). These functions are useful both in their own right and as examples of how to write C functions that return multiple rows.
F.29.1. Functions Provided

Table F-33. `tablefunc` functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>normal_rand(int numvals, float8 mean, float8 stddev)</code></td>
<td><code>setof float8</code></td>
<td>Produces a set of normally distributed random values</td>
</tr>
<tr>
<td><code>crosstab(text sql)</code></td>
<td><code>setof record</code></td>
<td>Produces a “pivot table” containing row names plus ( N ) value columns, where ( N ) is determined by the rowtype specified in the calling query.</td>
</tr>
<tr>
<td><code>crosstabN(text sql)</code></td>
<td><code>setof table_crosstab_N</code></td>
<td>Produces a “pivot table” containing row names plus ( N ) value columns. ( \text{crosstab2, crosstab3, and crosstab4 are predefined, but you can create additional crosstabN functions as described below} )</td>
</tr>
<tr>
<td><code>crosstab(text source_sql, text category_sql)</code></td>
<td><code>setof record</code></td>
<td>Produces a “pivot table” with the value columns specified by a second query.</td>
</tr>
<tr>
<td><code>crosstab(text sql, int N)</code></td>
<td><code>setof record</code></td>
<td>Obsolete version of <code>crosstab(text)</code>. The parameter ( N ) is now ignored, since the number of value columns is always determined by the calling query.</td>
</tr>
<tr>
<td><code>connectby(text relname, text keyid_fld, text parent_keyid_fld [, text orderby_fld ], text start_with, int max_depth [, text branch_delim ])</code></td>
<td><code>setof record</code></td>
<td>Produces a representation of a hierarchical tree structure.</td>
</tr>
</tbody>
</table>

F.29.1.1. `normal_rand`

`normal_rand(int numvals, float8 mean, float8 stddev)` returns `setof float8`

`normal_rand` produces a set of normally distributed random values (Gaussian distribution).

`numvals` is the number of values to be returned from the function. `mean` is the mean of the normal distribution of values and `stddev` is the standard deviation of the normal distribution of values.

For example, this call requests 1000 values with a mean of 5 and a standard deviation of 3:

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
```

```
  normal_rand
```

1902
Appendix F. Additional Supplied Modules

F.29.1.2. crosstab (text)

crosstab(text sql)
crosstab(text sql, int N)

The `crosstab` function is used to produce “pivot” displays, wherein data is listed across the page rather than down. For example, we might have data like

```
row1   val1
row1   val2
row1   val3
...
row2   val1
row2   val2
row2   val3
...
```

which we wish to display like

```
row1   val1   val2   val3   ...
row2   val1   val2   val3   ...
...
```

The `crosstab` function takes a text parameter that is a SQL query producing raw data formatted in the first way, and produces a table formatted in the second way.

The `sql` parameter is a SQL statement that produces the source set of data. This statement must return one `row_name` column, one `category` column, and one `value` column. N is an obsolete parameter, ignored if supplied (formerly this had to match the number of output value columns, but now that is determined by the calling query).

For example, the provided query might produce a set something like:

```
row_name   cat    value
----------+-------+-------
row1       cat1   val1
row1       cat2   val2
row1       cat3   val3
```
The `crosstab` function is declared to return `setof record`, so the actual names and types of the output columns must be defined in the `FROM` clause of the calling `SELECT` statement, for example:

```sql
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2 text);
```

This example produces a set something like:

```
<== value columns ==>
row_name | category_1 | category_2
---------+------------+------------
row1     | val1       | val2       
row2     | val5       | val6       
```

The `FROM` clause must define the output as one `row_name` column (of the same datatype as the first result column of the SQL query) followed by N `value` columns (all of the same datatype as the third result column of the SQL query). You can set up as many output value columns as you wish. The names of the output columns are up to you.

The `crosstab` function produces one output row for each consecutive group of input rows with the same `row_name` value. It fills the output value columns, left to right, with the `value` fields from these rows. If there are fewer rows in a group than there are output value columns, the extra output columns are filled with nulls; if there are more rows, the extra input rows are skipped.

In practice the SQL query should always specify `ORDER BY 1,2` to ensure that the input rows are properly ordered, that is, values with the same `row_name` are brought together and correctly ordered within the row. Notice that `crosstab` itself does not pay any attention to the second column of the query result; it’s just there to be ordered by, to control the order in which the third-column values appear across the page.

Here is a complete example:

```sql
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
  from ct
  where attribute = "att2" or attribute = "att3"
  order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);
```

```
row_name | category_1 | category_2 | category_3
---------+------------+------------+------------
row1     | val1       | val2       |          
row2     | val5       | val6       |          
```
Appendix F. Additional Supplied Modules

You can avoid always having to write out a `FROM` clause to define the output columns, by setting up a custom crosstab function that has the desired output row type wired into its definition. This is described in the next section. Another possibility is to embed the required `FROM` clause in a view definition.

**F.29.1.3. crosstabN (text)**

crosstabN(text sql)

The `crosstabN` functions are examples of how to set up custom wrappers for the general `crosstab` function, so that you need not write out column names and types in the calling `SELECT` query. The `tablefunc` module includes `crosstab2`, `crosstab3`, and `crosstab4`, whose output row types are defined as

```sql
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
    .
    .
    ,
    category_N TEXT
);
```

Thus, these functions can be used directly when the input query produces `row_name` and `value` columns of type `text`, and you want 2, 3, or 4 output values columns. In all other ways they behave exactly as described above for the general `crosstab` function.

For instance, the example given in the previous section would also work as

```sql
SELECT *
FROM crosstab3(
    'select rowid, attribute, value
     from ct
     where attribute = "att2" or attribute = "att3"
     order by 1,2');
```

These functions are provided mostly for illustration purposes. You can create your own return types and functions based on the underlying `crosstab()` function. There are two ways to do it:

- Create a composite type describing the desired output columns, similar to the examples in the installation script. Then define a unique function name accepting one `text` parameter and returning `setof your_type_name`, but linking to the same underlying `crosstabC` function. For example, if your source data produces row names that are `text`, and values that are `float8`, and you want 5 value columns:

  ```sql
  CREATE TYPE my_crosstab_float8_5_cols AS (}
F.29.1.4. crosstab(text, text)

crosstab(text source_sql, text category_sql)

The main limitation of the single-parameter form of crosstab is that it treats all values in a group alike, inserting each value into the first available column. If you want the value columns to correspond to specific categories of data, and some groups might not have data for some of the categories, that doesn’t work well. The two-parameter form of crosstab handles this case by providing an explicit list of the categories corresponding to the output columns.

source_sql is a SQL statement that produces the source set of data. This statement must return one row_name column, one category column, and one value column. It may also have one or more “extra” columns. The row_name column must be first. The category and value columns must be the last two columns, in that order. Any columns between row_name and category are treated as “extra”. The “extra” columns are expected to be the same for all rows with the same row_name value.

For example, source_sql might produce a set something like:

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;
```

<table>
<thead>
<tr>
<th>row_name</th>
<th>extra_col</th>
<th>cat</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>row1</td>
<td>extra1</td>
<td>cat1</td>
<td>val1</td>
</tr>
<tr>
<td>row1</td>
<td>extra1</td>
<td>cat2</td>
<td>val2</td>
</tr>
<tr>
<td>row1</td>
<td>extra1</td>
<td>cat4</td>
<td>val4</td>
</tr>
<tr>
<td>row2</td>
<td>extra2</td>
<td>cat1</td>
<td>val5</td>
</tr>
<tr>
<td>row2</td>
<td>extra2</td>
<td>cat2</td>
<td>val6</td>
</tr>
<tr>
<td>row2</td>
<td>extra2</td>
<td>cat3</td>
<td>val7</td>
</tr>
<tr>
<td>row2</td>
<td>extra2</td>
<td>cat4</td>
<td>val8</td>
</tr>
</tbody>
</table>
category_sql is a SQL statement that produces the set of categories. This statement must return only one column. It must produce at least one row, or an error will be generated. Also, it must not produce duplicate values, or an error will be generated. category_sql might be something like:

```sql
SELECT DISTINCT cat FROM foo ORDER BY 1;
```

```
<table>
<thead>
<tr>
<th>cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat1</td>
</tr>
<tr>
<td>cat2</td>
</tr>
<tr>
<td>cat3</td>
</tr>
<tr>
<td>cat4</td>
</tr>
</tbody>
</table>
```

The crosstab function is declared to return setof record, so the actual names and types of the output columns must be defined in the FROM clause of the calling SELECT statement, for example:

```sql
SELECT * FROM crosstab('...', '...')
AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

This will produce a result something like:

```
<== value columns ==>  
<table>
<thead>
<tr>
<th>row_name</th>
<th>extra</th>
<th>cat1</th>
<th>cat2</th>
<th>cat3</th>
<th>cat4</th>
</tr>
</thead>
<tbody>
<tr>
<td>row1</td>
<td>extra1</td>
<td>val1</td>
<td>val2</td>
<td>val3</td>
<td>val4</td>
</tr>
<tr>
<td>row2</td>
<td>extra2</td>
<td>val5</td>
<td>val6</td>
<td>val7</td>
<td>val8</td>
</tr>
</tbody>
</table>
```

The FROM clause must define the proper number of output columns of the proper data types. If there are N columns in the source_sql query’s result, the first N-2 of them must match up with the first N-2 output columns. The remaining output columns must have the type of the last column of the source_sql query’s result, and there must be exactly as many of them as there are rows in the category_sql query’s result.

The crosstab function produces one output row for each consecutive group of input rows with the same row_name value. The output row_name column, plus any “extra” columns, are copied from the first row of the group. The output value columns are filled with the value fields from rows having matching category values. If a row’s category does not match any output of the category_sql query, its value is ignored. Output columns whose matching category is not present in any input row of the group are filled with nulls.

In practice the source_sql query should always specify ORDER BY 1 to ensure that values with the same row_name are brought together. However, ordering of the categories within a group is not important. Also, it is essential to be sure that the order of the category_sql query’s output matches the specified output column order.

Here are two complete examples:

```sql
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);
```
```sql
select * from crosstab(
    'select year, month, qty from sales order by 1',
    'select m from generate_series(1,12) m'
) as (
    year int,
    "Jan" int,
    "Feb" int,
    "Mar" int,
    "Apr" int,
    "May" int,
    "Jun" int,
    "Jul" int,
    "Aug" int,
    "Sep" int,
    "Oct" int,
    "Nov" int,
    "Dec" int
);  
```

data:

<table>
<thead>
<tr>
<th>year</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007</td>
<td>1000</td>
<td>1500</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>500</td>
<td></td>
<td></td>
<td></td>
<td>1500</td>
<td>2000</td>
</tr>
<tr>
<td>2008</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

```sql
CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(  
    'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
    'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)  
AS  
(
    rowid text,
    rowdt timestamp,
    temperature int4,
    test_result text,
    test_startdate timestamp,
    volts float8
);
```

data:

<table>
<thead>
<tr>
<th>rowid</th>
<th>rowdt</th>
<th>temperature</th>
<th>test_result</th>
<th>test_startdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1</td>
<td>Sat Mar 01 00:00:00 2003</td>
<td>42</td>
<td>PASS</td>
<td></td>
</tr>
<tr>
<td>test2</td>
<td>Sun Mar 02 00:00:00 2003</td>
<td>53</td>
<td>FAIL</td>
<td>Sat Mar 01 00:00:00 2003</td>
</tr>
</tbody>
</table>

---

1908
You can create predefined functions to avoid having to write out the result column names and types in each query. See the examples in the previous section. The underlying C function for this form of crosstab is named crosstab_hash.

F.29.1.5. connectby

`connectby(text relname, text keyid_fld, text parent_keyid_fld
 [, text orderby_fld ], text start_with, int max_depth
 [, text branch_delim ])`

The `connectby` function produces a display of hierarchical data that is stored in a table. The table must have a key field that uniquely identifies rows, and a parent-key field that references the parent (if any) of each row. `connectby` can display the sub-tree descending from any row.

Table F-34. connectby parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relname</td>
<td>Name of the source relation</td>
</tr>
<tr>
<td>keyid_fld</td>
<td>Name of the key field</td>
</tr>
<tr>
<td>parent_keyid_fld</td>
<td>Name of the parent-key field</td>
</tr>
<tr>
<td>orderby_fld</td>
<td>Name of the field to order siblings by (optional)</td>
</tr>
<tr>
<td>start_with</td>
<td>Key value of the row to start at</td>
</tr>
<tr>
<td>max_depth</td>
<td>Maximum depth to descend to, or zero for unlimited depth</td>
</tr>
<tr>
<td>branch_delim</td>
<td>String to separate keys with in branch output (optional)</td>
</tr>
</tbody>
</table>

The key and parent-key fields can be any data type, but they must be the same type. Note that the `start_with` value must be entered as a text string, regardless of the type of the key field.

The `connectby` function is declared to return `setof record`, so the actual names and types of the output columns must be defined in the `FROM` clause of the calling `SELECT` statement, for example:

```sql
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, 
AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

The first two output columns are used for the current row’s key and its parent row’s key; they must match the type of the table’s key field. The third output column is the depth in the tree and must be of type `integer`. If a `branch_delim` parameter was given, the next output column is the branch display and must be of type `text`. Finally, if an `orderby_fld` parameter was given, the last output column is a serial number, and must be of type `integer`.

The “branch” output column shows the path of keys taken to reach the current row. The keys are separated by the specified `branch_delim` string. If no branch display is wanted, omit both the `branch_delim` parameter and the branch column in the output column list.

If the ordering of siblings of the same parent is important, include the `orderby_fld` parameter to specify which field to order siblings by. This field can be of any sortable data type. The output column list must include a final integer serial-number column, if and only if `orderby_fld` is specified.
connectby generates internally. Therefore, include double quotes if the names are mixed-case or contain special characters. You may also need to schema-qualify the table name.

In large tables, performance will be poor unless there is an index on the parent-key field.

It is important that the branch_delim string not appear in any key values, else connectby may incorrectly report an infinite-recursion error. Note that if branch_delim is not provided, a default value of `~` is used for recursion detection purposes.

Here is an example:

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);

-- with branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
AS t(keyid text, parent_keyid text, level int, branch text);
keyid | parent_keyid | level | branch
-------+--------------+-------+---------------------
 row2 | | 0 | row2
 row4 | row2 | 1 | row2~row4
 row6 | row4 | 2 | row2~row4~row6
 row8 | row6 | 3 | row2~row4~row6~row8
 row5 | row2 | 1 | row2~row5
 row9 | row5 | 2 | row2~row5~row9
(6 rows)

-- without branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
AS t(keyid text, parent_keyid text, level int);
keyid | parent_keyid | level
-------+--------------+-------
 row2 | | 0
 row4 | row2 | 1
 row6 | row4 | 2
 row8 | row6 | 3
 row5 | row2 | 1
 row9 | row5 | 2
(6 rows)

-- with branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
AS t(keyid text, parent_keyid text, level int, branch text, pos int);
keyid | parent_keyid | level | branch | pos
-------+--------------+-------+---------------------+-----
 row2 | | 0 | row2 | 1
 row5 | row2 | 1 | row2~row5 | 2
 row9 | row5 | 2 | row2~row5~row9 | 3
 row4 | row2 | 1 | row2~row4 | 4
(6 rows)
```
Appendix F. Additional Supplied Modules

F.29.2. Author
Joe Conway

F.30. test_parser

test_parser is an example of a custom parser for full-text search. It doesn’t do anything especially useful, but can serve as a starting point for developing your own parser.

test_parser recognizes words separated by white space, and returns just two token types:

mydb=# SELECT * FROM ts_token_type('testparser');

tokid | alias | description
-------+-------+---------------
  3 | word | Word
 12 | blank | Space symbols
(2 rows)

These token numbers have been chosen to be compatible with the default parser’s numbering. This allows us to use its headline() function, thus keeping the example simple.

F.30.1. Usage

Running the installation script creates a text search parser testparser. It has no user-configurable parameters.

You can test the parser with, for example,

mydb=# SELECT * FROM ts_parse('testparser', 'That’s my first own parser');

tokid | token
-------+--------
  3 | That’s
 12 |
Real-world use requires setting up a text search configuration that uses the parser. For example,

```sql
mydb=# CREATE TEXT SEARCH CONFIGURATION testcfg ( PARSER = testparser );
CREATE TEXT SEARCH CONFIGURATION

mydb=# ALTER TEXT SEARCH CONFIGURATION testcfg
mydb-# ADD MAPPING FOR word WITH english_stem;
ALTER TEXT SEARCH CONFIGURATION

mydb=# SELECT to_tsvector('testcfg', 'That’s my first own parser');
to_tsvector
------------------------------
'that':1 'first':3 'parser':5
(1 row)

mydb=# SELECT ts_headline('testcfg', 'Supernovae stars are the brightest phenomena in galaxies',
mydb(# to_tsquery('testcfg', 'star'));
ts_headline
-----------------------------------------------------------------
Supernovae <b>stars</b> are the brightest phenomena in galaxies
(1 row)
```

**F.31. tsearch2**

The tsearch2 module provides backwards-compatible text search functionality for applications that used contrib/tsearch2 before text searching was integrated into core PostgreSQL in release 8.3.

**F.31.1. Portability Issues**

Although the built-in text search features were based on contrib/tsearch2 and are largely similar to it, there are numerous small differences that will create portability issues for existing applications:

- Some functions’ names were changed, for example rank to ts_rank. The replacement tsearch2 module provides aliases having the old names.
- The built-in text search data types and functions all exist within the system schema pg_catalog. In an installation using contrib/tsearch2, these objects would usually have been in the public schema, though some users chose to place them in a separate schema of their own. Explicitly schema-qualified references to the objects will therefore fail in either case. The replacement tsearch2 module provides alias objects that are stored in public (or another schema if necessary) so that such references will still work.
Appendix F. Additional Supplied Modules

- There is no concept of a “current parser” or “current dictionary” in the built-in text search features, only of a current search configuration (set by the `default_text_search_config` parameter). While the current parser and current dictionary were used only by functions intended for debugging, this might still pose a porting obstacle in some cases. The replacement `tsearch2` module emulates these additional state variables and provides backwards-compatible functions for setting and retrieving them.

There are some issues that are not addressed by the replacement `tsearch2` module, and will therefore require application code changes in any case:

- The old `tsearch2` trigger function allowed items in its argument list to be names of functions to be invoked on the text data before it was converted to `tsvector` format. This was removed as being a security hole, since it was not possible to guarantee that the function invoked was the one intended. The recommended approach if the data must be massaged before being indexed is to write a custom trigger that does the work for itself.

- Text search configuration information has been moved into core system catalogs that are noticeably different from the tables used by `contrib/tsearch2`. Any applications that examined or modified those tables will need adjustment.

- If an application used any custom text search configurations, those will need to be set up in the core catalogs using the new text search configuration SQL commands. The replacement `tsearch2` module offers a little bit of support for this by making it possible to load an old set of `contrib/tsearch2` configuration tables into PostgreSQL 8.3. (Without the module, it is not possible to load the configuration data because values in the `regprocedure` columns cannot be resolved to functions.) While those configuration tables won’t actually do anything, at least their contents will be available to be consulted while setting up an equivalent custom configuration in 8.3.

- The old `reset_tsearch()` and `get_covers()` functions are not supported.

- The replacement `tsearch2` module does not define any alias operators, relying entirely on the built-in ones. This would only pose an issue if an application used explicitly schema-qualified operator names, which is very uncommon.

F.31.2. Converting a pre-8.3 Installation

The recommended way to update a pre-8.3 installation that uses `contrib/tsearch2` is:

1. Make a dump from the old installation in the usual way, but be sure not to use `-c (--clean)` option of `pg_dump` or `pg_dumpall`.
2. In the new installation, create empty database(s) and install the replacement `tsearch2` module into each database that will use text search. This must be done before loading the dump data! If your old installation had the `contrib/tsearch2` objects in a schema other than `public`, be sure to adjust the `tsearch2` installation script so that the replacement objects are created in that same schema.
3. Load the dump data. There will be quite a few errors reported due to failure to recreate the original `contrib/tsearch2` objects. These errors can be ignored, but this means you cannot restore the dump in a single transaction (e.g., you cannot use `pg_restore`’s `-i` switch).
4. Examine the contents of the restored `contrib/tsearch2` configuration tables (e.g., `pg_ts_cfg` and so on), and create equivalent built-in text search configurations as needed. You may drop the old configuration tables once you’ve extracted all the useful information from them.
5. Test your application.

At a later time you may wish to rename application references to the alias text search objects, so that you can eventually uninstall the replacement tsearch2 module.

### F.31.3. References

Tsearch2 Development Site http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

## F.32. uuid-ossp

The **uuid-ossp** module provides functions to generate universally unique identifiers (UUIDs) using one of several standard algorithms. There are also functions to produce certain special UUID constants.

This module depends on the OSSP UUID library, which can be found at http://www.ossp.org/pkg/lib/uuid/.

### F.32.1. uuid-ossp Functions

The relevant standards ITU-T Rec. X.667, ISO/IEC 9834-8:2005, and RFC 4122 specify four algorithms for generating UUIDs, identified by the version numbers 1, 3, 4, and 5. (There is no version 2 algorithm.) Each of these algorithms could be suitable for a different set of applications.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuid_generate_v1()</td>
<td>This function generates a version 1 UUID. This involves the MAC address of the computer and a time stamp. Note that UUIDs of this kind reveal the identity of the computer that created the identifier and the time at which it did so, which might make it unsuitable for certain security-sensitive applications.</td>
</tr>
<tr>
<td>uuid_generate_v1mc()</td>
<td>This function generates a version 1 UUID but uses a random multicast MAC address instead of the real MAC address of the computer.</td>
</tr>
<tr>
<td>uuid_generate_v3(namespace uuid, name text)</td>
<td>This function generates a version 3 UUID in the given namespace using the specified input name. The namespace should be one of the special constants produced by the <code>uuid_ns_*()</code> functions shown below. (It could be any UUID in theory.) The name is an identifier in the selected namespace.</td>
</tr>
<tr>
<td>uuid_generate_v4()</td>
<td>This function generates a version 4 UUID, which is derived entirely from random numbers.</td>
</tr>
</tbody>
</table>
Appendix F. Additional Supplied Modules

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuid_generate_v5(namespace uuid, name text)</td>
<td>This function generates a version 5 UUID, which works like a version 3 UUID except that SHA-1 is used as a hashing method. Version 5 should be preferred over version 3 because SHA-1 is thought to be more secure than MD5.</td>
</tr>
</tbody>
</table>

For example:

```sql
SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org');
```

The name parameter will be MD5-hashed, so the cleartext cannot be derived from the generated UUID. The generation of UUIDs by this method has no random or environment-dependent element and is therefore reproducible.

### Table F-36. Functions Returning UUID Constants

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuid_nil()</td>
<td>A “nil” UUID constant, which does not occur as a real UUID.</td>
</tr>
<tr>
<td>uuid_ns_dns()</td>
<td>Constant designating the DNS namespace for UUIDs.</td>
</tr>
<tr>
<td>uuid_ns_url()</td>
<td>Constant designating the URL namespace for UUIDs.</td>
</tr>
<tr>
<td>uuid_ns_oid()</td>
<td>Constant designating the ISO object identifier (OID) namespace for UUIDs.</td>
</tr>
<tr>
<td>uuid_ns_x500()</td>
<td>Constant designating the X.500 distinguished name (DN) namespace for UUIDs.</td>
</tr>
</tbody>
</table>

### F.32.2. Author

Peter Eisentraut <peter_e@gmx.net>

### F.33. vacuumlo

vacuumlo is a simple utility program that will remove any “orphaned” large objects from a PostgreSQL database. An orphaned large object (LO) is considered to be any LO whose OID does not appear in any oid or lo data column of the database.

If you use this, you may also be interested in the lo_manage trigger in contrib/lo (see Section F.14). lo_manage is useful to try to avoid creating orphaned LOs in the first place.

### F.33.1. Usage

```
vacuumlo [options] database [database2 ... databaseN]
```
All databases named on the command line are processed. Available options include:

- `v`
  Write a lot of progress messages.
- `n`
  Don’t remove anything, just show what would be done.
- `U username`
  Username to connect as.
- `W`
  Force vacuumlo to prompt for a password before connecting to a database.
  This option is never essential, since vacuumlo will automatically prompt for a password if the server demands password authentication. However, vacuumlo will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.
- `h hostname`
  Database server’s host.
- `p port`
  Database server’s port.

**F.33.2. Method**

First, it builds a temporary table which contains all of the OIDs of the large objects in that database. It then scans through all columns in the database that are of type `oid` or `lo`, and removes matching entries from the temporary table. The remaining entries in the temp table identify orphaned LOs. These are removed.

**F.33.3. Author**

Peter Mount <peter@retep.org.uk>

**F.34. xml2**

The `xml2` module provides XPath querying and XSLT functionality.

**F.34.1. Deprecation notice**

From PostgreSQL 8.3 on, there is XML-related functionality based on the SQL/XML standard in the core server. That functionality covers XML syntax checking and XPath queries, which is what this module does, and more, but the API is not at all compatible. It is planned that this module will be removed in PostgreSQL 8.4 in favor of the newer standard API, so you are encouraged to try
converting your applications. If you find that some of the functionality of this module is not available in an adequate form with the newer API, please explain your issue to pgsql-hackers@postgresql.org so that the deficiency can be addressed.

### F.34.2. Description of functions

These functions provide straightforward XML parsing and XPath queries. All arguments are of type `text`, so for brevity that is not shown.

#### Table F-37. Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xml_is_well_formed(document)</code> returns bool</td>
<td>This parses the document text in its parameter and returns true if the document is well-formed XML. (Note: before PostgreSQL 8.2, this function was called <code>xml_valid()</code>. That is the wrong name since validity and well-formedness have different meanings in XML. The old name is still available, but is deprecated.)</td>
</tr>
<tr>
<td><code>xpath_string(document, query)</code> returns text</td>
<td>These functions evaluate the XPath query on the supplied document, and cast the result to the specified type.</td>
</tr>
<tr>
<td><code>xpath_number(document, query)</code> returns float4</td>
<td></td>
</tr>
<tr>
<td><code>xpath_bool(document, query)</code> returns bool</td>
<td></td>
</tr>
</tbody>
</table>
| `xpath_nodeset(document, query, toptag, itemtag)` returns text | This evaluates query on document and wraps the result in XML tags. If the result is multivalued, the output will look like: `<toptag> <itemtag>Value 1 which could be an XML fragment</itemtag> <itemtag>Value 2....</itemtag> </toptag>
   If either toptag or itemtag is an empty string, the relevant tag is omitted. |
| `xpath_nodeset(document, query)` returns text  | Like `xpath_nodeset(document, query, toptag, itemtag)` but result omits both tags. |
| `xpath_nodeset(document, query, itemtag)`     | Like `xpath_nodeset(document, query, toptag, itemtag)` but result omits toptag.   |
| `xpath_list(document, query, separator)`     | This function returns multiple values separated by the specified separator, for example `Value 1,Value 2,Value 3` if separator is `,` |
| `xpath_list(document, query)` returns text    | This is a wrapper for the above function that uses `,` as the separator.          |
**F.34.3. xpath_table**

xpath_table is a table function that evaluates a set of XPath queries on each of a set of documents and returns the results as a table. The primary key field from the original document table is returned as the first column of the result so that the result set can readily be used in joins.

**Table F-38. Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>the name of the “key” field — this is just a field to be used as the first column of the output table, i.e. it identifies the record from which each output row came (see note below about multiple values)</td>
</tr>
<tr>
<td>document</td>
<td>the name of the field containing the XML document</td>
</tr>
<tr>
<td>relation</td>
<td>the name of the table or view containing the documents</td>
</tr>
<tr>
<td>xpaths</td>
<td>one or more XPath expressions, separated by</td>
</tr>
<tr>
<td>criteria</td>
<td>the contents of the WHERE clause. This cannot be omitted, so use true or 1=1 if you want to process all the rows in the relation</td>
</tr>
</tbody>
</table>

These parameters (except the XPath strings) are just substituted into a plain SQL SELECT statement, so you have some flexibility — the statement is

```sql
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

so those parameters can be *anything* valid in those particular locations. The result from this SELECT needs to return exactly two columns (which it will unless you try to list multiple fields for key or document). Beware that this simplistic approach requires that you validate any user-supplied values to avoid SQL injection attacks.

The function has to be used in a FROM expression, with an AS clause to specify the output columns; for example

```sql
SELECT * FROM
xpath_table(‘article_id’,
    ‘article_xml’,
    ‘articles’,
    ‘article/author|article/pages|article/title’,
    ‘date_entered > “2003-01-01” ‘)
AS t(article_id integer, author text, page_count integer, title text);
```

The AS clause defines the names and types of the columns in the output table. The first is the “key” field and the rest correspond to the XPath queries. If there are more XPath queries than result columns, the extra queries will be ignored. If there are more result columns than XPath queries, the extra columns will be NULL.

Notice that this example defines the page_count result column as an integer. The function deals internally with string representations, so when you say you want an integer in the output, it will take the string representation of the XPath result and use PostgreSQL input functions to transform it into
Appendix F. Additional Supplied Modules

An integer (or whatever type the AS clause requests). An error will result if it can’t do this — for example if the result is empty — so you may wish to just stick to text as the column type if you think your data has any problems.

The calling SELECT statement doesn’t necessarily have be be just SELECT * — it can reference the output columns by name or join them to other tables. The function produces a virtual table with which you can perform any operation you wish (e.g. aggregation, joining, sorting etc). So we could also have:

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
    '/article/title|/article/author/@id',
    'xpath_string(article_xml,"/article/@date") > "2003-03-20" ') AS t(article_id integer, title text, author_id integer),
    tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

as a more complicated example. Of course, you could wrap all of this in a view for convenience.

**F.34.3.1. Multivalued results**

The xpath_table function assumes that the results of each XPath query might be multi-valued, so the number of rows returned by the function may not be the same as the number of input documents. The first row returned contains the first result from each query, the second row the second result from each query. If one of the queries has fewer values than the others, NULLs will be returned instead.

In some cases, a user will know that a given XPath query will return only a single result (perhaps a unique document identifier) — if used alongside an XPath query returning multiple results, the single-valued result will appear only on the first row of the result. The solution to this is to use the key field as part of a join against a simpler XPath query. As an example:

```
CREATE TABLE test {
    id int4 NOT NULL,
    xml text,
    CONSTRAINT pk PRIMARY KEY (id)
};

INSERT INTO test VALUES (1, '<doc num="C1"><line num="L1"><a>1</a><b>2</b><c>3</c></line>
    <line num="L2"><a>1</a><b>2</b><c>3</c></line>
'></doc>');

INSERT INTO test VALUES (2, '<doc num="C2"><line num="L1"><a>11</a><b>22</b><c>33</c></line>
    <line num="L2"><a>11</a><b>22</b><c>33</c></line>
'></doc>');

SELECT * FROM
    xpath_table('id','xml','test',
        '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
        'true')
    AS t(id int4, doc_num varchar(10), line_num varchar(10), val1 int4, val2 int4, val3 int4)
WHERE id = 1 ORDER BY doc_num, line_num
```

<table>
<thead>
<tr>
<th>id</th>
<th>doc_num</th>
<th>line_num</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1919</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To get doc_num on every line, the solution is to use two invocations of xpath_table and join the results:

```
SELECT t.*, i.doc_num FROM
  xpath_table('id', 'xml', 'test',
    '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
    'true')
AS t(id int4, line_num varchar(10), val1 int4, val2 int4, val3 int4),
xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
AS i(id int4, doc_num varchar(10))
WHERE i.id = t.id AND i.id = 1
ORDER BY doc_num, line_num;
```

<table>
<thead>
<tr>
<th>id</th>
<th>line_num</th>
<th>val1</th>
<th>val2</th>
<th>val3</th>
<th>doc_num</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>C1</td>
</tr>
<tr>
<td>1</td>
<td>L2</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>C1</td>
</tr>
</tbody>
</table>

(2 rows)

**F.34.4. XSLT functions**

The following functions are available if libxslt is installed:

**F.34.4.1. xslt_process**

```
xslt_process(text document, text stylesheet, text paramlist) returns text
```

This function applies the XSL stylesheets to the document and returns the transformed result. The paramlist is a list of parameter assignments to be used in the transformation, specified in the form `a=1,b=2`. Note that the parameter parsing is very simple-minded: parameter values cannot contain commas!

There is also a two-parameter version of `xslt_process` which does not pass any parameters to the transformation.

**F.34.5. Author**

John Gray <jgray@azuli.co.uk>

Development of this module was sponsored by Torchbox Ltd. (www.torchbox.com). It has the same BSD licence as PostgreSQL.
Appendix G. External Projects

PostgreSQL is a complex software project, and managing the project is difficult. We have found that many enhancements to PostgreSQL can be more efficiently developed separately from the core project.

To help our community with the development of their external projects, we have created PgFoundry¹, a website that provides hosting for PostgreSQL-related projects that are maintained outside the core PostgreSQL distribution. PgFoundry is built using the GForge software project and is similar to SourceForge.net² in its feature set, providing mailing lists, forums, bug tracking, CVS, and web hosting. If you have a PostgreSQL-related open source project that you would like to have hosted at PgFoundry, please feel free to create a new project.

G.1. Client Interfaces

There are only two client interfaces included in the base PostgreSQL distribution:

- libpq is included because it is the primary C language interface, and because many other client interfaces are built on top of it.
- ecpg is included because it depends on the server-side SQL grammar, and is therefore sensitive to changes in PostgreSQL itself.

All other language interfaces are external projects and are distributed separately. Table G-1 includes a list of some of these projects. Note that some of these packages might not be released under the same license as PostgreSQL. For more information on each language interface, including licensing terms, refer to its website and documentation.

Table G-1. Externally Maintained Client Interfaces

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>Comments</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBD::Pg</td>
<td>Perl</td>
<td>Perl DBI driver</td>
<td><a href="http://search.cpan.org/dist/DBD-Pg/">http://search.cpan.org/dist/DBD-Pg/</a></td>
</tr>
<tr>
<td>JDBC</td>
<td>JDBC</td>
<td>Type 4 JDBC driver</td>
<td><a href="http://jdbc.postgresql.org/">http://jdbc.postgresql.org/</a></td>
</tr>
<tr>
<td>libpqxx</td>
<td>C++</td>
<td>New-style C++ interface</td>
<td><a href="http://pqxx.org/">http://pqxx.org/</a></td>
</tr>
<tr>
<td>Npgsql</td>
<td>.NET</td>
<td>.NET data provider</td>
<td><a href="http://npgsql.projects.postgresql.org/">http://npgsql.projects.postgresql.org/</a></td>
</tr>
<tr>
<td>pgtclng</td>
<td>Tcl</td>
<td></td>
<td><a href="http://sourceforge.net/projects/pgtclng/">http://sourceforge.net/projects/pgtclng/</a></td>
</tr>
<tr>
<td>psqlODBC</td>
<td>ODBC</td>
<td>ODBC driver</td>
<td><a href="http://psqlodbc.projects.postgresql.org/">http://psqlodbc.projects.postgresql.org/</a></td>
</tr>
</tbody>
</table>

¹. http://www.pgfoundry.org/
Appendix G. External Projects

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>Comments</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>psycopg</td>
<td>Python</td>
<td>DB API 2.0-compliant</td>
<td><a href="http://www.initd.org/">http://www.initd.org/</a></td>
</tr>
</tbody>
</table>

G.2. Procedural Languages

PostgreSQL includes several procedural languages with the base distribution: PL/PgSQL, PL/Tcl, PL/Perl, and PL/Python.

In addition, there are a number of procedural languages that are developed and maintained outside the core PostgreSQL distribution. Table G-2 lists some of these packages. Note that some of these projects might not be released under the same license as PostgreSQL. For more information on each procedural language, including licensing information, refer to its website and documentation.

Table G-2. Externally Maintained Procedural Languages

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/Java</td>
<td>Java</td>
<td><a href="http://pljava.projects.postgresql.org/">http://pljava.projects.postgresql.org/</a></td>
</tr>
<tr>
<td>PL/PHP</td>
<td>PHP</td>
<td><a href="http://www.commandprompt.com/community/plphp/">http://www.commandprompt.com/community/plphp/</a></td>
</tr>
<tr>
<td>PL/Py</td>
<td>Python</td>
<td><a href="http://python.projects.postgresql.org/">http://python.projects.postgresql.org/</a></td>
</tr>
<tr>
<td>PL/R</td>
<td>R</td>
<td><a href="http://www.joeconway.com/plr/">http://www.joeconway.com/plr/</a></td>
</tr>
<tr>
<td>PL/Ruby</td>
<td>Ruby</td>
<td><a href="http://raa.ruby-lang.org/project/pl-ruby/">http://raa.ruby-lang.org/project/pl-ruby/</a></td>
</tr>
<tr>
<td>PL/Scheme</td>
<td>Scheme</td>
<td><a href="http://plscheme.projects.postgresql.org/">http://plscheme.projects.postgresql.org/</a></td>
</tr>
<tr>
<td>PL/sh</td>
<td>Unix shell</td>
<td><a href="http://plsh.projects.postgresql.org/">http://plsh.projects.postgresql.org/</a></td>
</tr>
</tbody>
</table>

G.3. Extensions

PostgreSQL is designed to be easily extensible. For this reason, extensions loaded into the database can function just like features that are packaged with the database. The contrib/ directory shipped with the source code contains a large number of extensions. The README file in that directory contains a summary. They include conversion tools, full-text indexing, XML tools, and additional data types and indexing methods. Other extensions are developed independently, like PostGIS\(^3\). Even PostgreSQL replication solutions are developed externally. For example, Slony-I\(^4\) is a popular master/slave replication solution that is developed independently from the core project.

There are several administration tools available for PostgreSQL. The most popular is pgAdmin III\(^5\), and there are several commercially available ones as well.

---

Appendix H. The Source Code Repository

The PostgreSQL source code is stored and managed using the Git version control system. A public mirror of the master repository is available; it is updated within a minute of any change to the master repository.


Note that building PostgreSQL from the source repository requires reasonably up-to-date versions of bison, flex, and Perl. These tools are not needed to build from a distribution tarball since the files they are used to build are included in the tarball. Other tool requirements are the same as shown in Chapter 15.

H.1. Getting The Source Via Git

With Git you will make a copy of the entire code repository on your local machine, so you will have access to all history and branches offline. This is the fastest and most flexible way to develop or test patches.

Git

1. You will need an installed version of Git, which you can get from http://git-scm.com. Many systems already have a recent version of Git installed by default, or available in their package distribution system.

2. To begin using the Git repository, make a clone of the official mirror:

   git clone git://git.postgresql.org/git/postgresql.git

   This will copy the full repository to your local machine, so it may take a while to complete, especially if you have a slow Internet connection. The files will be placed in a new subdirectory postgresql of your current directory.

   The Git mirror can also be reached via the HTTP protocol, if for example a firewall is blocking access to the Git protocol. Just change the URL prefix to http, as in:

   git clone http://git.postgresql.org/git/postgresql.git

   The HTTP protocol is less efficient than the Git protocol, so it will be slower to use.

3. Whenever you want to get the latest updates in the system, cd into the repository, and run:

   git fetch

   Git can do a lot more things than just fetch the source. For more information, consult the Git man pages, or see the website at http://git-scm.com.
Appendix I. Documentation

PostgreSQL has four primary documentation formats:

- Plain text, for pre-installation information
- HTML, for on-line browsing and reference
- PDF or PostScript, for printing
- man pages, for quick reference.

Additionally, a number of plain-text README files can be found throughout the PostgreSQL source tree, documenting various implementation issues.

HTML documentation and man pages are part of a standard distribution and are installed by default. PDF and PostScript format documentation is available separately for download.

I.1. DocBook

The documentation sources are written in DocBook, which is a markup language superficially similar to HTML. Both of these languages are applications of the Standard Generalized Markup Language, SGML, which is essentially a language for describing other languages. In what follows, the terms DocBook and SGML are both used, but technically they are not interchangeable.

DocBook allows an author to specify the structure and content of a technical document without worrying about presentation details. A document style defines how that content is rendered into one of several final forms. DocBook is maintained by the OASIS group. The official DocBook site has good introductory and reference documentation and a complete O’Reilly book for your online reading pleasure. The NewbieDoc Docbook Guide is very helpful for beginners. The FreeBSD Documentation Project also uses DocBook and has some good information, including a number of style guidelines that might be worth considering.

I.2. Tool Sets

The following tools are used to process the documentation. Some might be optional, as noted.

DocBook DTD

This is the definition of DocBook itself. We currently use version 4.2; you cannot use later or earlier versions. Note that there is also an XML version of DocBook — do not use that.

Appendix I. Documentation

ISO 8879 character entities\(^6\)

These are required by DocBook but are distributed separately because they are maintained by ISO.

OpenJade\(^7\)

This is the base package of SGML processing. It contains an SGML parser, a DSSSL processor (that is, a program to convert SGML to other formats using DSSSL stylesheets), as well as a number of related tools. Jade is now being maintained by the OpenJade group, no longer by James Clark.

DocBook DSSSL Stylesheets\(^8\)

These contain the processing instructions for converting the DocBook sources to other formats, such as HTML.

DocBook2X tools\(^9\)

This optional package is used to create man pages. It has a number of prerequisite packages of its own. Check the web site.

JadeTeX\(^10\)

If you want to, you can also install JadeTeX to use TeX as a formatting backend for Jade. JadeTeX can create PostScript or PDF files (the latter with bookmarks).

However, the output from JadeTeX is inferior to what you get from the RTF backend. Particular problem areas are tables and various artifacts of vertical and horizontal spacing. Also, there is no opportunity to manually polish the results.

We have documented experience with several installation methods for the various tools that are needed to process the documentation. These will be described below. There might be some other packaged distributions for these tools. Please report package status to the documentation mailing list, and we will include that information here.

I.2.1. Linux RPM Installation

Most vendors provide a complete RPM set for DocBook processing in their distribution. Look for an “SGML” option while installing, or the following packages: sgml-common, docbook, stylesheets, openjade (or jade). Possibly sgml-tools will be needed as well. If your distributor does not provide these then you should be able to make use of the packages from some other, reasonably compatible vendor.

I.2.2. FreeBSD Installation

The FreeBSD Documentation Project is itself a heavy user of DocBook, so it comes as no surprise that there is a full set of “ports” of the documentation tools available on FreeBSD. The following ports

---

need to be installed to build the documentation on FreeBSD.

- textproc/sp
- textproc/openjade
- textproc/iso8879
- textproc/dsssl-docbook-modular
- textproc/docbook-420

A number of things from /usr/ports/print (tex, jadetex) might also be of interest.
It’s possible that the ports do not update the main catalog file in /usr/local/share/sgml/catalog.ports or order isn’t proper. Be sure to have the following lines in beginning of file:

```
CATALOG "openjade/catalog"
CATALOG "iso8879/catalog"
CATALOG "docbook/dsssl/modular/catalog"
CATALOG "docbook/4.2/catalog"
```

If you do not want to edit the file you can also set the environment variable SGML_CATALOG_FILES to a colon-separated list of catalog files (such as the one above).

More information about the FreeBSD documentation tools can be found in the FreeBSD Documentation Project’s instructions\(^\text{11}\).

### I.2.3. Debian Packages

There is a full set of packages of the documentation tools available for Debian GNU/Linux. To install, simply use:

```
apt-get install openjade1.3
apt-get install docbook
apt-get install docbook-dsssl
```

(The plain openjade package installs OpenJade 1.4, which seems not to work.)

### I.2.4. Manual Installation from Source

The manual installation process of the DocBook tools is somewhat complex, so if you have pre-built packages available, use them. We describe here only a standard setup, with reasonably standard installation paths, and no “fancy” features. For details, you should study the documentation of the respective package, and read SGML introductory material.

I.2.4.1. Installing OpenJade

1. The installation of OpenJade offers a GNU-style `./configure; make; make install` build process. Details can be found in the OpenJade source distribution. In a nutshell:

```
./configure --enable-default-catalog=/usr/local/share/sgml/catalog
make
make install
```

Be sure to remember where you put the “default catalog”; you will need it below. You can also leave it off, but then you will have to set the environment variable `SGML_CATALOG_FILES` to point to the file whenever you use jade later on. (This method is also an option if OpenJade is already installed and you want to install the rest of the tool chain locally.)

2. Additionally, you should install the files `dsssl.dtd`, `fot.dtd`, `style-sheet.dtd`, and `catalog` from the `dsssl` directory somewhere, perhaps into `/usr/local/share/sgml/dsssl`. It’s probably easiest to copy the entire directory:

```
cp -R dsssl /usr/local/share/sgml
```

3. Finally, create the file `/usr/local/share/sgml/catalog` and add this line to it:

```
CATALOG "dsssl/catalog"
```

(This is a relative path reference to the file installed in step 2. Be sure to adjust it if you chose your installation layout differently.)

I.2.4.2. Installing the DocBook DTD Kit

1. Obtain the DocBook V4.2 distribution\(^{12}\).

2. Create the directory `/usr/local/share/sgml/docbook-4.2` and change to it. (The exact location is irrelevant, but this one is reasonable within the layout we are following here.)

```
$ mkdir /usr/local/share/sgml/docbook-4.2
$ cd /usr/local/share/sgml/docbook-4.2
```

3. Unpack the archive.

```
$ unzip -a ...../docbook-4.2.zip
```

(The archive will unpack its files into the current directory.)

4. Edit the file `/usr/local/share/sgml/catalog` (or whatever you told jade during installation) and put a line like this into it:

```
CATALOG "docbook-4.2/docbook.cat"
```

5. Download the ISO 8879 character entities archive\(^{13}\), unpack it, and put the files in the same directory you put the DocBook files in.

```
$ cd /usr/local/share/sgml/docbook-4.2
$ unzip ...../ISOEnts.zip
```

6. Run the following command in the directory with the DocBook and ISO files:

```
perl -pi -e 's/iso-(.*)gml/ISO\1/g' docbook.cat
```

(This fixes a mixup between the names used in the DocBook catalog file and the actual names of the ISO character entity files.)

---

\(^{12}\) http://www.docbook.org/sgml/4.2/docbook-4.2.zip

\(^{13}\) http://www.oasis-open.org/cover/ISOEnts.zip
I.2.4.3. Installing the DocBook DSSSL Style Sheets

To install the style sheets, unzip and untar the distribution and move it to a suitable place, for example /usr/local/share/sgml. (The archive will automatically create a subdirectory.)

```bash
$ gunzip docbook-dsssl-1.xx.tar.gz
$ tar -C /usr/local/share/sgml -xf docbook-dsssl-1.xx.tar
```

The usual catalog entry in /usr/local/share/sgml/catalog can also be made:

```
CATALOG "docbook-dsssl-1.xx/catalog"
```

Because stylesheets change rather often, and it’s sometimes beneficial to try out alternative versions, PostgreSQL doesn’t use this catalog entry. See Section I.2.5 for information about how to select the stylesheets instead.

I.2.4.4. Installing JadeTeX

To install and use JadeTeX, you will need a working installation of TeX and LaTeX2e, including the supported tools and graphics packages, Babel, AMS fonts and AMS-LaTeX, the PSNFSS extension and companion kit of “the 35 fonts”, the dvips program for generating PostScript, the macro packages fancyhdr, hyperref, minitoc, url and o2enc. All of these can be found on your friendly neighborhood CTAN site[^14]. The installation of the TeX base system is far beyond the scope of this introduction. Binary packages should be available for any system that can run TeX.

Before you can use JadeTeX with the PostgreSQL documentation sources, you will need to increase the size of TeX’s internal data structures. Details on this can be found in the JadeTeX installation instructions.

Once that is finished you can install JadeTeX:

```bash
$ gunzip jadetex-xxx.tar.gz
$ tar xf jadetex-xxx.tar
$ cd jadetex
$ make install
$ mktexlsr
```

The last two need to be done as root.

I.2.5. Detection by configure

Before you can build the documentation you need to run the configure script as you would when building the PostgreSQL programs themselves. Check the output near the end of the run, it should look something like this:

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V4.2... yes
checking for DocBook stylesheets... /usr/lib/sgml/stylesheets/nwalsh-modular
checking for sgmlspl... sgmlspl
```

[^14]: http://www.ctan.org
Appendix I. Documentation

If neither onsgmls nor nsgmls were found then you will not see the remaining 4 lines. nsgmls is part of the Jade package. You can pass the environment variables JADE and NSGMLS to configure to point to the programs if they are not found automatically. If “DocBook V4.2” was not found then you did not install the DocBook DTD kit in a place where Jade can find it, or you have not set up the catalog files correctly. See the installation hints above. The DocBook stylesheets are looked for in a number of relatively standard places, but if you have them some other place then you should set the environment variable DOCBOOKSTYLE to the location and rerun configure afterwards.

I.3. Building The Documentation

Once you have everything set up, change to the directory doc/src/sgml and run one of the commands described in the following subsections to build the documentation. (Remember to use GNU make.)

I.3.1. HTML

To build the HTML version of the documentation:

```
doc/src/sgml$ gmake html
```

This is also the default target.

To create a proper index, the build might process several identical stages. If you do not care about the index, and just want to proof-read the output, use draft:

```
doc/src/sgml$ gmake draft
```

To allow for easier handling in the final distribution, the files comprising the HTML documentation can be stored in a tar archive that is unpacked at installation time. To create the HTML documentation package, use the commands:

```
cd doc/src
gmake postgres.tar.gz
```

In the distribution, these archives live in the doc directory and are installed by default with gmake install.

I.3.2. Manpages

We use the docbook2man utility to convert DocBook refentry pages to *roff output suitable for man pages. The man pages are also distributed as a tar archive, similar to the HTML version. To create the man page package, use the commands:

```
cd doc/src
gmake man.tar.gz
```

which will result in a tar file being generated in the doc/src directory.

To generate quality man pages, it might be necessary to use a hacked version of the conversion utility or do some manual postprocessing. All man pages should be manually inspected before distribution.
I.3.3. Print Output via JadeTeX

If you want to use JadeTeX to produce a printable rendition of the documentation, you can use one of the following commands:

- To generate PostScript via DVI in A4 format:
  
  \texttt{doc/src/sgml\$ gmake postgres-A4.ps}

  In U.S. letter format:

  \texttt{doc/src/sgml\$ gmake postgres-US.ps}

- To make a PDF:
  
  \texttt{doc/src/sgml\$ gmake postgres-A4.pdf}
  
  or

  \texttt{doc/src/sgml\$ gmake postgres-US.pdf}

  (Of course you can also make a PDF version from the PostScript, but if you generate PDF directly, it will have hyperlinks and other enhanced features.)

When using JadeTeX to build the PostgreSQL documentation, you will probably need to increase some of TeX’s internal parameters. These can be set in the file \texttt{texmf.cnf}. The following settings worked at the time of this writing:

\begin{verbatim}
hash_extra.jadetex = 200000
hash_extra.pdfjadetex = 200000
pool_size.jadetex = 2000000
pool_size.pdfjadetex = 2000000
string_vacancies.jadetex = 150000
string_vacancies.pdfjadetex = 150000
max_strings.jadetex = 300000
max_strings.pdfjadetex = 300000
save_size.jadetex = 15000
save_size.pdfjadetex = 15000
\end{verbatim}

I.3.4. Print Output via RTF

You can also create a printable version of the PostgreSQL documentation by converting it to RTF and applying minor formatting corrections using an office suite. Depending on the capabilities of the particular office suite, you can then convert the documentation to PostScript or PDF. The procedure below illustrates this process using Applixware.

\textbf{Note:} It appears that current versions of the PostgreSQL documentation trigger some bug in or exceed the size limit of OpenJade. If the build process of the RTF version hangs for a long time and the output file still has size 0, then you might have hit that problem. (But keep in mind that a normal build takes 5 to 10 minutes, so don’t abort too soon.)
Appendix I. Documentation

Applixware RTF Cleanup

OpenJade omits specifying a default style for body text. In the past, this undiagnosed problem led to a long process of table of contents generation. However, with great help from the Applixware folks the symptom was diagnosed and a workaround is available.

1. Generate the RTF version by typing:
   
   ```
   doc/src/sgml$ gmake postgres.rtf
   ```

2. Repair the RTF file to correctly specify all styles, in particular the default style. If the document contains `refentry` sections, one must also replace formatting hints which tie a preceding paragraph to the current paragraph, and instead tie the current paragraph to the following one. A utility, `fixrtf`, is available in `doc/src/sgml` to accomplish these repairs:
   
   ```
   doc/src/sgml$ ./fixrtf --refentry postgres.rtf
   ```

   The script adds `\s0 Normal\` as the zeroth style in the document. According to Applixware, the RTF standard would prohibit adding an implicit zeroth style, though Microsoft Word happens to handle this case. For repairing `refentry` sections, the script replaces `\keepn` tags with `\keep`.

3. Open a new document in Applixware Words and then import the RTF file.
4. Generate a new table of contents (ToC) using Applixware.
   
   a. Select the existing ToC lines, from the beginning of the first character on the first line to the last character of the last line.
   
   b. Build a new ToC using Tools→Book Building→Create Table of Contents. Select the first three levels of headers for inclusion in the ToC. This will replace the existing lines imported in the RTF with a native Applixware ToC.
   
   c. Adjust the ToC formatting by using Format→Style, selecting each of the three ToC styles, and adjusting the indents for First and Left. Use the following values:

<table>
<thead>
<tr>
<th>Style</th>
<th>First Indent (inches)</th>
<th>Left Indent (inches)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOC-Heading 1</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>TOC-Heading 2</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>TOC-Heading 3</td>
<td>1.2</td>
<td>1.2</td>
</tr>
</tbody>
</table>

5. Work through the document to:
   
   - Adjust page breaks.
   - Adjust table column widths.

6. Replace the right-justified page numbers in the Examples and Figures portions of the ToC with correct values. This only takes a few minutes.

7. Delete the index section from the document if it is empty.

8. Regenerate and adjust the table of contents.
   
   a. Select the ToC field.
   
   b. Select Tools→Book Building→Create Table of Contents.
   
   c. Unbind the ToC by selecting Tools→Field Editing→Unprotect.
Appendix I. Documentation

d. Delete the first line in the ToC, which is an entry for the ToC itself.

9. Save the document as native Applixware Words format to allow easier last minute editing later.

10. “Print” the document to a file in PostScript format.

I.3.5. Plain Text Files

Several files are distributed as plain text, for reading during the installation process. The INSTALL file corresponds to Chapter 15, with some minor changes to account for the different context. To recreate the file, change to the directory doc/src/sgml and enter gmake INSTALL. This will create a file INSTALL.html that can be saved as text with Netscape Navigator and put into the place of the existing file. Netscape seems to offer the best quality for HTML to text conversions (over lynx and w3m).

The file HISTORY can be created similarly, using the command gmake HISTORY. For the file src/test/regress/README the command is gmake regress_README.

I.3.6. Syntax Check

Building the documentation can take very long. But there is a method to just check the correct syntax of the documentation files, which only takes a few seconds:

doc/src/sgml$ gmake check

I.4. Documentation Authoring

SGML and DocBook do not suffer from an oversupply of open-source authoring tools. The most common tool set is the Emacs/XEmacs editor with appropriate editing mode. On some systems these tools are provided in a typical full installation.

I.4.1. Emacs/PSGML

PSGML is the most common and most powerful mode for editing SGML documents. When properly configured, it will allow you to use Emacs to insert tags and check markup consistency. You could use it for HTML as well. Check the PSGML web site\[15\] for downloads, installation instructions, and detailed documentation.

There is one important thing to note with PSGML: its author assumed that your main SGML DTD directory would be /usr/local/lib/sgml. If, as in the examples in this chapter, you use /usr/local/share/sgml, you have to compensate for this, either by setting SGML_CATALOG_FILES environment variable, or you can customize your PSGML installation (its manual tells you how).

Put the following in your ~/.emacs environment file (adjusting the path names to be appropriate for your system):

\[15\] http://www.lysator.liu.se/projects/about_psgml.html
Appendix I. Documentation

; ********** for SGML mode (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-default-dtd-file "/reference.ced")
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files "/usr/local/share/sgml/catalog")
(setq sgml-ecat-files nil)

/autoload ’sgml-mode "psgml" "Major mode to edit SGML files." t )

and in the same file add an entry for SGML into the (existing) definition for auto-mode-alist:

(setq auto-mode-alist
’(’”\.sgml$" . sgml-mode)
)

The PostgreSQL distribution includes a parsed DTD definitions file reference.ced. You might find that when using PSGML, a comfortable way of working with these separate files of book parts is to insert a proper DOCTYPE declaration while you’re editing them. If you are working on this source, for instance, it is an appendix chapter, so you would specify the document as an “appendix” instance of a DocBook document by making the first line look like this:

<!DOCTYPE appendix PUBLIC "+//OASIS//DTD DocBook V4.2//EN”>

This means that anything and everything that reads SGML will get it right, and I can verify the document with nsgmls -s docguide.sgml. (But you need to take out that line before building the entire documentation set.)

I.4.2. Other Emacs modes

GNU Emacs ships with a different SGML mode, which is not quite as powerful as PSGML, but it’s less confusing and lighter weight. Also, it offers syntax highlighting (font-lock), which can be very helpful.

Norm Walsh offers a major mode16 specifically for DocBook which also has font-lock and a number of features to reduce typing.

---

I.5. Style Guide

I.5.1. Reference Pages

Reference pages should follow a standard layout. This allows users to find the desired information more quickly, and it also encourages writers to document all relevant aspects of a command. Consistency is not only desired among PostgreSQL reference pages, but also with reference pages provided by the operating system and other packages. Hence the following guidelines have been developed. They are for the most part consistent with similar guidelines established by various operating systems.

Reference pages that describe executable commands should contain the following sections, in this order. Sections that do not apply can be omitted. Additional top-level sections should only be used in special circumstances; often that information belongs in the “Usage” section.

Name

This section is generated automatically. It contains the command name and a half-sentence summary of its functionality.

Synopsis

This section contains the syntax diagram of the command. The synopsis should normally not list each command-line option; that is done below. Instead, list the major components of the command line, such as where input and output files go.

Description

Several paragraphs explaining what the command does.

Options

A list describing each command-line option. If there are a lot of options, subsections can be used.

Exit Status

If the program uses 0 for success and non-zero for failure, then you do not need to document it. If there is a meaning behind the different non-zero exit codes, list them here.

Usage

Describe any sublanguage or run-time interface of the program. If the program is not interactive, this section can usually be omitted. Otherwise, this section is a catch-all for describing run-time features. Use subsections if appropriate.

Environment

List all environment variables that the program might use. Try to be complete; even seemingly trivial variables like SHELL might be of interest to the user.

Files

List any files that the program might access implicitly. That is, do not list input and output files that were specified on the command line, but list configuration files, etc.

Diagnostics

Explain any unusual output that the program might create. Refrain from listing every possible error message. This is a lot of work and has little use in practice. But if, say, the error messages have a standard format that the user can parse, this would be the place to explain it.
Appendix I. Documentation

Notes

Anything that doesn’t fit elsewhere, but in particular bugs, implementation flaws, security considerations, compatibility issues.

Examples

Examples

History

If there were some major milestones in the history of the program, they might be listed here. Usually, this section can be omitted.

See Also

Cross-references, listed in the following order: other PostgreSQL command reference pages, PostgreSQL SQL command reference pages, citation of PostgreSQL manuals, other reference pages (e.g., operating system, other packages), other documentation. Items in the same group are listed alphabetically.

Reference pages describing SQL commands should contain the following sections: Name, Synopsis, Description, Parameters, Outputs, Notes, Examples, Compatibility, History, See Also. The Parameters section is like the Options section, but there is more freedom about which clauses of the command can be listed. The Outputs section is only needed if the command returns something other than a default command-completion tag. The Compatibility section should explain to what extent this command conforms to the SQL standard(s), or to which other database system it is compatible. The See Also section of SQL commands should list SQL commands before cross-references to programs.
Appendix J. Acronyms

This is a list of acronyms commonly used in the PostgreSQL documentation and in discussions about PostgreSQL.

ANSI
American National Standards Institute

API
Application Programming Interface

ASCII
American Standard Code for Information Interchange

BKI
Backend Interface

CA
Certificate Authority

CIDR
Classless Inter-Domain Routing

CPAN
Comprehensive Perl Archive Network

CRL
Certificate Revocation List

CSV
Comma Separated Values

CVE
Common Vulnerabilities and Exposures

CVS
Concurrent Versions System

DBA
Database Administrator

Appendix J. Acronyms

DBI
Database Interface (Perl)\(^\text{12}\)

DBMS
Database Management System\(^\text{13}\)

DDL
Data Definition Language\(^\text{14}\), SQL commands such as `CREATE TABLE, ALTER USER`

DML
Data Manipulation Language\(^\text{15}\), SQL commands such as `INSERT, UPDATE, DELETE`

DST
Daylight Saving Time\(^\text{16}\)

ECPG
Embedded C for PostgreSQL

ESQL
Embedded SQL\(^\text{17}\)

FAQ
Frequently Asked Questions\(^\text{18}\)

FSM
Free Space Map

GEQO
Genetic Query Optimizer

GIN
Generalized Inverted Index

GiST
Generalized Search Tree

GMT
Greenwich Mean Time\(^\text{19}\)

GSSAPI
Generic Security Services Application Programming Interface\(^\text{20}\)

GUC
Grand Unified Configuration, the PostgreSQL subsystem that handles server configuration

HBA
Host-Based Authentication

\(^{12}\) http://dbi.perl.org/
\(^{13}\) http://en.wikipedia.org/wiki/Dbms
\(^{14}\) http://en.wikipedia.org/wiki/Data_Definition_Language
\(^{15}\) http://en.wikipedia.org/wiki/Data_Manipulation_Language
\(^{16}\) http://en.wikipedia.org/wiki/Daylight_saving_time
\(^{17}\) http://en.wikipedia.org/wiki/Embedded_SQL
\(^{18}\) http://en.wikipedia.org/wiki/FAQ
\(^{19}\) http://en.wikipedia.org/wiki/GMT
\(^{20}\) http://en.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface
Appendix J. Acronyms

HOT
Heap-Only Tuples

IEC
International Electrotechnical Commission

IEEE
Institute of Electrical and Electronics Engineers

IPC
Inter-Process Communication

ISO
International Standards Organization

ISSN
International Standard Serial Number

JDBC
Java Database Connectivity

LDAP
Lightweight Directory Access Protocol

MSVC
Microsoft Visual C

MVCC
Multi-Version Concurrency Control

NLS
National Language Support

ODBC
Open Database Connectivity

OID
Object Identifier

OLAP
Online Analytical Processing

OLTP
Online Transaction Processing

Appendix J. Acronyms

ORDBMS
Object-Relational Database Management System

PAM
Pluggable Authentication Modules

PGSQL
PostgreSQL

PGXS
PostgreSQL Extension System

PID
Process Identifier

PITR
Point-In-Time Recovery (Continuous Archiving)

PL
Programming Languages (server-side)

POSIX
Portable Operating System Interface

RDBMS
Relational Database Management System

RFC
Request For Comments

SGML
Standard Generalized Markup Language

SPI
Server Programming Interface

SQL
Structured Query Language

SRF
Set-Returning Function

SSH
Secure Shell

SSL
Secure Sockets Layer

SSPI
    Security Support Provider Interface\textsuperscript{44}
SYSV
    Unix System V\textsuperscript{45}
TCP/IP
    Transmission Control Protocol (TCP) / Internet Protocol (IP)\textsuperscript{46}
TID
    Tuple Identifier
TOAST
    The Oversized-Attribute Storage Technique
TPC
    Transaction Processing Performance Council\textsuperscript{47}
URL
    Uniform Resource Locator\textsuperscript{48}
UTC
    Coordinated Universal Time\textsuperscript{49}
UTF
    Unicode Transformation Format\textsuperscript{50}
UTF8
    Eight-Bit Unicode Transformation Format\textsuperscript{51}
UUID
    Universally Unique Identifier
WAL
    Write-Ahead Log
XID
    Transaction Identifier
XML
    Extensible Markup Language\textsuperscript{52}

\textsuperscript{45}. http://en.wikipedia.org/wiki/System_V
\textsuperscript{46}. http://en.wikipedia.org/wiki/Transmission_Control_Protocol
\textsuperscript{47}. http://www.tpc.org/
\textsuperscript{48}. http://en.wikipedia.org/wiki/URL
\textsuperscript{49}. http://en.wikipedia.org/wiki/Coordinated_Universal_Time
\textsuperscript{50}. http://www.unicode.org/
\textsuperscript{51}. http://en.wikipedia.org/wiki/Utf8
\textsuperscript{52}. http://en.wikipedia.org/wiki/XML
Bibliography

Selected references and readings for SQL and PostgreSQL.

Some white papers and technical reports from the original POSTGRES development team are available at the University of California, Berkeley, Computer Science Department web site.

SQL Reference Books


PostgreSQL-Specific Documentation


Discusses SQL history and syntax, and describes the addition of INTERSECT and EXCEPT constructs into PostgreSQL. Prepared as a Master’s Thesis with the support of O. Univ. Prof. Dr. Georg Gottlob and Univ. Ass. Mag. Katrin Seyr at Vienna University of Technology.


Zelaine Fong, *The design and implementation of the POSTGRES query optimizer*, University of California, Berkeley, Computer Science Department.

Bibliography

Proceedings and Articles


M. Stonebraker, L. A. Rowe, and M. Hirohama, “The implementation of POSTGRES”, *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, March 1990.


autovacuum_analyze_scale_factor configuration parameter, 375
autovacuum_analyze_threshold configuration parameter, 375
autovacuum_freeze_max_age configuration parameter, 375
autovacuum_max_workers configuration parameter, 374
autovacuum_naptime configuration parameter, 374
autovacuum_vacuum_cost_delay configuration parameter, 375
autovacuum_vacuum_cost_limit configuration parameter, 375
autovacuum_vacuum_scale_factor configuration parameter, 375
autovacuum_vacuum_threshold configuration parameter, 374
average, 12, 207

B
B-tree
(see index)
backslash escapes, 25
backslash_quote configuration parameter, 381
backup, 225, 426
base type, 630
BEGIN, 909
BETWEEN, 132
BETWEEN SYMMETRIC, 132
bgwriter_delay configuration parameter, 357
bgwriter_lru_maxpages configuration parameter, 358
bgwriter_lru_multiplier configuration parameter, 358
bitmap scan, 244, 362
bit_and, 207
bit_length, 137
bit_or, 207
BLOB
(see large object)
block_size configuration parameter, 383
bonjour_name configuration parameter, 351
Boolean
data type, 104
operators
(see operators, logical)
bool_and, 207
bool_or, 207
booting
starting the server during, 333
box (data type), 108
BSD/OS
IPC configuration, 337
shared library, 654
btree_gist, 1798
btrim, 138
bytea, 95
in libpq, 498

C
C, 476, 538
canceling
SQL command, 503
CASCADE
with DROP, 68
foreign key action, 47
CASE, 203
determination of result type, 237
case sensitivity
do of SQL commands, 24
cast
I/O conversion, 933
cbrt, 134
ceiling, 134
center, 184
char, 93
character, 93
character set, 378, 384, 412
character string
concatenation, 137
constant, 24
data types, 93
length, 137
character varying, 93
char_length, 137
check constraint, 41
checkpoint, 466, 911
checkpoint_completion_target configuration parameter, 361
checkpoint_segments configuration parameter, 360
checkpoint_timeout configuration parameter, 361
checkpoint_warning configuration parameter, 361
ccheck_function_bodies configuration parameter, 377
chkpass, 1798
chr, 138
cid, 128
cidr, 110
circle, 109
client authentication, 398
timeout during, 352
client_encoding configuration parameter, 378
client_min_messages configuration parameter, 367
clock_timestamp, 171
CLOSE, 912
CLUSTER, 914
of databases
(see database cluster)
clusterdb, 1162
clustering, 445
cmax, 48
cmin, 48
COALESCE, 204
column, 6, 39
adding, 50
removing, 50
renaming, 52
system column, 48
column data type
changing, 51
column reference, 31
col_description, 222
COMMENT, 917
about database objects, 222
in SQL, 29
COMMIT, 920
COMMIT PREPARED, 921
commit_delay configuration parameter, 360
commit_siblings configuration parameter, 360
comparison
operators, 131
row-wise, 213
subquery result row, 210
compiling
libpq applications, 518
composite type, 124, 630
constant, 125
constructor, 36
computed field, 637
concurrency, 288
conditional expression, 203
configuration
of the server, 348
of the server
functions, 224
configure, 315
config_file configuration parameter, 349
conjunction, 131
connection service file, 515
constant, 24
constraint, 41
adding, 50
check, 41
foreign key, 45
name, 41
NOT NULL, 43
primary key, 45
removing, 51
unique, 44
constraint exclusion, 65, 364
constraint_exclusion configuration parameter, 364
CONTINUE
in PL/pgSQL, 741
continuous archiving, 426
convert, 138
convert_from, 138
convert_to, 138
COPY, 8, 922
with libpq, 506
correlation, 209
cos, 136
cot, 136
count, 12
covariance
population, 209
sample, 209
cpu_index_tuple_cost configuration parameter, 363
cpu_operator_cost configuration parameter, 363
cpu_tuple_cost configuration parameter, 363
CREATE DATABASE, 393
CREATE AGGREGATE, 930
CREATE CAST, 933
CREATE CONSTRAINT TRIGGER, 937
CREATE CONVERSION, 939
CREATE DATABASE, 941
CREATE DOMAIN, 944
CREATE FUNCTION, 946
CREATE GROUP, 952
CREATE INDEX, 953
CREATE LANGUAGE, 958
CREATE OPERATOR, 961
CREATE OPERATOR CLASS, 964
CREATE OPERATOR FAMILY, 967
CREATE ROLE, 388, 969
CREATE RULE, 974
CREATE SCHEMA, 977
CREATE SEQUENCE, 979
CREATE TABLE, 6, 983
CREATE TABLE AS, 994
CREATE TABLESPACE, 396, 997
CREATE TEXT SEARCH CONFIGURATION, 999
CREATE TEXT SEARCH DICTIONARY, 1001
CREATE TEXT SEARCH PARSER, 1003
CREATE TEXT SEARCH TEMPLATE, 1005
CREATE TRIGGER, 1007
CREATE TYPE, 1010
CREATE USER, 1017
CREATE VIEW, 1018
createdb, 2, 394, 1165
createlang, 1168
createuser, 388, 1171
cross compilation, 320
cross join, 73
crypt, 404
cstring, 129
c(cid), 48, 704
cube, 1799
current_database, 217
current_date, 171
current_schema, 217
current_time, 171
current_timestamp, 171
current_user, 217
currval, 201
cursor
  CLOSE, 912
  DECLARE, 1022
  FETCH, 1076
  in PL/pgSQL, 745
  MOVE, 1095
  showing the query plan, 1073
custom_variable_classes configuration parameter, 384
data area
  (see database cluster)
data partitioning, 445
data type, 87
    base, 630
    category, 231
    composite, 630
    constant, 27
    conversion, 230
    enumerated (enum), 105
    internal organization, 645
    numeric, 88
    type cast, 34
    user-defined, 669
database, 393
    creating, 2
    privilege to create, 389
database activity
    monitoring, 449
database cluster, 6, 331
data_directory configuration parameter, 349
date, 96, 98
    constants, 101
    current, 179
    output format, 101
    (see also formatting)
DateStyle configuration parameter, 378
date_part, 171, 174
date_trunc, 171, 178
DBI, 783
dblink, 1803
db_user_namespace configuration parameter, 353
deadlock, 295
deadlock_timeout configuration parameter, 380
DEALLOCATE, 1021
debug_assertions configuration parameter, 385
1946
debug_pretty_print configuration parameter, 369
debug_print_parse configuration parameter, 369
debbug_print_plan configuration parameter, 369
debbug_print_rewritten configuration parameter, 369
decimal
(see numeric)
DECLARE, 1022
decode, 138
default value, 40
   changing, 51
default_statistics_target configuration parameter, 364
default_tablespace configuration parameter, 376
default_text_search_config configuration parameter, 379
default_transaction_isolation configuration parameter, 377
default_transaction_read_only configuration parameter, 377
default_with_oids configuration parameter, 382
degrees, 134
delay, 181
DELETE, 14, 71, 1025
deleting, 71
diameter, 184
dict_int, 1834
dict_xsyn, 1834
Digital UNIX
   (see Tru64 UNIX)
dirty read, 288
DISCARD, 1028
disjunction, 131
disk drive, 468
disk space, 419
disk usage, 461
DISTINCT, 9, 83
document
text search, 252
dollar quoting, 25
double precision, 90
DROP AGGREGATE, 1029
DROP CAST, 1031
DROP CONVERSION, 1033
DROP DATABASE, 396, 1034
DROP DOMAIN, 1035
DROP FUNCTION, 1036
DROP GROUP, 1038
DROP INDEX, 1039
DROP LANGUAGE, 1040
DROP OPERATOR, 1041
DROP OPERATOR CLASS, 1043
DROP OPERATOR FAMILY, 1045
DROP OWNED, 1047
DROP ROLE, 388, 1049
DROP RULE, 1051
DROP SCHEMA, 1053
DROP SEQUENCE, 1055
DROP TABLE, 7, 1056
DROP TABLESPACE, 1058
DROP TEXT SEARCH CONFIGURATION, 1060
DROP TEXT SEARCH DICTIONARY, 1062
DROP TEXT SEARCH PARSER, 1063
DROP TEXT SEARCH TEMPLATE, 1064
DROP TRIGGER, 1065
DROP TYPE, 1067
DROP USER, 1068
DROP VIEW, 1069
dropdb, 396, 1175
droplang, 1178
dropuser, 388, 1181
DTD, 115
DTrace, 320, 457
duplicate, 9
duplicates, 83
dynamic loading, 379, 644
dynamic_library_path, 645
dynamic_library_path configuration parameter, 379
dynamic_distance, 1835
ECPG, 538, 1184
effective_cache_size configuration parameter, 363
elog, 1365
   in PL/Perl, 785
   in PL/Python, 795
   in PL/Tcl, 776
embedded SQL
   in C, 538
enabled role, 601
enable BitmapScan configuration parameter, 362
enable_hashagg configuration parameter, 362
enable_hashjoin configuration parameter, 362
enable_indexscan configuration parameter, 362
enable_mergejoin configuration parameter, 362
enable_nestloop configuration parameter, 362
enable_seqscan configuration parameter, 362
enable_sort configuration parameter, 362
enable_tidscan configuration parameter, 362
code, 138
crypt, 343
END, 1070
environment variable, 513
ereport, 1365
error codes
libpq, 491
list of, 1416
error message, 484
escape string syntax, 25
escape_string_warning configuration parameter, 382
escaping strings
in libpq, 497
every, 207
EXCEPT, 83
exceptions
in PL/PgSQL, 744
EXECUTE, 1071
EXISTS, 210
EXIT
in PL/pgSQL, 740
exp, 134
EXPLAIN, 299, 1073
explain_pretty_print configuration parameter, 379
expression
order of evaluation, 38
syntax, 30
extending SQL, 630
extensions, 1922
external_pid_file configuration parameter, 350
extract, 171, 174
extra_float_digits configuration parameter, 378
failover, 445
false, 104
FAQ, xlvii
fast path, 504
FETCH, 1076
field
computed, 637
field selection, 32
flex, 314
float4
(see real)
float8
(see double precision)
floating point, 90
floating-point
display, 378
floor, 134
foreign key, 15, 45
formatting, 164
format_type, 220
free space map, 355
FreeBSD
IPC configuration, 338
shared library, 654
start script, 333
FROM
missing, 381
from_collapse_limit configuration parameter, 365
fsync configuration parameter, ??
full text search, 251
data types, 112
functions and operators, 112
full_page_writes configuration parameter, ??
function, 131
in the FROM clause, 78
internal, 643
invocation, 33
output parameter, 637
polymorphic, 631
type resolution in an invocation, 234
user-defined, 632
in C, 644
in SQL, 632
fuzzystrmatch, 1837
generate_series, 216
geqo configuration parameter, 364
geqo_effort configuration parameter, 364
geqo_generations configuration parameter, 364
geqo_pool_size configuration parameter, 364
geqo_selection_bias configuration parameter, 364
geqo_threshold configuration parameter, 364
get_bit, 148
get_byte, 148
GIN
 gin_fuzzy_search_limit configuration parameter, 380
GiST
 global data
 in PL/Python, 795
 in PL/Tcl, 774
GRANT, 390, 1080
GREATEST, 205
determination of result type, 237
GROUP BY, 79
GSSAPI, 404
GUID, 114
hash
 (see index)
has_database_privilege, 219
has_function_privilege, 219
has_language_privilege, 219
has_schema_privilege, 219
has_tablespace_privilege, 219
has_table_privilege, 219
HAVING, 13, 81
hba_file configuration parameter, 349
height, 184
hierarchical database, 6
high availability, 426, 445
history
INSERT, 7, 69, 1086
inserting, 69
installation, 312
on Windows, 326
instr, 765
int2
(see smallint)
int4
(see integer)
int8
(see bigint)
intagg, 1842
intarray, 1844
integer, 27, 89
integer_datetimes configuration parameter, 383
interfaces
externally maintained, 1921
internal, 129
INTERSECT, 83
interval, 96, 100
ipcclean, 1244
IRIX
shared library, 654
IS DISTINCT FROM, 133, 213
IS DOCUMENT, 197
IS FALSE, 133
IS NOT DISTINCT FROM, 133, 213
IS NOT FALSE, 133
IS NOT NULL, 132
IS NOT TRUE, 133
IS NOT UNKNOWN, 133
IS NULL, 132, 383
IS TRUE, 133
IS UNKNOWN, 133
isclosed, 184
isfinite, 171
isnan, 1847
ISNULL, 132
isopen, 184
join, 10, 73
controlling the order, 305
cross, 73
left, 74
natural, 74
outer, 11, 73
right, 74
self, 11
join_collapse_limit configuration parameter, 365
justify_days, 171
justify_hours, 171
justify_interval, 171
Kerberos, 404
key word
list of, 1430
syntax of, 23
krb_caseins_users configuration parameter, 353
krb_REALM configuration parameter, 352
krb_server_hostname configuration parameter, 352
krb_server_keyfile configuration parameter, 352
krb_srvname configuration parameter, 353
label
(see alias)
language_handler, 129
large object, 528
lastval, 201
lc_collate configuration parameter, 383
lc_cctype configuration parameter, 383
lc_messages configuration parameter, 378
lc_monetary configuration parameter, 379
lc_numeric configuration parameter, 379
lc_time configuration parameter, 379
LDAP, 318, 407
LDAP connection parameter lookup, 515
ldconfig, 324
LEAST, 205
determination of result type, 237
left join, 74
length, 184
of a binary string
(see binary strings, length)
of a character string
(see character string, length)
length(tsvector), 262
libedit, 313
libperl, 313
MAC address
(see macaddr)
macaddr (data type), 111
MacOS X
IPC configuration, 339
shared library, 654
magic block, 644
maintenance, 419
maintenance_work_mem configuration parameter, 354
make, 312
MANPATH, 324
max, 12
max_connections configuration parameter, 350
max_files_per_process configuration parameter, 356
max_fsm_pages configuration parameter, 355
max_fsm_relations configuration parameter, 355
max_function_args configuration parameter, 384
max_identifier_length configuration parameter, 384
max_index_keys configuration parameter, 384
max_locks_per_transaction configuration parameter, 381
max_prepared_transactions configuration parameter, 354
max_stack_depth configuration parameter, 355
md5, 138, 404
memory context
in SPI, 832
min, 12
mod, 134
monitoring
database activity, 449
MOVE, 1095
MVCC, 288

NaN
(see not a number)
natural join, 74
negation, 131
NetBSD
IPC configuration, 338
shared library, 655
start script, 333
network
data types, 109
Network Attached Storage (NAS)
(see Network File Systems)
Network File Systems, 332
nextval, 201
NFS
(see Network File Systems)
onblocking connection, 479, 499
nonrepeatable read, 288
NOT (operator), 131
not a number
double precision, 91
numeric (data type), 90
NOT IN, 210, 213
not-null constraint, 43
notice processor, 512
notice receiver, 512
notice processing
in libpq, 512
NOTIFY, 1097
in libpq, 505
NOTNULL, 132
now, 171
npoints, 184
null value
with check constraints, 43
comparing, 132
default value, 40
in DISTINCT, 83
in libpq, 495
in PL/Perl, 781
PL/Python, 792
with unique constraints, 44
NULLIF, 205
number
constant, 27
numeric, 27
numeric (data type), 89
numnode, 263
NVL, 204

name
qualified, 53
syntax of, 23
unqualified, 55
object identifier
data type, 128
object-oriented database, 6
obj_description, 222
octet_length, 137
OFFSET, 85
oid, 128
column, 48
in libpq, 497
oid2name, 1858
ONLY, 73
opaque, 129
OpenBSD
IPC configuration, 338
shared library, 655
start script, 333
OpenSSL, 318
(see also SSL)
operator, 131
invocation, 33
logical, 131
precedence, 29
syntax, 28
type resolution in an invocation, 231
user-defined, 673
operator class, 248, 678
operator family, 248, 684
OR (operator), 131
Oracle
porting from PL/SQL to PL/pgSQL, 762
ORDER BY, 9, 84
and locales, 411
ordering operator, 686
outer join, 73
output function
of a data type, 669
output function, 669
overlay, 137
overloading
functions, 641
operators, 673
owner, 390

pageinspect, 1862
palloc, 653
PAM, 318, 408
parameter

syntax, 32
parenthesis, 31
partitioning, 60
password, 389
  authentication, 404
  of the superuser, 332
password file, 515
password_encryption configuration parameter, 352
PATH, 324
  for schemas, 375
path (data type), 108
pattern matching, 150
patterns
  in psql and pg_dump, 1223
pclose, 184
performance, 299
Perl, 780
permission
  (see privilege)
pfree, 653
pgbench, 1863
PGcancel, 503
PGCLIENTENCODING, 514
PGconn, 476
PGCONNECT_TIMEOUT, 514
pgcrypto, 1870
PGDATA, 331
PGDATABASE, 513
PGDATESTYLE, 514
PGEGO, 514
PGGSSLIB, 514
PGHOST, 513
PGHOSTADDR, 513
PGKRB5SRVNAME, 514
PGLOCALDIR, 514
PGOPTIONS, 514
PGPASSFILE, 513
PGPASSWORD, 513
PGPORT, 513
PGREALM, 514
PGREQUIRESSL, 514
PGresult, 489
pgrowlocks, 1884
PGSERVICE, 514
PGSSLKEY, 514
PGSSLMODE, 514
pgstatuser, 1889
PGSYSCONFDIR, 514
PGTZ, 514
PGUSER, 513
pgxs, 655
pg_trgm, 1891
pg_trigger, 1307
pg_try_advisory_lock, 229
pg_try_advisory_lock_shared, 229
pg_ts_config, 1309
pg_ts_config_is_visible, 220
pg_ts_config_map, 1309
pg_ts_dict, 1310
pg_ts_dict_is_visible, 220
pg_ts_parser, 1310
pg_ts_parser_is_visible, 220
pg_ts_template, 1311
pg_ts_template_is_visible, 220
pg_type, 1311
pg_type_is_visible, 220
pg_user, 1332
pg_views, 1332
pg_xlogfile_name, 225
pg_xlogfile_name_offset, 225
phantom read, 288
pi, 134
PIC, 653
PID
determining PID of server process
in libpq, 485
PITR, 426
PITR standby, 426
PL/Perl, 780
PL/PerlU, 787
PL/pgSQL, 723
PL/Python, 791
PL/SQL (Oracle)
porting to PL/pgSQL, 762
PL/Tcl, 772
plainto_tsquery, 258
point, 107
point-in-time recovery, 426
polygon, 109
polymorphic function, 631
polymorphic type, 631
popen, 184
port, 477
port configuration parameter, 350
position, 137
POSTGRES, xl, 1, 332, 394, 1253
postgres user, 331
Postgres95, xl
postgresql.conf, 348
postmaster, 1260
post_auth_delay configuration parameter, 385
power, 134
PQbackendPID, 485
PQbinaryTuples, 494
with COPY, 507
PQcancel, 504
PQclear, 492
PQclientEncoding, 510
PQcmdStatus, 496
PQcmdTuples, 496
PQcommdefaults, 481
PQconnectdb, 476
PQconnectionNeedsPassword, 485
PQconnectionUsedPassword, 485
PQconnectPoll, 479
PQconnectStart, 479
PQconsumeInput, 502
PQdb, 482
PQdescribePortal, 489
PQdescribePrepared, 489
PQencryptPassword, 512
PQendcopy, 510
PQerrorMessage, 484
PQescapeBytea, 499
PQescapeByteaConn, 498
PQescapeString, 497
PQescapeStringConn, 497
PQexec, 486
PQexecParams, 486
PQexecPrepared, 488
PQfformat, 494
with COPY, 507
PQfinish, 481
PQflush, 503
PQfmod, 494
PQfn, 504
PQfnumber, 493
PQftime, 493
PQftime, 493
PQfreeCancel, 504
PQfreetem, 499
PQfsiz, 494
PQtable, 493
PQtablecol, 493
PQftype, 494
PQgetCancel, 503
PQgetCopyData, 508
PQgetisnull, 495
PQgetlength, 495
PQgetline, 509
PQgetLineAsync, 509
PQgetresult, 501
PQgetssl, 485
PQgetvalue, 495
PQhost, 482
PQisBusy, 502
PQisnonblocking, 503
PQisthreadsafe, 517
PQmakeEmptyPGresult, 492
PQnfields, 492
PQnotifies, 505
PQnparams, 495
PQntuples, 492
PQoidStatus, 497
PQoidValue, 497
PQoptions, 483
PQparameterStatus, 483
PQparamtype, 495
PQpass, 482
PQport, 482
PQprepare, 488
PQprint, 496
PQprotocolVersion, 484
PQputCopyData, 507
PQputCopyEnd, 507
PQputline, 509
PQputnbytes, 510
PQrequestCancel, 504
PQreset, 481
PQresetPoll, 481
PQresetStart, 481
PQresultStatus, 490
PQresultErrorField, 490
PQresultErrorMessage, 490
PQresStatus, 489
PQsendDescribePortal, 501
PQsendDescribePrepared, 501
PQsendPrepare, 500
PQsendQuery, 500
PQsendQueryParams, 500
PQsendQueryPrepared, 501
PQserverVersion, 484
PQsetClientEncoding, 511
PQsetdb, 478
PQsetdbLogin, 478
PQsetErrorVerbosity, 511
PQsetnonlocking, 503
PQsetNoticeProcessor, 512
PQsetNoticeReceiver, 512
PQsocket, 485
PQstatus, 483
PQtrace, 511
PQtransactionStatus, 483
PQtty, 483
PQunescapeBytea, 499
PQuntrace, 511
PQuser, 482
predicate locking, 291
PREPARE, 1099
PREPARE TRANSACTION, 1101
prepared statements
creating, 1099
executing, 1071
removing, 1021
showing the query plan, 1073
preparing a query
in PL/Tcl, 775
in PL/pgSQL, 758
in PL/Python, 796
pre_auth_delay configuration parameter, 385
primary key, 45
privilege, 52, 390
querying, 218
with rules, 716
for schemas, 56
with views, 716
procedural language, 721
externally maintained, 1922
handler for, 1377
protocol
frontend-backend, 1334
ps
to monitor activity, 449
psql, 3, 1209
Python, 791
qualified name, 53
query, 8, 72
query plan, 299
query tree, 697
querytree, 263
quotation marks
and identifiers, 24
escaping, 24
quote_ident, 138
use in PL/PgSQL, 734
quote_literal, 138
use in PL/PgSQL, 734
radians, 134
radius, 184
RAISE, 751
random, 134
random_page_cost configuration parameter, 363
range table, 697
read-only transaction, ??
readline, 312
real, 90
REASSIGN OWNED, 1103
record, 129
rectangle, 108
referential integrity, 15, 45
regclass, 128
regconfig, 128
regdictionary, 128
regexp_matches, 152
regexp_replace, 152
regexp_split_to_array, 152
regexp_split_to_table, 152
regex_flavor configuration parameter, 382
regoper, 128
regoperator, 128
regproc, 128
regprocedure, 128
regression intercept, 209
regression slope, 209
regression test, 322
regression tests, 469
regtype, 128
regular expression, 151, 152
(see also pattern matching)
regular expressions, 382
reindex, 424, 1104
reindexdb, 1234
relation, 6
relational database, 6
RELEASE SAVEPOINT, 1107
repeat, 138
replace, 138
replication, 445
reporting errors
in PL/PgSQL, 751
RESET, 1109
RESTRICT
with DROP, 68
foreign key action, 47
RETURN NEXT
in PL/PgSQL, 737
RETURN QUERY
in PL/PgSQL, 737
RETURNING INTO
in PL/pgSQL, 732
REVOKE, 390, 1111
right join, 74
role, 388
applicable, 582
enabled, 601
membership in, 390
privilege to create, 389
ROLLBACK, 1114
psql, 1227
ROLLBACK PREPARED, 1115
ROLLBACK TO SAVEPOINT, 1116
round, 134
routine maintenance, 419
row, 6, 36, 39
row estimation
planner, 1409
row type, 124
constructor, 36
row-wise comparison, 213
rpad, 138
rtrim, 138
rule, 697
and views, 699
for DELETE, 705
for INSERT, 705
for SELECT, 699
compared with triggers, 717
for UPDATE, 705
SAVEPOINT, 1118
savpoints
defining, 1118
releasing, 1107
rolling back, 1116
scalar
(see expression)
schema, 53, 393
creating, 53
current, 55, 217
public, 54
removing, 54
SCO OpenServer
IPC configuration, 340
search path, 55
SPI_gettypeid, 829
SPI_getvalue, 826
SPI_is_cursor_plan, 811
spi_lastoid, 776
SPI_modifytuple, 838
SPI_palloc, 832
SPI_pfree, 835
SPI_pop, 801
SPI_prepare, 806
SPI_prepare_cursor, 808
SPI_push, 800
SPI_realloc, 834
SPI_returntuple, 837
SPI_saveplan, 823
SPI_scroll_cursor_fetch, 820
SPI_scroll_cursor_move, 821
split_part, 138
sql_inheritance configuration parameter, 382
sqrt, 134
ssh, 346
SSL, 345, 516
with libpq, 478, 485
ssl configuration parameter, 352
sslinfo, 1900
ssl_ciphers configuration parameter, 352
ssl_renegotiation_limit configuration parameter, 352
SSPI, 404
STABLE, 642
standard deviation, 209
population, 209
sample, 209
standard_conforming_strings configuration parameter, 382
standby server, 426
START TRANSACTION, 1146
statement_timeout configuration parameter, 377
statement_timestamp, 171
statistics, 209, 449
of the planner, 303, 420
STONITH, 426
string
(see character string)
strings
backslash quotes, 381
escape warning, 382
standard conforming, 382
strip, 263
strpos, 138
subquery, 12, 35, 77, 210
substring, 137, 148, 151, 152
sum, 12
superuser, 4, 389
superuser_reserved_connections configuration parameter, 351
synchronize_seqscans configuration parameter, 382
synchronous commit, 464
synchronous_commit configuration parameter, 359
syntax
SQL, 23
syslog_facility configuration parameter, 367
table expression, 72
table function, 78
tablefunc, 1901
tableoid, 48
tablespace, 396
default, 376
temporary, 376
tan, 136
target list, 698
Tcl, 772
tcp_keepalives_count configuration parameter, 352
tcp_keepalives_idle configuration parameter, 351
tcp_keepalives_interval configuration parameter, 351
template0, 394
template1, 394, 394
temp_buffers configuration parameter, 354
temp_tablespaces configuration parameter, 376
test, 469
test_parser, 1911
text, 93

text search, 251
  data types, 112
  functions and operators, 112
  indexes, 281

threads
  with libpq, 517

tid, 128

time, 96, 99
  constants, 101
  current, 179
  output format, 101
    (see also formatting)

  time span, 96
  time with time zone, 96, 99
  time without time zone, 96, 99
  time zone, 102, 378
    conversion, 178
    input abbreviations, 1427
  time zone data, 319
  time zone names, 378
  timelines, 426
  timeofday, 171
  timeout
    client authentication, 352
    deadlock, 380

  timestamp, 96, 99
  timestamp with time zone, 96, 99
  timestamp without time zone, 96, 99
  timezone configuration parameter, 378
  timezone_abbreviations configuration parameter, 378
  TOAST, 1400
    and user-defined types, 672
    per-column storage settings, 884
    versus large objects, 528

token, 23

to_ascii, 138

to_char, 164
  and locales, 411

to_date, 164

to_hex, 138

to_number, 164

to_timestamp, 164

to_tsquery, 257

to_tsvector, 256

trace_notify configuration parameter, 385
trace_sort configuration parameter, 385
track_activities configuration parameter, 373
track_count configuration parameter, 373

transact, 16

transaction ID

  wraparound, 421

transaction isolation, 288
transaction isolation level, 288, ??
  read committed, 289
  serializable, 290

transaction log
  (see WAL)

transaction_timestamp, 171

transform_null_equals configuration parameter, 383

translate, 138

  arguments for trigger functions, 690
  for updating a derived tsvector column, 265
    in C, 691
    in PL/pgSQL, 751
    in PL/Python, 795
    in PL/Tcl, 776
    compared with rules, 717

  trim, 137

Tru64 UNIX
  shared library, 655

true, 104

trunc, 134

TRUNCATE, 1147

trusted
  PL/Perl, 787

tsearch2, 1912

tsquery (data type), 113

tsvector (data type), 112

  tsvector concatenation, 262
  ts_debug, 278
  ts_headline, 261
  ts_lexize, 281
  ts_parse, 280
  ts_rank, 259
  ts_rank_cd, 259
  ts_rewrite, 264
  ts_stats, 266
  ts_token_type, 280

  txid_current, 223
  txid_current_snapshot, 223
  txid_snapshot_xip, 223
  txid_snapshot_xmax, 223
  txid_snapshot_xmin, 223
  txid_visible_in_snapshot, 223

type
  (see data type)
    polymorphic, 631

  type cast, 27, 34
UNION, 83
  determination of result type, 237
unique constraint, 44
Unix domain socket, 476
UnixWare
  IPC configuration, 340
  shared library, 655
unix_socket_directory configuration parameter, 351
unix_socket_group configuration parameter, 351
unix_socket_permissions configuration parameter, 351
UNLISTEN, 1149
unqualified name, 55
UPDATE, 13, 70, 1151
update_process_title configuration parameter, 373
updating, 70
upgrading, 314, 443
upper, 137
  and locales, 411
user, 388
  current, 217
UUID, 114, 319
uuid-ossp, 1914
vacuum, 419, 1155
vacuumdb, 1237
vacuumlo, 1915
vacuum_cost_delay configuration parameter, 357
vacuum_cost_limit configuration parameter, 357
vacuum_cost_page_dirty configuration parameter, 357
vacuum_cost_page_hit configuration parameter, 357
vacuum_cost_page_miss configuration parameter, 357
vacuum_freeze_min_age configuration parameter, 377
value expression, 30
VALUES, 85, 1158
  determination of result type, 237
varchar, 93
variance, 209
population, 209
sample, 209
version, 4, 218
  compatibility, 443
view, 15
  implementation through rules, 699
  updating, 710
void, 129
VOLATILE, 642
volatility
  functions, 642
WAL, 463
wal_buffers configuration parameter, 360
wal_debug configuration parameter, 386
wal_sync_method configuration parameter, 359
wal_writer_delay configuration parameter, 360
warm standby, 426
WHERE, 78
where to log, 365
WHILE
  in PL/pgSQL, 742
width, 184
width_bucket, 134
witness server, 426
work_mem configuration parameter, 354
xid, 128
xmax, 48
xmin, 48
XML, 114
XML export, 197
XML option, 115, 378
xml2, 1916
xmlagg, 196
xmlbinary configuration parameter, 377
xmlcomment, 193
xmlconcat, 193
xmlelement, 194
xmlforest, 195
xmloption configuration parameter, 378
xmlparse, 115
xmlpi, 196
xmlroot, 196
xmlserialize, 115
XPath, 197

Y

yacc, 314

Z

zero_damaged_pages configuration parameter, 386
zlib, 313, 320