PostgreSQL 7.3.2 User's Guide

The PostgreSQL Global Development Group

PostgreSQL 7.3.2 User's Guide

by The PostgreSQL Global Development Group

Legal Notice

Table of Contents

List of Tables

List of Examples

Preface

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. 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.

PostgreSQL is an open-source descendant of this original Berkeley code. It provides SQL92/SQL99 language support and other modern features.

POSTGRES pioneered many of the object-relational concepts now becoming available in some commercial databases. Traditional relational database management systems (RDBMS) support a data model consisting of a collection of named relations, containing attributes of a specific type. In current commercial systems, possible types include floating point numbers, integers, character strings, money, and dates. It is commonly recognized that this model is inadequate for future data-processing applications. The relational model successfully replaced previous models in part because of its "Spartan simplicity". However, this simplicity makes the implementation of certain applications very difficult. PostgreSQL offers substantial additional power by incorporating the following additional concepts in such a way that users can easily extend the system:

- inheritance
- data types
- functions

Other features provide additional power and flexibility:

- constraints
- triggers
- rules
- transactional integrity

These features put PostgreSQL into the category of databases referred to as *object-relational*. Note that this is distinct from those referred to as *object-oriented*, which in general are not as well suited to supporting traditional relational database languages. So, although PostgreSQL has some object-oriented features, it is firmly in the relational database world. In fact, some commercial databases have recently incorporated features pioneered by PostgreSQL.

1. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

2. A Short History of PostgreSQL

The object-relational database management system now known as PostgreSQL (and briefly called Postgres95) is derived from the POSTGRES package written at the University of California at Berkeley. With over a decade of development behind it, PostgreSQL is the most advanced open-source database available anywhere, offering multiversion concurrency control, supporting almost all SQL constructs (including subselects, transactions, and user-defined types and functions), and having a wide range of language bindings available (including C, C++, Java, Perl, Tcl, and Python).

2.1. The Berkeley POSTGRES Project

Implementation of the POSTGRES DBMS 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 rewrite 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 Informix[2], which is now owned by IBM[3].) picked up the code and commercialized it. POSTGRES became the primary data manager for the Sequoia 2000[4] scientific computing project in late 1992.

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.

2.2. Postgres95

In 1994, Andrew Yu and Jolly Chen added a SQL language interpreter to POSTGRES. 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

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

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. Aggregates were re-implemented. Support for the GROUP BY query clause was also added. The `libpq` interface remained available for C programs.
- In addition to the monitor program, a new program (psql) was provided for interactive SQL queries using GNU Readline.
- 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 backend.
- 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.

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

Major enhancements in PostgreSQL include:

- Table-level locking has been replaced by multiversion concurrency control, which allows readers to continue reading consistent data during writer activity and enables hot backups from pg_dump while the database stays available for queries.
- Important backend features, including subselects, defaults, constraints, and triggers, have been implemented.
- Additional SQL92-compliant language features have been added, including primary keys, quoted identifiers, literal string type coercion, type casting, and binary and hexadecimal integer input.
- Built-in types have been improved, including new wide-range date/time types and additional geometric type support.
- Overall backend code speed has been increased by approximately 20-40%, and backend start-up time has decreased by 80% since version 6.0 was released.

3. What's In This Book

This book 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 data commands. The rest of the book treats several aspects that are important for tuning a database for optimal performance.

The information in this book 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 book is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should look into the *PostgreSQL Reference Manual*.

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

This book covers PostgreSQL 7.3.2 only. For information on other versions, please read the documentation that accompanies that release.

4. Overview of Documentation Resources

The PostgreSQL documentation is organized into several books:

PostgreSQL Tutorial

An informal introduction for new users.

PostgreSQL User's Guide

Documents the SQL query language environment, including data types and functions, as well as user-level performance tuning. Every PostgreSQL user should read this.

PostgreSQL Administrator's Guide

Installation and server management information. Everyone who runs a PostgreSQL server, either for personal use or for other users, needs to read this.

PostgreSQL Programmer's Guide

Advanced information for application programmers. Topics include type and function extensibility, library interfaces, and application design issues.

PostgreSQL Reference Manual

Reference pages for SQL command syntax, and client and server programs. This book is auxiliary to the User's, Administrator's, and Programmer's Guides.

PostgreSQL Developer's Guide

Information for PostgreSQL developers. This is intended for those who are contributing to the PostgreSQL project; application development information appears in the *Programmer's Guide*.

In addition to this manual set, there are other resources to help you with PostgreSQL installation and use:

man pages

The *Reference Manual*'s pages in the traditional Unix man format. There is no difference in content.

FAQs

Frequently Asked Questions (FAQ) lists document both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

Web Site

The PostgreSQL web site[5] carries details on the latest release, upcoming features, 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 User's Lounge[6] section of the PostgreSQL web site for details.

Yourself!

PostgreSQL is an open-source effort. 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. 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.

Even those without a lot of experience can provide corrections and minor changes in the documentation, and that is a good way to start. The <`pgsql-docs@postgresql.org`> mailing list is the place to get going.

5. Terminology and Notation

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 documentation set does not have fixed presumptions about system administration procedures.

We use `/usr/local/pgsql/` as the root directory of the installation and `/usr/local/pgsql/data` as the directory with the database files. These directories may vary on your site, details can be derived in the *Administrator's Guide*.

5. http://www.postgresql.org
6. http://www.postgresql.org/users-lounge/

In a command synopsis, brackets ([and]) indicate an optional phrase or keyword. Anything in braces ({ and }) and containing vertical bars (|) indicates that you must choose one alternative.

Examples will show commands executed from various accounts and programs. Commands executed from a Unix shell may be preceded with a dollar sign ("$"). Commands executed from particular user accounts such as root or postgres are specially flagged and explained. SQL commands may be preceded with "=>" or will have no leading prompt, depending on the context.

> **Note:** The notation for flagging commands is not universally consistent throughout the documentation set. Please report problems to the documentation mailing list <pgsql-docs@postgresql.org>.

6. 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 it 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.

6.1. 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 the 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.

Before you continue, check on the TODO list and in the FAQ to see if your bug is already known. If you cannot decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

6.2. 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 query-language 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 "mid-size 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:** 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 concerned environment variables or configuration files that you changed from the default. Again, be exact. 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 `postmaster --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. You can also look into the `README` file in the source directory or at the name of your distribution file or package name. If you run a prepackaged version, such as RPMs, say so, including any subversion the package may have. If you are talking about a CVS snapshot, mention that, including its date and time.

  If your version is older than 7.3.2 we will almost certainly tell you to upgrade. There are tons of bug fixes in each new release, that is why we make new releases.

- Platform information. This includes the kernel name and version, C library, processor, memory information. 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 compilers, make, etc. 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.

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 choose non-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 “postmaster” process; please don’t say “the postmaster crashed” when you mean a single backend 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.

6.3. 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 http://www.postgresql.org/. Entering a bug report this way causes it to be mailed to the <pgsql-bugs@postgresql.org> mailing list.

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 pgsql-hackers, if the problem needs more review.

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-ports@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.

Chapter 1. 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.

1.1. 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 the *PostgreSQL Reference Manual*.

1.1.1. 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 B.

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, digits (`0-9`), or underscores, although the SQL standard will not define a key word that contains digits or starts or ends with an underscore.

The system uses no more than `NAMEDATALEN`-1 characters 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 (but at the time PostgreSQL is built, `NAMEDATALEN` can be changed in `src/include/postgres_ext.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 other than a double quote itself. 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. [1]

1.1.2. 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. The implicit constants are described below; explicit constants are discussed afterwards.

1.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes (“’”), e.g., `'This is a string'`. SQL allows single quotes to be embedded in strings by typing two adjacent single quotes

1. 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.

(e.g., `'Dianne"s horse'`). In PostgreSQL single quotes may alternatively be escaped with a backslash ("\", e.g., `'Dianne\'s horse'`).

C-style backslash escapes are also available: `\b` is a backspace, `\f` is a form feed, `\n` is a newline, `\r` is a carriage return, `\t` is a tab, and `\`*`xxx`*, where *`xxx`* is an octal number, is the character with the corresponding ASCII code. Any other character following a backslash is taken literally. Thus, to include a backslash in a string constant, type two backslashes.

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

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 in one constant. For example:

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

is equivalent to

```
SELECT 'foobar';
```

but

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

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

1.1.2.2. Bit-String Constants

Bit-string constants look like 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.

1.1.2.3. Numeric Constants

Numeric constants are accepted in these general forms:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

where *`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 may not 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 `integer` if its value fits in type `integer` (32 bits); otherwise it is presumed to be type `bigint` if its value fits in type `bigint` (64 bits); otherwise it is taken to be type `numeric`. Constants that contain decimal points and/or exponents are always initially presumed to be type `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 `real` (`float4`) by writing

```
REAL '1.23'  -- string style
1.23::REAL   -- PostgreSQL (historical) style
```

1.1.2.4. Constants of Other Types

A constant of an *arbitrary* type can be entered using any one of the following notations:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

The string's text is passed to the input conversion routine for the type called *`type`*. The result is a constant of the indicated type. The explicit type cast may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is passed as an argument to a non-overloaded function), in which case it is automatically coerced.

It is also possible to specify a type coercion using a function-like syntax:

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

but not all type names may be used in this way; see Section 1.2.6 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 1.2.6. But the form *`type`*`'string'` can only be used to specify the type of a literal constant. Another restriction on *`type`*`'string'` is that it does not work for array types; use `::` or `CAST()` to specify the type of an array constant.

1.1.2.5. Array constants

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. (For all built-in types, this is the comma character ",".) 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.

Individual array elements can be placed between double-quote marks (") to avoid ambiguity problems with respect to whitespace. Without quote marks, the array-value parser will skip leading whitespace.

(Array constants are actually only a special case of the generic type constants discussed in the previous section. The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

1.1.3. Operators

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

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

There are a few restrictions on operator names, however:

- $ (dollar) cannot be a single-character operator, although it can be part of a multiple-character operator name.
- -- 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:

  ~ ! @ # % ^ & | ` ? $

  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.

1.1.4. 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 the positional parameters in the body of a function definition. In other contexts the dollar sign may be part of an operator name.
- 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 5.12 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 5.12.) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
- The asterisk (`*`) has a special meaning when used in the `SELECT` command or with the `COUNT` aggregate function.
- The period (`.`) is used in floating-point constants, and to separate schema, table, and column names.

1.1.5. Comments

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

```
-- This is a standard SQL92 comment
```

Alternatively, C-style block comments can be used:

```
/* 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 SQL99 but unlike C, so that one can comment out larger blocks of code that may contain existing block comments.

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

1.1.6. Lexical Precedence

Table 1-1 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 may 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

```
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 1-1. Operator Precedence (decreasing)

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

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 1-1 for "any other" operator. This is true no matter which specific operator name appears inside `OPERATOR()`.

1.2. 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; see Section 1.1.2.
- A column reference.
- A positional parameter reference, in the body of a function declaration.
- An operator invocation.
- A function call.
- An aggregate expression.
- A type cast.
- A scalar subquery.
- 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 6. An example is the `IS NULL` clause.

We have already discussed constants in Section 1.1.2. The following sections discuss the remaining options.

1.2.1. Column References

A column can be referenced in the form

```
correlation.columnname
```

or

```
correlation.columnname[subscript]
```

(Here, the brackets [ ] are meant to appear literally.)

correlation is the name of a table (possibly qualified), or an alias for a table defined by means of a FROM clause, or 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 may be omitted if the column name is unique across all the tables being used in the current query. (See also Chapter 4.)

If *column* is of an array type, then the optional *subscript* selects a specific element or elements in the array. If no subscript is provided, then the whole array is selected. (See Section 5.12 for more about arrays.)

1.2.2. Positional Parameters

A positional parameter reference is used to indicate a parameter that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. The form of a parameter reference is:

```
$number
```

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

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

Here the $1 will be replaced by the first function argument when the function is invoked.

1.2.3. 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 1.1.3, or is one of the keywords AND, OR, and NOT, or is a qualified operator name

```
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 6 describes the built-in operators.

1.2.4. 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 6. Other functions may be added by the user.

1.2.5. 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 a qualified 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 yields a non-null value. (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 `ALL` is the default. The third form invokes the aggregate for all distinct non-null values of the expression 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 `count()` aggregate function.

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

The predefined aggregate functions are described in Section 6.14. Other aggregate functions may be added by the user.

1.2.6. Type Casts

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

```
CAST ( expression AS type )
expression::type
```

The `CAST` syntax conforms to SQL; the syntax with `::` 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 function is available. Notice that this is subtly different from the use of casts with constants, as shown in Section 1.1.2.4. 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 may 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:

```
typename ( expression )
```

However, this only works for types whose names are also valid as function names. For example, `double precision` can't be used this way, but the equivalent `float8` can. Also, the names `interval`, `time`, and `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. (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, but this is not something that a portable application should rely on.)

1.2.7. Scalar Subqueries

A scalar subquery is an ordinary `SELECT` query in parentheses that returns exactly one row with one column. (See Chapter 4 for information about writing queries.) The `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 6.15.

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

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

1.2.8. Expression Evaluation

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 may 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 6.12) may 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.

Chapter 2. 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 views, functions, and triggers. Detailed information on these topics is found in the *PostgreSQL Programmer's Guide*.

2.1. 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 4. 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 5. 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 1.1.1. 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 (
    product_no integer,
    name text,
    price numeric
);
```

(The `numeric` type can store fractional components, as would be typical of monetary amounts.)

> **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 don't need a table anymore, 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 the error messages.

If you need to modify a table that already exists look into Section 2.6 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 3 and read the rest of this chapter later.

2.2. System Columns

Every table has several *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.

`oid`

The object identifier (object ID) of a row. This is a serial number that is automatically added by PostgreSQL to all table rows (unless the table was created `WITHOUT OIDS`, in which case this column is not present). This column is of type `oid` (same name as the column); see Section 5.10 for more information about the type.

`tableoid`

The OID of the table containing this row. This attribute is particularly handy for queries that select from inheritance hierarchies, since without it, it's difficult to tell which individual table a row came from. The `tableoid` can be joined against the `oid` column of `pg_class` to obtain the table name.

`xmin`

The identity (transaction ID) of the inserting transaction for this tuple. (Note: In this context, a tuple is an individual state of a row; each update of a row creates a new tuple for the same logical row.)

`cmin`

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

`xmax`

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

`cmax`

The command identifier within the deleting transaction, or zero.

`ctid`

The physical location of the tuple within its table. Note that although the `ctid` can be used to locate the tuple very quickly, a row's `ctid` will change each time it is updated or moved by `VACUUM FULL`. Therefore `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.

2.3. 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, the 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 knowing what this value is. (Details about data manipulation commands are in Chapter 3.)

If no default value is declared explicitly, the null value is the default value. This usually makes sense because a null value can be thought 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 PRIMARY KEY,
    name text,
    price numeric DEFAULT 9.99
);
```

The default value may be a scalar expression, which will be evaluated whenever the default value is inserted (*not* when the table is created).

2.4. 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 prob-

ably only accept positive values. But there is no 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 only be 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.

2.4.1. 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 an arbitrary expression. For instance, to require positive product prices, you could use:

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

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 `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.

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 the column definitions. Column constraints can also be written as table constraints, while the reverse is not necessarily possible. 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.

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 one 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 should be used.

2.4.2. 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 that way.

Of course, a column can have more than one constraint. Just write the constraints after one 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 affect 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 defines the default behavior that the column may be null. The `NULL` constraint is not defined in the SQL standard and should not be used in portable applications. (It was only added to PostgreSQL to be compatible with 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.

> **Tip:** In most database designs the majority of columns should be marked not null.

2.4.3. 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

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

when written as a column constraint, and

```
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:

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

It is also possible to assign names to unique constraints:

```
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 (at least) two rows in the table where the values of each of the corresponding columns that are part of the constraint are equal. However, null values are not considered equal in this consideration. That means, in the presence of a multicolumn unique constraint it is possible to store an unlimited number of 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 may not follow this rule. So be careful when developing applications that are intended to be portable.

2.4.4. 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:

```
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:

```
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, in fact, 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 (while it can have many unique and not-null constraints). 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.

2.4.5. 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:

```
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:

```
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

```
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 referenced column.

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:

```
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 needs to match the number and type of the referenced columns.

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:

```
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)
);
```

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

We know that the foreign keys disallow creation of orders that don't relate to any products. But what if a product is removed after an order is created that references it? SQL allows you to specify 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` can also be written as `NO ACTION` and it's also the default if you don't specify anything. There are two other options for what should happen with the foreign key columns when a primary key is deleted: `SET NULL` and `SET DEFAULT`. 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 deletion of the primary key will fail.

Analogous to `ON DELETE` there is also `ON UPDATE` which is invoked when a primary key is changed (updated). The possible actions are the same.

More information about updating and deleting data is in Chapter 3.

Finally, we should mention that a foreign key must reference columns that are either 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 `CREATE TABLE` entry in the *PostgreSQL Reference Manual*.

2.5. Inheritance

Let's create two tables. The capitals table contains state capitals which are also cities. Naturally, the capitals table should inherit from cities.

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, a row of capitals *inherits* all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is `text`, a native PostgreSQL type for variable length ASCII strings. The type of the attribute population is `float`, a native PostgreSQL type for double precision floating-point numbers. State capitals have an extra attribute, 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 descendants.

Note: The inheritance hierarchy is actually a directed acyclic graph.

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

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

which returns:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

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

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

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

Here the "ONLY" before cities indicates that the query should be run over only cities 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.

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

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

which returns:

```
 tableoid |   name    | altitude
----------+-----------+----------
```

```
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

(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:

```
 relname  |   name    | altitude
----------+-----------+----------
 cities   | Las Vegas |     2174
 cities   | Mariposa  |     1953
 capitals | Madison   |      845
```

Deprecated: In previous versions of PostgreSQL, the default was not to get access to child tables. This was found to be error prone and is also in violation of the SQL standard. Under the old syntax, to get the sub-tables you append `*` to the table name. For example

```
SELECT * from cities*;
```

You can still explicitly specify scanning child tables by appending `*`, as well as explicitly specify not scanning child tables by writing "ONLY". But beginning in version 7.1, the default behavior for an undecorated table name is to scan its child tables too, whereas before the default was not to do so. To get the old default behavior, set the configuration option `SQL_Inheritance` to off, e.g.,

```
SET SQL_Inheritance TO OFF;
```

or add a line in your `postgresql.conf` file.

A 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. Thus, in the above example, specifying that another table's column `REFERENCES cities(name)` would allow the other table to contain city names but not capital names. This deficiency will probably be fixed in some future release.

2.6. Modifying Tables

When you create a table and you realize that you made a mistake, or the requirements of the application changed, 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 on existing tables.

You can

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

All these actions are performed using the `ALTER TABLE` command.

2.6.1. Adding a Column

To add a column, use this command:

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

The new column will initially be filled with null values in the existing rows of the table.

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

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

A new column cannot have a not-null constraint since the column initially has to contain null values. But you can add a not-null constraint later. Also, you cannot define a default value on a new column. According to the SQL standard, this would have to fill the new columns in the existing rows with the default value, which is not implemented yet. But you can adjust the column default later on.

2.6.2. Removing a Column

To remove a column, use this command:

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

2.6.3. 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;
```

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

2.6.4. 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;
```

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.)

2.6.5. Changing the Default

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

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

To remove any default value, use

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

This is equivalent to setting the default to null, at least in PostgreSQL. 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.

2.6.6. Renaming a Column

To rename a column:

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

2.6.7. Renaming a Table

To rename a table:

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

2.7. 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. (There are also users that have the superuser privilege. Those users can always access any object.)

> **Note:** To change the owner of a table, index, sequence, or view, use the `ALTER TABLE` command.

There are several different privileges: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `RULE`, `REFERENCES`, `TRIGGER`, `CREATE`, `TEMPORARY`, `EXECUTE`, `USAGE`, and `ALL PRIVILEGES`. 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.

To assign privileges, the `GRANT` command is used. So, 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;
```

The user executing this command must be the owner of the table. To grant a privilege to a group, use

```
GRANT SELECT ON accounts TO GROUP staff;
```

The special "user" name `PUBLIC` can be used to grant a privilege to every user on the system. Writing `ALL` in place of a specific privilege specifies that all privileges will be granted.

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

```
REVOKE ALL ON accounts FROM PUBLIC;
```

The special privileges of the table 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 table owner can choose to revoke his own ordinary privileges, for example to make a table read-only for himself as well as others.

2.8. Schemas

A PostgreSQL database cluster (installation) 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` may contain tables named `mytable`. Unlike databases, schemas are not rigidly separated: a user may 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.

2.8.1. Creating a Schema

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

```
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:

```
schema.table
```

Actually, the even more general syntax

```
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

```
CREATE TABLE myschema.mytable (
 ...
);
```

This works anywhere a table name is expected, including the table modification commands and the data access commands discussed in the following chapters.

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

```
DROP SCHEMA myschema;
```

To drop a schema including all contained objects, use

```
DROP SCHEMA myschema CASCADE;
```

See Section 2.10 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:

```
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 2.8.6 for how this can be useful.

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

2.8.2. 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:

```
CREATE TABLE products ( ... );
```

and

```
CREATE TABLE public.products ( ... );
```

2.8.3. 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. Since no such schema exists yet, this 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 6.13 for other ways to access 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.

2.8.4. Schemas and Privileges

By default, users cannot see the 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 may 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 the `CREATE` privilege on the schema `public`. This allows all users that manage to connect to a given database to create objects there. If you do not want to allow that, you can revoke that privilege:

```
REVOKE CREATE ON 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 reserved word, hence the different capitalization; recall the guidelines from Section 1.1.1.)

2.8.5. 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 may 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 may 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 catalog named the same as your table. (With the default search path, an unqualified reference to your table name would be resolved as the system catalog instead.) System catalogs 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.

2.8.6. 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.
- 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 path, as they choose.

2.8.7. Portability

In the SQL standard, the notion of objects in the same schema being owned by different users does not exist. Moreover, some implementations don't 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.

2.9. 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.

- Views
- Functions, operators, data types, domains
- Triggers and rewrite rules

2.10. 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 2.4.5, with the orders table depending on it, would result in an error message such as this:

```
DROP TABLE products;
NOTICE:  constraint $1 on table orders depends on table products
ERROR:  Cannot drop table products because other objects depend on it
        Use DROP ... CASCADE to drop the dependent objects too
```

The error message contains a useful hint: If you don't 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 restrict 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 implements it that way, 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.

Chapter 3. 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.

3.1. 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 2:

```
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;
```

Tip: To do "bulk loads", that is, inserting a lot of data, take a look at the `COPY` command (see *PostgreSQL Reference Manual*). It is not as flexible as the `INSERT` command, but more efficient.

3.2. 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 2 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 may 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 may 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 also refer to the old value. 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 condition after the `WHERE` 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 condition does not have to be an equality test. Many other operators are available (see Chapter 6). But the expression needs to evaluate to a Boolean result.

You can also 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;
```

3.3. 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 discussed 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 4. 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.

4.1. 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.

The simplest 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 retrieve individual rows and columns.) 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 4.3 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();
```

4.2. 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 transformations produce a virtual table that provides the rows that are passed to the select list to compute the output rows of the query.

4.2.1. The FROM Clause

The FROM clause derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_reference [, table_reference [, ...]]
```

A table reference may be a table name (possibly schema-qualified), or a derived table such as a subquery, 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 may 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 supertable of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its subtable successors, unless the keyword 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.

4.2.1.1. 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

```
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. A cross join is equivalent to an INNER JOIN ON TRUE.

Tip: FROM *T1* CROSS JOIN *T2* is equivalent to FROM *T1*, *T2*.

Qualified joins

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join col-
umn 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.

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 AND t1.b = t2.b AND t1.c = t2.c)` with the exception that if `ON` is used there will be two columns `a`, `b`, 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`* may be joined tables. Parentheses may 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`

```
 num | name
-----+------
```

```
   1 | a
   2 | b
   3 | c
```

and `t2`

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

then we get the following results for the various joins:

```
=> 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)

=> 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)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> 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)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 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:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

4.2.1.2. 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 further processing. 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.n
```

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 SQL syntax. What will actually happen (this is a PostgreSQL extension to the standard) is that an implicit table reference is added to the FROM clause, so the query is processed as if it were written as

```
SELECT * FROM my_table AS m, my_table AS my_table WHERE my_table.a > 5;
```

which 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.,

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
```

Additionally, an alias is required if the table reference is a subquery (see Section 4.2.1.3).

Parentheses are used to resolve ambiguities. The following statement will assign the alias b to the result of the join, unlike the previous example:

```
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

Another form of table aliasing also gives temporary names to the columns of the table:

```
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,

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

is valid SQL, but

```
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.

4.2.1.3. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and *must* be assigned a table alias name. (See Section 4.2.1.2.) For example:

```
FROM (SELECT * FROM table1) AS alias_name
```

This example is equivalent to `FROM table1 AS alias_name`. More interesting cases, which can't be reduced to a plain join, arise when the subquery involves grouping or aggregation.

4.2.2. The WHERE Clause

The syntax of the `WHERE` clause is

```
WHERE search_condition
```

where *`search_condition`* is any value expression as defined in Section 1.2 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 in the table generated in the `FROM` clause; this is not required, but otherwise the `WHERE` clause will be fairly useless.

> **Note:** Before the implementation of the `JOIN` syntax, it was necessary to put the join condition of an inner join in the `WHERE` 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 products. 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 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. Qualifying the column name adds clarity even when it is not needed. This shows how the column naming scope of an outer query extends into its inner queries.

4.2.3. The GROUP BY and HAVING Clauses

After passing the `WHERE` filter, the derived input table may be subject to grouping, using the `GROUP BY` clause, and elimination of group rows using the `HAVING` clause.

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference]...
```

The `GROUP BY` clause is used to group together 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 purpose is to reduce each group 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:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

In the second query, we could not have written `SELECT * FROM test1 GROUP BY x`, because there is no single value for the column `y` that could be associated with each group. The grouped-by columns can be referenced in the select list since they have a known constant value per 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:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Here `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 6.14.

> **Tip:** Grouping without aggregate expressions effectively calculates the set of distinct values in a column. This can also be achieved using the `DISTINCT` clause (see Section 4.3.3).

Here is another example: `sum(sales)` on a table grouped by product code gives the total sales for each product, not the total sales on all products.

```
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 may 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 group of sales of a product. For each product, a summary row is returned 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:

```
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:

```
=> 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:

```
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, 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 everywhere.

4.3. 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.

4.3.1. 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 1.2). 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 4.2.1.2. 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.b, tbl1.c FROM ...
```

(See also Section 4.2.2.)

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 retrieved 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.

4.3.2. 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 via 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 4.2.1.2). 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.

4.3.3. DISTINCT

After the select list has been processed, the result table may optionally be subject to the elimination of duplicates. The `DISTINCT` key word is written directly after the `SELECT` to enable this:

```
SELECT DISTINCT select_list ...
```

(Instead of `DISTINCT` the word `ALL` can be used to select 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:

```
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 subselects in FROM the construct can be avoided, but it is often the most convenient alternative.

4.4. Combining Queries

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

```
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

```
query1 UNION query2 UNION query3
```

which really says

```
(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 all duplicate rows, in the sense of DISTINCT, unless UNION ALL is used.

INTERSECT returns all rows that are both in the result of *query1* and in the result of *query2*. Duplicate rows are eliminated unless INTERSECT ALL is used.

EXCEPT returns all rows that are in the result of *query1* but not in the result of *query2*. (This is sometimes called the *difference* between two queries.) Again, duplicates are eliminated unless 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 both return the same number of columns, and that the corresponding columns have compatible data types, as described in Section 7.5.

4.5. 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 random 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 ORDER BY clause specifies the sort order:

```
SELECT select_list
    FROM table_expression
```

```
ORDER BY column1 [ASC | DESC] [, column2 [ASC | DESC] ...]
```

column1, etc., refer to select list columns. These can be either the output name of a column (see Section 4.3.2) or the number of a column. Some examples:

```
SELECT a, b FROM table1 ORDER BY a;
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, sum(b) FROM table1 GROUP BY a ORDER BY 1;
```

As an extension to the SQL standard, PostgreSQL also allows ordering by arbitrary expressions:

```
SELECT a, b FROM table1 ORDER BY a + b;
```

References to column names in the FROM clause that are renamed in the select list are also allowed:

```
SELECT a AS b FROM table1 ORDER BY a;
```

But these extensions do not work in queries involving UNION, INTERSECT, or EXCEPT, and are not portable to other SQL databases.

Each column specification may be followed by an optional ASC or DESC to set the sort direction to ascending or descending. ASC order is the default. Ascending order puts smaller values first, where "smaller" is defined in terms of the < operator. Similarly, descending order is determined with the > operator.

If more than one sort column is specified, the later entries are used to sort rows that are equal under the order imposed by the earlier sort columns.

4.6. 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
    [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 to the client. 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 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 may 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.

Chapter 5. Data Types

PostgreSQL has a rich set of native data types available to users. Users may add new types to PostgreSQL using the `CREATE TYPE` command.

Table 5-1 shows all general-purpose data types included in the standard distribution. 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 5-1. Data Types

Type Name	Aliases	Description
`bigint`	`int8`	signed eight-byte integer
`bigserial`	`serial8`	autoincrementing eight-byte integer
`bit`		fixed-length bit string
`bit varying(`*n*`)`	`varbit(`*n*`)`	variable-length bit string
`boolean`	`bool`	logical Boolean (true/false)
`box`		rectangular box in 2D plane
`bytea`		binary data
`character varying(`*n*`)`	`varchar(`*n*`)`	variable-length character string
`character(`*n*`)`	`char(`*n*`)`	fixed-length character string
`cidr`		IP network address
`circle`		circle in 2D plane
`date`		calendar date (year, month, day)
`double precision`	`float8`	double precision floating-point number
`inet`		IP host address
`integer`	`int`, `int4`	signed four-byte integer
`interval(`*p*`)`		general-use time span
`line`		infinite line in 2D plane (not implemented)
`lseg`		line segment in 2D plane
`macaddr`		MAC address
`money`		currency amount
`numeric [ (`*p*`, `*s*`) ]`	`decimal [ (`*p*`, `*s*`) ]`	exact numeric with selectable precision
`path`		open and closed geometric path in 2D plane
`point`		geometric point in 2D plane
`polygon`		closed geometric path in 2D plane

Type Name	Aliases	Description
real	float4	single precision floating-point number
smallint	int2	signed two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(*p*)] [without time zone]		time of day
time [(*p*)] with time zone	timetz	time of day, including time zone
timestamp [(*p*)] without time zone	timestamp	date and time
timestamp [(*p*)] [with time zone]	timestamptz	date and time, including time zone

Compatibility: The following types (or spellings thereof) are specified by SQL: bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time, timestamp (both with or without time zone).

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 open and closed paths, or have several possibilities for formats, such as the date and time types. Most of the input and output functions corresponding to the base types (e.g., integers and floating-point numbers) do some error-checking. Some of the input and output functions are not invertible. That is, the result of an output function may lose precision when compared to the original input.

Some of the operators and functions (e.g., addition and multiplication) do not perform run-time error-checking in the interests of improving execution speed. On some systems, for example, the numeric operators for some data types may silently underflow or overflow.

5.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and fixed-precision decimals. Table 5-2 lists the available types.

Table 5-2. Numeric Types

Type name	Storage size	Description	Range
smallint	2 bytes	small range fixed-precision	-32768 to +32767
integer	4 bytes	usual choice for fixed-precision	-2147483648 to +2147483647

Type name	Storage size	Description	Range
bigint	8 bytes	large range fixed-precision	-9223372036854775808 to 9223372036854775807
decimal	variable	user-specified precision, exact	no limit
numeric	variable	user-specified precision, exact	no limit
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double precision	8 bytes	variable-precision, inexact	15 decimal digits precision
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

The syntax of constants for the numeric types is described in Section 1.1.2. The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Chapter 6 for more information. The following sections describe the types in detail.

5.1.1. The Integer Types

The types `smallint`, `integer`, `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 may 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`) and `smallint`. The type `bigint`, and the type names `int2`, `int4`, and `int8` are extensions, which are shared with various other SQL database systems.

> **Note:** If you have a column of type `smallint` or `bigint` with an index, you may encounter problems getting the system to use that index. For instance, a clause of the form
>
> ```
> ... WHERE smallint_column = 42
> ```
>
> will not use an index, because the system assigns type `integer` to the constant 42, and PostgreSQL currently cannot use an index when two different data types are involved. A workaround is to single-quote the constant, thus:
>
> ```
> ... WHERE smallint_column = '42'
> ```
>
> This will cause the system to delay type resolution and will assign the right type to the constant.

5.1.2. Arbitrary Precision Numbers

The type `numeric` can store numbers with up to 1,000 digits of precision and perform calculations exactly. It is especially recommended for storing monetary amounts and other quantities where exactness is required. However, the `numeric` type is very slow compared 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 precision and the scale of the numeric type can be configured. To declare a column of type `numeric` use the syntax

```
NUMERIC(precision, scale)
```

The precision must be positive, the scale zero or positive. Alternatively,

```
NUMERIC(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 precision or scale of a value is greater than the declared precision or scale of a column, the system will attempt to round the value. If the value cannot be rounded so as to satisfy the declared limits, an error is raised.

The types `decimal` and `numeric` are equivalent. Both types are part of the SQL standard.

5.1.3. Floating-Point Types

The data types `real` and `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 may 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 `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 may or may not work as expected.

Normally, 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 normally has a range of around -1E+308 to +1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may 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.

5.1.4. The Serial Types

The `serial` data type is not a true type, but merely a notational convenience for setting up identifier columns (similar to the `AUTO_INCREMENT` property supported by some other databases). In the current implementation, specifying

```
CREATE TABLE tablename (
    colname SERIAL
);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
    colname integer DEFAULT nextval('tablename_colname_seq') NOT NULL
);
```

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.

To use a `serial` column to insert the next value of the sequence into the table, specify that the `serial` column should be assigned the default value. This can be done either be excluding from the column from the list of columns in the `INSERT` statement, or through the use of the `DEFAULT` keyword.

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 by a `serial` type is automatically dropped when the owning column is dropped, and cannot be dropped otherwise. (This was not true in PostgreSQL releases before 7.3. Note that this automatic drop linkage will not occur for a sequence created by reloading a dump from a pre-7.3 database; the dump file does not contain the information needed to establish the dependency link.) Furthermore, this dependency between sequence and column is made only for the `serial` column itself; if any other

columns reference the sequence (perhaps by manually calling the `nextval()`) function), they may be broken if the sequence is removed. Using `serial` columns in fashion is considered bad form.

> **Note:** Prior to PostgreSQL 7.3, `serial` implied `UNIQUE`. This is no longer automatic. If you wish a serial column to be `UNIQUE` or a `PRIMARY KEY` it must now be specified, just as with any other data type.

5.2. Monetary Type

> **Note:** The `money` type is deprecated. Use `numeric` or `decimal` instead, in combination with the `to_char` function. The money type may become a locale-aware layer over the `numeric` type in a future release.

The `money` type stores a currency amount with fixed decimal point representation; see Table 5-3. The output format is locale-specific.

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 in the latter form.

Table 5-3. Monetary Types

Type Name	Storage	Description	Range
money	4 bytes	currency amount	-21474836.48 to +21474836.47

5.3. Character Types

Table 5-4. Character Types

Type name	Description
`character varying(`*n*`)`, `varchar(`*n*`)`	variable-length with limit
`character(`*n*`)`, `char(`*n*`)`	fixed-length, blank padded
`text`	variable unlimited length

Table 5-4 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.

> **Note:** 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.)

> **Note:** Prior to PostgreSQL 7.2, strings that were too long were always truncated without raising an error, in either explicit or implicit casting contexts.

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 supports the more general `text` type, which stores strings of any length. Unlike `character varying`, `text` does not require an explicit declared upper limit on the size of the string. Although the type `text` is not in the SQL standard, many other RDBMS packages have it as well.

The storage requirement for data of these types is 4 bytes plus the actual string, and in case of `character` plus the padding. Long strings are compressed by the system automatically, so the physical requirement on disk may be less. Long values are also stored in background tables so they don't interfere with rapid access to the 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 the increased storage size when using the blank-padded type.

Refer to Section 1.1.2.1 for information about the syntax of string literals, and to Chapter 6 for information about available operators and functions.

Example 5-1. Using the character types

```
    CREATE TABLE test1 (a character(4));
    INSERT INTO test1 VALUES ('ok');
    SELECT a, char_length(a) FROM test1; -- ❶
      a   | char_length
------+-------------
 ok   |           4

    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 l |           5
```

❶ The `char_length` function is discussed in Section 6.4.

There are two other fixed-length character types in PostgreSQL, shown in Table 5-5. The `name` type exists *only* for storage of internal catalog names 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 may 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.

Table 5-5. Specialty Character Types

Type Name	Storage	Description
`"char"`	1 byte	single character internal type
`name`	64 bytes	sixty-three character internal type

5.4. Binary Strings

The `bytea` data type allows storage of binary strings; see Table 5-6.

Table 5-6. Binary String Types

Type Name	Storage	Description
`bytea`	4 bytes plus the actual binary string	Variable (not specifically limited) length binary string

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from characters strings by two characteristics: First, binary strings specifically allow storing octets of zero value and other "non-printable" octets. Second, operations on binary strings process the actual bytes, whereas the encoding and processing of character strings depends on locale settings.

When entering `bytea` values, octets of certain values *must* be escaped (but all octet values *may* 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. Some octet values have alternate escape sequences, as shown in Table 5-7.

Table 5-7. `bytea` Literal Escaped Octets

Decimal Octet Value	Description	Input Escaped Representation	Example	Printed Result
0	zero octet	`'\\000'`	`SELECT '\\000'::bytea;`	`\000`
39	single quote	`'\''` or `'\\047'`	`SELECT '\''::bytea;`	`'`
92	backslash	`'\\\\'` or `'\\134'`	`SELECT '\\\\'::bytea;`	`\\`

Note that the result in each of the examples in Table 5-7 was exactly one octet in length, even though the output representation of the zero octet and backslash are more than one character. `Bytea` output octets are also escaped. In general, each "non-printable" octet decimal value 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 alternate output representation. Details are in Table 5-8.

Table 5-8. `bytea` Output Escaped Octets

Decimal Octet Value	Description	Output Escaped Representation	Example	Printed Result
92	backslash	`\\`	`SELECT '\\134'::bytea;`	`\\`
0 to 31 and 127 to 255	"non-printable" octets	`\###` (octal value)	`SELECT '\\001'::bytea;`	`\001`
32 to 126	"printable" octets	ASCII representation	`SELECT '\\176'::bytea;`	~

To use the `bytea` escaped octet notation, string literals (input strings) must contain two backslashes because they must pass through two parsers in the PostgreSQL server. The first backslash is interpreted as an escape character by the string-literal parser, and therefore is consumed, leaving the characters that follow. The remaining backslash is recognized by the `bytea` input function as the prefix of a three digit octal value. For example, a string literal passed to the backend as `'\\001'` becomes `'\001'` after passing through the string-literal 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.

For a similar reason, a backslash must be input as `'\\\\'` (or `'\\134'`). The first and third backslashes are interpreted as escape characters by the string-literal parser, and therefore are consumed, leaving two backslashes in the string passed to the `bytea` input function, which interprets them as representing a single backslash. For example, a string literal passed to the server as `'\\\\'` becomes `'\\'` after passing through the string-literal parser. The `'\\'` is then sent to the `bytea` input function, where it is converted to a single octet with a decimal value of 92.

A single quote is a bit different in that it must be input as `'\''` (or `'\\047'`), *not* as `'\\''`. This is because,

while the literal parser interprets the single quote as a special character, and will consume the single backslash, the `bytea` input function does *not* recognize a single quote as a special octet. Therefore a string literal passed to the backend as `'\''` becomes `''` after passing through the string-literal parser. The `''` is then sent to the `bytea` input function, where it is retains its single octet decimal value of 39.

Depending on the front end to PostgreSQL you use, you may have additional work to do in terms of escaping and unescaping `bytea` strings. For example, you may also have to escape line feeds and carriage returns if your interface automatically translates these. Or you may have to double up on backslashes if the parser for your language or choice also treats them as an escape character.

The SQL standard defines a different binary string type, called `BLOB` or `BINARY LARGE OBJECT`. The input format is different compared to `bytea`, but the provided functions and operators are mostly the same.

5.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types, shown in Table 5-9.

Table 5-9. Date/Time Types

Type	Description	Storage	Earliest	Latest	Resolution
`timestamp [ (`*p*`) ] [ without time zone ]`	both date and time	8 bytes	4713 BC	AD 1465001	1 microsecond / 14 digits
`timestamp [ (`*p*`) ] with time zone`	both date and time	8 bytes	4713 BC	AD 1465001	1 microsecond / 14 digits
`interval [ (`*p*`) ]`	time intervals	12 bytes	-178000000 years	178000000 years	1 microsecond
`date`	dates only	4 bytes	4713 BC	32767 AD	1 day
`time [ (`*p*`) ] [ without time zone ]`	times of day only	8 bytes	00:00:00.00	23:59:59.99	1 microsecond
`time [ (`*p*`) ] with time zone`	times of day only	12 bytes	00:00:00.00+12	23:59:59.99-12	1 microsecond

`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 may be less than 6, since timestamp values are stored as seconds since 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 timestamps are stored as eight-byte integers (a compile-time option), microsecond precision is available over the full range of values.

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.

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. PostgreSQL uses your operating system's underlying features to provide output time-zone support, and these systems usually contain information for only the time period 1902 through 2038 (corresponding to the full range of conventional Unix system time). `timestamp with time zone` and `time with time zone` will use time zone information only within that year range, and assume that times outside that range are in UTC.

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.

5.5.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional PostgreSQL, and others. For some formats, ordering of month and day in date input can be ambiguous and there is support for specifying the expected ordering of these fields. The command `SET DateStyle TO 'US'` or `SET DateStyle TO 'NonEuropean'` specifies the variant "month before day", the command `SET DateStyle TO 'European'` sets the variant "day before month".

PostgreSQL is more flexible in handling date/time than the SQL standard requires. See Appendix A 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 1.1.2.4 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.

5.5.1.1. Dates

Table 5-10 shows some possible inputs for the `date` type.

Table 5-10. Date Input

Example	Description
January 8, 1999	unambiguous
1999-01-08	ISO-8601 format, preferred

Example	Description
1/8/1999	U.S.; read as August 1 in European mode
8/1/1999	European; read as August 1 in U.S. mode
1/18/1999	U.S.; read as January 18 in any mode
19990108	ISO-8601 year, month, day
990108	ISO-8601 year, month, day
1999.008	year and day of year
99008	year and day of year
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

5.5.1.2. Times

The `time` type can be specified as `time` or as `time without time zone`. The optional precision *p* should be between 0 and 6, and defaults to the precision of the input time literal.

Table 5-11 shows the valid `time` inputs.

Table 5-11. Time Input

Example	Description
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	same as 04:05; AM does not affect value
`04:05 PM`	same as 16:05; input hour must be <= 12
`allballs`	same as 00:00:00

The type `time with time zone` accepts all input also legal for the `time` type, appended with a legal time zone, as shown in Table 5-12.

Table 5-12. Time With Time Zone Input

Example	Description
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601

Refer to Table 5-13 for more examples of time zones.

5.5.1.3. Time stamps

The time stamp types are `timestamp [ (p) ] without time zone` and `timestamp [ (p) ] with time zone`. Writing just `timestamp` is equivalent to `timestamp without time zone`.

> **Note:** Prior to PostgreSQL 7.3, writing just `timestamp` was equivalent to `timestamp with time zone`. This was changed for SQL spec compliance.

Valid input for the time stamp types consists of a concatenation of a date and a time, followed by an optional `AD` or `BC`, followed by an optional time zone. (See Table 5-13.) 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 optional precision *p* should be between 0 and 6, and defaults to the precision of the input `timestamp` literal.

For `timestamp without time zone`, any explicit time zone specified in the input is silently ignored. That is, the resulting date/time value is derived from the explicit date/time fields in the input value, and is not adjusted for time zone.

For `timestamp with time zone`, the internally stored value is always in UTC (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 `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 6.8.3).

Conversions between `timestamp without time zone` and `timestamp with time zone` normally assume that the `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`.

Table 5-13. Time Zone Input

Time Zone	Description
PST	Pacific Standard Time
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST

5.5.1.4. Intervals

`interval` values can be written with the following syntax:

```
      Quantity Unit [Quantity Unit...] [Direction]
    @ Quantity Unit [Quantity Unit...] [Direction]
```

where: `Quantity` is a number (possibly signed), `Unit` is `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals 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 precision *p* should be between 0 and 6, and defaults to the precision of the input literal.

5.5.1.5. Special values

The following SQL-compatible functions can be used as date or time values for the corresponding data type: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`. The latter two accept an optional precision specification. (See also Section 6.8.4.)

PostgreSQL also supports several special date/time input values for convenience, as shown in Table 5-14. 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.

Table 5-14. Special Date/Time Inputs

Input string	Description
`epoch`	1970-01-01 00:00:00+00 (Unix system time zero)
`infinity`	later than all other timestamps (not available for type `date`)
`-infinity`	earlier than all other timestamps (not available for type `date`)
`now`	current transaction time
`today`	midnight today
`tomorrow`	midnight tomorrow
`yesterday`	midnight yesterday
`zulu`, `allballs`, `z`	00:00:00.00 GMT

5.5.2. Date/Time Output

Output formats can be set to one of the four styles ISO 8601, SQL (Ingres), traditional PostgreSQL, and German, using the `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 5-15 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 5-15. Date/Time Output Styles

Style Specification	Description	Example
ISO	ISO 8601/SQL standard	1997-12-17 07:37:16-08
SQL	traditional style	12/17/1997 07:37:16.00 PST
PostgreSQL	original style	Wed Dec 17 07:37:16 1997 PST
German	regional style	17.12.1997 07:37:16.00 PST

The SQL style has European and non-European (U.S.) variants, which determines whether month follows day or vice versa. (See Section 5.5.1 for how this setting also affects interpretation of input values.) Table 5-16 shows an example.

Table 5-16. Date Order Conventions

Style Specification	Description	Example
European	*day*/*month*/*year*	17/12/1997 15:37:16.00 MET
US	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST

`interval` output looks like the input format, except that units like `week` or `century` are converted to years and days. In ISO mode the output looks like

```
[ Quantity Units [ ... ] ] [ Days ] Hours:Minutes [ ago ]
```

The date/time styles can be selected by the user using the `SET DATESTYLE` command, the `datestyle` parameter in the `postgresql.conf` configuration file, and the `PGDATESTYLE` environment variable on the server or client. The formatting function `to_char` (see Section 6.7) is also available as a more flexible way to format the date/time output.

5.5.3. Time Zones

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 can have no meaning unless associated with a date as well as a time since the offset may vary through the year with daylight-saving time boundaries.
- The default time zone is specified as a constant integer offset from GMT/UTC. It is 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 compatibility with other SQL implementations). PostgreSQL assumes your local time zone for any type containing only date or time. Further, time zone support is derived from the underlying operating system time-zone capabilities, and hence can handle daylight-saving time and other expected behavior.

PostgreSQL obtains time-zone support from the underlying operating system for dates between 1902 and 2038 (near the typical date limits for Unix-style systems). Outside of this range, all dates are assumed to be specified and used in Universal Coordinated Time (UTC).

All dates and times are stored internally in UTC, traditionally known as Greenwich Mean Time (GMT). Times are converted to local time on the database server before being sent to the client frontend, hence by default are in the server time zone.

There are several ways to select the time zone used by the server:

- The `TZ` environment variable on the server host is used by the server as the default time zone, if no other is specified.
- The `timezone` configuration parameter can be set in `postgresql.conf`.
- The `PGTZ` environment variable, if set at the client, is used by libpq applications to send a `SET TIME ZONE` command to the server upon connection.
- The SQL command `SET TIME ZONE` sets the time zone for the session.

Note: If an invalid time zone is specified, the time zone becomes UTC (on most systems anyway).

Refer to Appendix A for a list of available time zones.

5.5.4. 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.

5.6. Boolean Type

PostgreSQL provides the standard SQL type `boolean`. `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:

```
TRUE
't'
'true'
```

```
'y'
'yes'
'1'
```

For the “false” state, the following values can be used:

```
FALSE
'f'
'false'
'n'
'no'
'0'
```

Using the key words `TRUE` and `FALSE` is preferred (and SQL-compliant).

Example 5-2. 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 5-2 shows that `boolean` values are output using the letters `t` and `f`.

> **Tip:** Values of the `boolean` type cannot be cast directly to other types (e.g., `CAST (`*`boolval`*` AS integer)` does not work). This can be accomplished using the `CASE` expression: `CASE WHEN `*`boolval`*` THEN 'value if true' ELSE 'value if false' END`. See also Section 6.12.

`boolean` uses 1 byte of storage.

5.7. Geometric Types

Geometric data types represent two-dimensional spatial objects. Table 5-17 shows the geometric types available in PostgreSQL. The most fundamental type, the point, forms the basis for all of the other types.

Geometric Type	Storage	Representation	Description

Table 5-17. Geometric Types

Geometric Type	Storage	Representation	Description
`point`	16 bytes	(x,y)	Point in space
`line`	32 bytes	((x1,y1),(x2,y2))	Infinite line (not fully implemented)
`lseg`	32 bytes	((x1,y1),(x2,y2))	Finite line segment
`box`	32 bytes	((x1,y1),(x2,y2))	Rectangular box
`path`	16+16n bytes	((x1,y1),...)	Closed path (similar to polygon)
`path`	16+16n bytes	[(x1,y1),...]	Open path
`polygon`	40+16n bytes	((x1,y1),...)	Polygon (similar to closed path)
`circle`	24 bytes	<(x,y),r>	Circle (center and radius)

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 6.9.

5.7.1. Point

Points are the fundamental two-dimensional building block for geometric types. `point` is specified using the following syntax:

```
( x , y )
  x , y
```

where the arguments are

x

the x-axis coordinate as a floating-point number

y

the y-axis coordinate as a floating-point number

5.7.2. Line Segment

Line segments (`lseg`) are represented by pairs of points. `lseg` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

(*x1*,*y1*)
(*x2*,*y2*)

the end points of the line segment

5.7.3. Box

Boxes are represented by pairs of points that are opposite corners of the box. `box` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

(*x1*,*y1*)
(*x2*,*y2*)

opposite corners of the box

Boxes are output using the first syntax. The corners are reordered on input to store the upper right corner, then the lower left corner. Other corners of the box can be entered, but the lower left and upper right corners are determined from the input and stored corners.

5.7.4. Path

Paths are represented by connected sets of points. Paths can be *open*, where the first and last points in the set are not connected, and *closed*, where the first and last point are connected. Functions `popen(p)` and `pclose(p)` are supplied to force a path to be open or closed, and functions `isopen(p)` and `isclosed(p)` are supplied to test for either type in a query.

`path` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

End points of the line segments comprising the path. A leading square bracket (`[`) indicates an open path, while a leading parenthesis (`(`) indicates a closed path.

Paths are output using the first syntax.

5.7.5. Polygon

Polygons are represented by sets of points. Polygons should probably be considered equivalent to closed paths, but are stored differently and have their own set of support routines.

`polygon` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

End points of the line segments comprising the boundary of the polygon

Polygons are output using the first syntax.

5.7.6. Circle

Circles are represented by a center point and a radius. `circle` is specified using the following syntax:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

where the arguments are

(*x*,*y*)

center of the circle

r

radius of the circle

Circles are output using the first syntax.

5.8. Network Address Data Types

PostgreSQL offers data types to store IP and MAC addresses, shown in Table 5-18. It is preferable to use these types over plain text types, because these types offer input error checking and several specialized operators and functions.

Table 5-18. Network Address Data Types

Name	Storage	Description	Range
`cidr`	12 bytes	IP networks	valid IPv4 networks
`inet`	12 bytes	IP hosts and networks	valid IPv4 hosts or networks
`macaddr`	6 bytes	MAC addresses	customary formats

IPv6 is not yet supported.

5.8.1. `inet`

The `inet` type holds an IP host address, and optionally the identity of the subnet it is in, all in one field. The subnet identity is represented by the number of bits in the network part of the address (the "netmask"). If the netmask is 32, then the value does not indicate a subnet, only a single host. 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 *x.x.x.x/y* where *x.x.x.x* is an IP address and *y* is the number of bits in the netmask. If the */y* part is left off, then the netmask is 32, and the value represents just a single host. On display, the */y* portion is suppressed if the netmask is 32.

5.8.2. `cidr`

The `cidr` type holds an IP network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying classless networks is *x.x.x.x/y* where *x.x.x.x* is the network and *y* is the number of bits in the netmask. If *y* is omitted, it is calculated using assumptions from the older classful numbering system, except that it will be at least large enough to include all of the octets written in the input.

Table 5-19 shows some examples.

Table 5-19. `cidr` Type Input Examples

`CIDR` Input	`CIDR` Displayed	`abbrev(CIDR)`
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8

5.8.3. `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 `host()`, `text()`, and `abbrev()` functions.

5.8.4. `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 shown forms.

The directory `contrib/mac` in the PostgreSQL source distribution contains tools that can be used to map MAC addresses to hardware manufacturer names.

5.9. 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.

> **Note:** Prior to PostgreSQL 7.2, `BIT` data was always silently truncated or zero-padded on the right, with or without an explicit cast. This was changed to comply with the SQL standard.

Refer to Section 1.1.2.2 for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see Chapter 6.

Example 5-3. 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
```

5.10. Object Identifier Types

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. Also, an OID system column is added to user-created tables (unless `WITHOUT OIDS` is specified at table creation time). Type `oid` represents an object identifier. There are also several aliases for `oid`: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regclass`, and `regtype`. Table 5-20 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 (which is implemented as unsigned 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, one may write 'mytable'::regclass to get the OID of table mytable, rather than SELECT oid FROM pg_class WHERE relname = 'mytable'. (In reality, a much more complicated SELECT would be needed to deal with selecting the right OID when there are multiple tables named mytable in different schemas.)

Table 5-20. Object Identifier Types

Type name	References	Description	Value example
oid	any	numeric object identifier	564182
regproc	pg_proc	function name	sum
regprocedure	pg_proc	function with argument types	sum(int4)
regoper	pg_operator	operator name	+
regoperator	pg_operator	operator with argument types	*(integer,integer) or -(NONE,integer)
regclass	pg_class	relation name	pg_type
regtype	pg_type	type name	integer

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.

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 they are unique. Recommended practice when using OIDs for row identification is to create a unique constraint on the OID column of each table for which the OID will be used. Never assume that OIDs are unique across tables; use the combination of tableoid and row OID if you need a database-wide identifier. (Future releases of PostgreSQL are likely to use a separate OID counter for each table, so that tableoid *must* be included to arrive at a globally unique identifier.)

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. 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 the *PostgreSQL Administrator's Guide* for details. However, it is unwise to depend on uniqueness of transaction IDs over the long term (more than one billion transactions).

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. 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 tuples processed.

A final identifier type used by the system is `tid`, or tuple 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 tuple within its table.

5.11. 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 5-21 lists the existing pseudo-types.

Table 5-21. Pseudo-Types

Type name	Description
`record`	Identifies a function returning an unspecified row type
`any`	Indicates that a function accepts any input data type whatever
`anyarray`	Indicates that a function accepts any array data type
`void`	Indicates that a function returns no value
`trigger`	A trigger function is declared to return `trigger`
`language_handler`	A procedural language call handler is declared to return `language_handler`
`cstring`	Indicates that a function accepts or returns a null-terminated C string
`internal`	Indicates that a function accepts or returns a server-internal data type
`opaque`	An obsolete type name that formerly served all the above purposes

Functions coded in C (whether built-in or dynamically loaded) may 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 may 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` as a result type (plus `trigger` when the function is used as a trigger).

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.

5.12. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in type or user-defined type can be created. To illustrate their use, we create this table:

```
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 columns including a `text` string (`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.

Now we do some `INSERT`s. Observe that to write an array value, we enclose the element values within curly braces and separate them by commas. If you know C, this is not unlike the syntax for initializing structures. (More details appear below.)

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

Now, we can run some queries on `sal_emp`. 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. 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},{""}}
(1 row)
```

We could also have written

```
SELECT schedule[1:2][1] FROM sal_emp WHERE name = 'Bill';
```

with the same result. An array subscripting operation is taken to represent an array slice if any of the subscripts are written in the form *lower*:*upper*. A lower bound of 1 is assumed for any subscript where only one value is specified.

An array value can be replaced completely:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or updated at a single element:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

or updated in a slice:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

An array can be enlarged by assigning to an element adjacent to those already present, or by assigning to a slice that is adjacent to or overlaps the data already present. For example, if an array value currently has 4 elements, it will have five elements after an update that assigns to `array[5]`. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

Array slice assignment allows creation of arrays that do not use one-based subscripts. For example one might assign to `array[-2:7]` to create an array with subscript values running from -2 to 7.

The syntax for `CREATE TABLE` allows fixed-length arrays to be defined:

```
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 runtime behavior.

The current dimensions of any array value can be retrieved with the `array_dims` function:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:1]
(1 row)
```

`array_dims` produces a `text` result, which is convenient for people to read but perhaps not so convenient for programs.

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):

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is unknown. Although it is not part of the primary PostgreSQL distribution, there is an extension available that defines new functions and operators for iterating over array values. Using this, the above query could be:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1:4] *= 10000;
```

To search the entire array (not just specified columns), you could use:

```
SELECT * FROM sal_emp WHERE pay_by_quarter *= 10000;
```

In addition, you could find rows where the array had all values equal to 10 000 with:

```
SELECT * FROM sal_emp WHERE pay_by_quarter **= 10000;
```

To install this optional module, look in the `contrib/array` directory of the PostgreSQL source distribution.

> **Tip:** Arrays are not sets; using arrays in the manner described in the previous paragraph is often a sign of database misdesign. The array field should generally be split off into a separate table. Tables can obviously be searched easily.

> **Note:** A limitation of the present array implementation is that individual elements of an array cannot be SQL null values. The entire array can be set to null, but you can't have an array with some elements null and some not. Fixing this is on the to-do list.

Array input and output syntax. The external 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. You may write whitespace before a left brace, after a right brace, or before any individual item string. Whitespace after an item is not ignored, however: after skipping leading whitespace, everything up to the next right brace or delimiter is taken as the item value.

Quoting array elements. As shown above, when writing an array value you may 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 white space must be double-quoted. To put a double quote or backslash in an array element value, precede it with a backslash. Alternatively, you can use backslash-escaping to protect all data characters that would otherwise be taken as array syntax or ignorable white space.

The array output routine will put double quotes around element values if they are empty strings or contain curly braces, delimiter characters, double quotes, backslashes, or white space. 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. (This is a change in behavior from pre-7.2 PostgreSQL releases.)

> **Tip:** 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
>
> ```
> INSERT ... VALUES ('{"\\\\","\\""}');
> ```
>
> The string-literal 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.)

Chapter 6. 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 the *PostgreSQL Programmer's Guide*. 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 this extended functionality is present in other SQL implementations, and in many cases this functionality is compatible and consistent between various products.

6.1. 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:

a	***b***	***a* AND *b***	***a* OR *b***
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT *a***
TRUE	FALSE
FALSE	TRUE
NULL	NULL

6.2. Comparison Operators

The usual comparison operators are available, shown in Table 6-1.

Table 6-1. Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

> **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 BETWEEN x AND y
```

is equivalent to

```
a >= x AND a <= y
```

Similarly,

```
a NOT BETWEEN x AND y
```

is equivalent to

```
a < x 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.

To check whether a value is or is not null, use the constructs

```
expression IS NULL
expression IS NOT NULL
```

or the equivalent, but nonstandard, constructs

```
expression ISNULL
expression NOTNULL
```

Do *not* write `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.)

Some applications may (incorrectly) require that *expression* = NULL returns true if *expression* evaluates to the null value. To support these applications, the run-time option transform_null_equals can be turned on (e.g., SET transform_null_equals TO ON;). PostgreSQL will then convert x = NULL clauses to x IS NULL. This was the default behavior in releases 6.5 through 7.1.

Boolean values can also be tested using the constructs

```
expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

These are similar to IS NULL in that they 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”.

6.3. 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 6-2 shows the available mathematical operators.

Table 6-2. Mathematical Operators

Name	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division (integer division truncates results)	4 / 2	2
%	modulo (remainder)	5 % 4	1
^	exponentiation	2.0 ^ 3.0	8
|/	square root	|/ 25.0	5
||/	cube root	||/ 27.0	3
!	factorial	5 !	120
!!	factorial (prefix operator)	!! 5	120
@	absolute value	@ -5.0	5
&	binary AND	91 & 15	11
|	binary OR	32 | 3	35
#	binary XOR	17 # 5	20
~	binary NOT	~1	-2
<<	binary shift left	1 << 4	16

Name	Description	Example	Result
>>	binary shift right	8 >> 2	2

The “binary” operators are also available for the bit string types `BIT` and `BIT VARYING`, as shown in Table 6-3. Bit string arguments to `&`, `|`, and # must be of equal length. When bit shifting, the original length of the string is preserved, as shown in the table.

Table 6-3. Bit String Binary Operators

Example	Result
B’10001’ & B’01101’	00001
B’10001’ | B’01101’	11101
B’10001’ # B’01101’	11110
~ B’10001’	01110
B’10001’ << 3	01000
B’10001’ >> 2	00100

Table 6-4 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 datatype 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 may therefore vary depending on the host system.

Table 6-4. Mathematical Functions

Function	Return Type	Description	Example	Result
`abs(`*x*`)`	(same as *x*)	absolute value	`abs(-17.4)`	17.4
`cbrt(dp)`	`dp`	cube root	`cbrt(27.0)`	3
`ceil(dp` or `numeric)`	(same as input)	smallest integer not less than argument	`ceil(-42.8)`	-42
`degrees(dp)`	`dp`	radians to degrees	`degrees(0.5)`	28.6478897565412
`exp(dp` or `numeric)`	(same as input)	exponential	`exp(1.0)`	2.71828182845905
`floor(dp` or `numeric)`	(same as input)	largest integer not greater than argument	`floor(-42.8)`	-43
`ln(dp` or `numeric)`	(same as input)	natural logarithm	`ln(2.0)`	0.693147180559945
`log(dp` or `numeric)`	(same as input)	base 10 logarithm	`log(100.0)`	2
`log(`*b* `numeric,` *x* `numeric)`	`numeric`	logarithm to base *b*	`log(2.0, 64.0)`	6.0000000000
`mod(`*y*`,` *x*`)`	(same as argument types)	remainder of *y*/*x*	`mod(9,4)`	1

Function	Return Type	Description	Example	Result
pi()	dp	"Pi" constant	pi()	3.14159265358979
pow(*x* dp, *e* dp)	dp	raise a number to exponent *e*	pow(9.0, 3.0)	729
pow(*x* numeric, *e* numeric)	numeric	raise a number to exponent *e*	pow(9.0, 3.0)	729
radians(dp)	dp	degrees to radians	radians(45.0)	0.785398163397448
random()	dp	random value between 0.0 and 1.0	random()	
round(dp or numeric)	(same as input)	round to nearest integer	round(42.4)	42
round(*v* numeric, *s* integer)	numeric	round to *s* decimal places	round(42.4382, 2)	42.44
sign(dp or numeric)	(same as input)	sign of the argument (-1, 0, +1)	sign(-8.4)	-1
sqrt(dp or numeric)	(same as input)	square root	sqrt(2.0)	1.4142135623731
trunc(dp or numeric)	(same as input)	truncate toward zero	trunc(42.8)	42
trunc(*v* numeric, *s* integer)	numeric	truncate to *s* decimal places	trunc(42.4382, 2)	42.43

Finally, Table 6-5 shows the available trigonometric functions. All trigonometric functions take arguments and return values of type `double precision`.

Table 6-5. Trigonometric Functions

Function	Description
acos(*x*)	inverse cosine
asin(*x*)	inverse sine
atan(*x*)	inverse tangent
atan2(*x*, *y*)	inverse tangent of *x*/*y*
cos(*x*)	cosine
cot(*x*)	cotangent
sin(*x*)	sine
tan(*x*)	tangent

6.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of all 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 the automatic padding when using the CHARACTER type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first. Some functions also exist natively for bit-string types.

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 6-6. These functions are also implemented using the regular syntax for function invocation. (See Table 6-7.)

Table 6-6. SQL String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	text	String concatenation	'Post' || 'greSQL'	PostgreSQL
bit_length(*string*)	integer	Number of bits in string	bit_length('jose')	32
char_length(*string*) or character_length(*string*)	integer	Number of characters in string	char_length('jose')	4
convert(*string* using *conversion_name*)	text	Change encoding using specified conversion name. Conversions can be defined by CREATE CONVERSION. Also there are some pre-defined conversion names. See Table 6-8 for available conversion names.	convert('PostgreSQL' using iso_8859_1_to_utf_8)	'PostgreSQL' in Unicode (UTF-8) encoding
lower(*string*)	text	Convert string to lower case	lower('TOM')	tom
octet_length(*string*)	integer	Number of bytes in string	octet_length('jose')	4
overlay(*string* placing *string* from integer [for integer])	text	Insert substring	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(*substring* in *string*)	integer	Location of specified substring	position('om' in 'Thomas')	3

Function	Return Type	Description	Example	Result
substring(*string* [from integer] [for integer])	text	Extract substring	substring('Thomas' from 2 for 3)	hom
substring(*string* from *pattern*)	text	Extract substring matching POSIX regular expression	substring('Thomas' from '...$')	mas
substring(*string* from *pattern* for *escape*)	text	Extract substring matching SQL regular expression	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [*characters*] from *string*)	text	Remove the longest string containing only the *characters* (a space by default) from the beginning/end/both ends of the *string*	trim(both 'x' from 'xTomxx')	Tom
upper(*string*)	text	Convert string to upper case	upper('tom')	TOM

Additional string manipulation functions are available and are listed in Table 6-7. Some of them are used internally to implement the SQL-standard string functions listed in Table 6-6.

Table 6-7. Other String Functions

Function	Return Type	Description	Example	Result
ascii(text)	integer	ASCII code of the first character of the argument.	ascii('x')	120
btrim(*string* text, *trim* text)	text	Remove (trim) the longest string consisting only of characters in *trim* from the start and end of *string*	btrim('xyxtrimyyx', 'xy')	trim
chr(integer)	text	Character with the given ASCII code	chr(65)	A

Function	Return Type	Description	Example	Result
convert(*string* text, [*src_encoding* name,] *dest_encoding* name)	text	Convert string to *dest_encoding*. The original encoding is specified by *src_encoding*. If *src_encoding* is omitted, database encoding is assumed.	convert('text_in_unicode', 'UNICODE', 'LATIN1')	text_in_unicode represented in ISO 8859-1
decode(*string* text, *type* text)	bytea	Decode binary data from *string* previously encoded with encode(). Parameter type is same as in encode().	decode('MTIzAAE=', 'base64')	123\000\001
encode(*data* bytea, *type* text)	text	Encode binary data to ASCII-only representation. Supported types are: base64, hex, escape.	encode('123\\000\\001', 'base64')	MTIzAAE=
initcap(text)	text	Convert first letter of each word (whitespace separated) to upper case	initcap('hi thomas')	Hi Thomas
length(*string*)	integer	Length of string	length('jose')	4
lpad(*string* text, *length* integer [, *fill* text])	text	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).	lpad('hi', 5, 'xy')	xyxhi

Function	Return Type	Description	Example	Result
ltrim(*string* text, *text* text)	text	Remove the longest string containing only characters from *trim* from the start of the string.	ltrim('zzzytrim', 'xyz')	trim
pg_client_encoding()	name	Current client encoding name.	pg_client_encoding()	SQL_ASCII
quote_ident(*string* text)	text	Return the given string suitably quoted to be used as an identifier in an SQL query 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.	quote_ident('Foo')	"Foo"
quote_literal(*string* text)	text	Return the given string suitably quoted to be used as a literal in an SQL query string. Embedded quotes and backslashes are properly doubled.	quote_literal('O'Reilly')	'O''Reilly'
repeat(text, integer)	text	Repeat text a number of times	repeat('Pg', 4)	PgPgPgPg
replace(*string* text, *from* text, *to* text)	text	Replace all occurrences in *string* of substring *from* with substring *to*	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef

Function	Return Type	Description	Example	Result
rpad(*string* text, *length* integer [, *fill* text])	text	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.	rpad('hi', 5, 'xy')	hixyx
rtrim(*string* text, *trim* text)	text	Remove the longest string containing only characters from *trim* from the end of the string.	rtrim('trimxxxx','x')	trim
split_part(*string* text, *delimiter* text, *column* integer)	text	Split *string* on *delimiter* returning the resulting (one based) *column* number.	split_part('abc~@~def~@~ghi','~@~',2)	def
strpos(*string*, *substring*)	text	Locate specified substring (same as position(*substring* in *string*), but note the reversed argument order)	strpos('high','ig')	2
substr(*string*, *from* [, *count*])	text	Extract specified substring (same as substring(*string* from *from* for *count*))	substr('alphabet', 3, 2)	ph
to_ascii(text [, *encoding*])	text	Convert text to ASCII from other encoding [a]	to_ascii('Karel')	Karel
to_hex(*number* integer or bigint)	text	Convert *number* to its equivalent hexadecimal representation	to_hex(9223372036854775807::bigint)	7fffffffffffffff

Function	Return Type	Description	Example	Result
`trans-late(`*`string`* `text,` *`from`* `text,` *`to`* `text)`	`text`	Any character in *`string`* that matches a character in the *`from`* set is replaced by the corresponding character in the *`to`* set.	`translate('12345', '14', 'ax')`	a23x5

Notes:
a. The `to_ascii` function supports conversion from `LATIN1`, `LATIN2`, and `WIN1250` only.

Table 6-8. Built-in Conversions

Conversion Name [a]	Source Encoding	Destination Encoding
`ascii_to_mic`	`SQL_ASCII`	`MULE_INTERNAL`
`ascii_to_utf_8`	`SQL_ASCII`	`UNICODE`
`big5_to_euc_tw`	`BIG5`	`EUC_TW`
`big5_to_mic`	`BIG5`	`MULE_INTERNAL`
`big5_to_utf_8`	`BIG5`	`UNICODE`
`euc_cn_to_mic`	`EUC_CN`	`MULE_INTERNAL`
`euc_cn_to_utf_8`	`EUC_CN`	`UNICODE`
`euc_jp_to_mic`	`EUC_JP`	`MULE_INTERNAL`
`euc_jp_to_sjis`	`EUC_JP`	`SJIS`
`euc_jp_to_utf_8`	`EUC_JP`	`UNICODE`
`euc_kr_to_mic`	`EUC_KR`	`MULE_INTERNAL`
`euc_kr_to_utf_8`	`EUC_KR`	`UNICODE`
`euc_tw_to_big5`	`EUC_TW`	`BIG5`
`euc_tw_to_mic`	`EUC_TW`	`MULE_INTERNAL`
`euc_tw_to_utf_8`	`EUC_TW`	`UNICODE`
`gb18030_to_utf_8`	`GB18030`	`UNICODE`
`gbk_to_utf_8`	`GBK`	`UNICODE`
`iso_8859_10_to_utf_8`	`LATIN6`	`UNICODE`
`iso_8859_13_to_utf_8`	`LATIN7`	`UNICODE`
`iso_8859_14_to_utf_8`	`LATIN8`	`UNICODE`
`iso_8859_15_to_utf_8`	`LATIN9`	`UNICODE`
`iso_8859_16_to_utf_8`	`LATIN10`	`UNICODE`
`iso_8859_1_to_mic`	`LATIN1`	`MULE_INTERNAL`
`iso_8859_1_to_utf_8`	`LATIN1`	`UNICODE`
`iso_8859_2_to_mic`	`LATIN2`	`MULE_INTERNAL`
`iso_8859_2_to_utf_8`	`LATIN2`	`UNICODE`

Conversion Name [a]	Source Encoding	Destination Encoding
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf_8	LATIN3	UNICODE
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf_8	LATIN4	UNICODE
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf_8	ISO_8859_5	UNICODE
iso_8859_5_to_windows_1251	ISO_8859_5	WIN
iso_8859_5_to_windows_866	ISO_8859_5	ALT
iso_8859_6_to_utf_8	ISO_8859_6	UNICODE
iso_8859_7_to_utf_8	ISO_8859_7	UNICODE
iso_8859_8_to_utf_8	ISO_8859_8	UNICODE
iso_8859_9_to_utf_8	LATIN5	UNICODE
johab_to_utf_8	JOHAB	UNICODE
koi8_r_to_iso_8859_5	KOI8	ISO_8859_5
koi8_r_to_mic	KOI8	MULE_INTERNAL
koi8_r_to_utf_8	KOI8	UNICODE
koi8_r_to_windows_1251	KOI8	WIN
koi8_r_to_windows_866	KOI8	ALT
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN
mic_to_windows_866	MULE_INTERNAL	ALT

Conversion Name [a]	Source Encoding	Destination Encoding
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf_8	SJIS	UNICODE
tcvn_to_utf_8	TCVN	UNICODE
uhc_to_utf_8	UHC	UNICODE
utf_8_to_ascii	UNICODE	SQL_ASCII
utf_8_to_big5	UNICODE	BIG5
utf_8_to_euc_cn	UNICODE	EUC_CN
utf_8_to_euc_jp	UNICODE	EUC_JP
utf_8_to_euc_kr	UNICODE	EUC_KR
utf_8_to_euc_tw	UNICODE	EUC_TW
utf_8_to_gb18030	UNICODE	GB18030
utf_8_to_gbk	UNICODE	GBK
utf_8_to_iso_8859_1	UNICODE	LATIN1
utf_8_to_iso_8859_10	UNICODE	LATIN6
utf_8_to_iso_8859_13	UNICODE	LATIN7
utf_8_to_iso_8859_14	UNICODE	LATIN8
utf_8_to_iso_8859_15	UNICODE	LATIN9
utf_8_to_iso_8859_16	UNICODE	LATIN10
utf_8_to_iso_8859_2	UNICODE	LATIN2
utf_8_to_iso_8859_3	UNICODE	LATIN3
utf_8_to_iso_8859_4	UNICODE	LATIN4
utf_8_to_iso_8859_5	UNICODE	ISO_8859_5
utf_8_to_iso_8859_6	UNICODE	ISO_8859_6
utf_8_to_iso_8859_7	UNICODE	ISO_8859_7
utf_8_to_iso_8859_8	UNICODE	ISO_8859_8
utf_8_to_iso_8859_9	UNICODE	LATIN5
utf_8_to_johab	UNICODE	JOHAB
utf_8_to_koi8_r	UNICODE	KOI8
utf_8_to_sjis	UNICODE	SJIS
utf_8_to_tcvn	UNICODE	TCVN
utf_8_to_uhc	UNICODE	UHC
utf_8_to_windows_1250	UNICODE	WIN1250
utf_8_to_windows_1251	UNICODE	WIN
utf_8_to_windows_1256	UNICODE	WIN1256
utf_8_to_windows_866	UNICODE	ALT
utf_8_to_windows_874	UNICODE	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2

Conversion Name [a]	Source Encoding	Destination Encoding
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf_8	WIN1250	UNICODE
windows_1251_to_iso_8859_5	WIN	ISO_8859_5
windows_1251_to_koi8_r	WIN	KOI8
windows_1251_to_mic	WIN	MULE_INTERNAL
windows_1251_to_utf_8	WIN	UNICODE
windows_1251_to_windows_866	WIN	ALT
windows_1256_to_utf_8	WIN1256	UNICODE
windows_866_to_iso_8859_5	ALT	ISO_8859_5
windows_866_to_koi8_r	ALT	KOI8
windows_866_to_mic	ALT	MULE_INTERNAL
windows_866_to_utf_8	ALT	UNICODE
windows_866_to_windows_1251	ALT	WIN
windows_874_to_utf_8	WIN874	UNICODE

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.

6.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating binary string values. Strings in this context mean values of the 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 6-9. Some functions are also implemented using the regular syntax for function invocation. (See Table 6-10.)

Table 6-9. SQL Binary String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	bytea	String concatenation	'\\\\Post'::bytea || '\\047greSQL\\000'::bytea	\\Post'greSQL\000
octet_length(*string*)	integer	Number of bytes in binary string	octet_length('jo\\000se'::bytea)	5

Function	Return Type	Description	Example	Result
position(*substring* in *string*)	integer	Location of specified substring	position('\\000om'::bytea in 'Th\\000omas'::bytea)	3
substring(*string* [from integer] [for integer])	bytea	Extract substring	substring('Th\\000omas'::bytea from 2 for 3)	h\000o
trim([both] *characters* from *string*)	bytea	Remove the longest string containing only the *characters* from the beginning/end/both ends of the *string*	trim('\\000'::bytea from '\\000Tom\\000'::bytea)	Tom
get_byte(*string*, *offset*)	integer	Extract byte from string.	get_byte('Th\\000omas'::bytea, 4)	109
set_byte(*string*, *offset*, *newvalue*)	bytea	Set byte in string.	set_byte('Th\\000omas'::bytea, 4, 64)	Th\000o@as
get_bit(*string*, *offset*)	integer	Extract bit from string.	get_bit('Th\\000omas'::bytea, 45)	1
set_bit(*string*, *offset*, *newvalue*)	bytea	Set bit in string.	set_bit('Th\\000omas'::bytea, 45, 0)	Th\000lmas

Additional binary string manipulation functions are available and are listed in Table 6-10. Some of them are used internally to implement the SQL-standard string functions listed in Table 6-9.

Table 6-10. Other Binary String Functions

Function	Return Type	Description	Example	Result
btrim(*string* bytea *trim* bytea)	bytea	Remove (trim) the longest string consisting only of characters in *trim* from the start and end of *string*.	btrim('\\000trim\\000'::bytea,'\\000'::bytea	trim
length(*string*)	integer	Length of binary string	length('jo\\000se'::bytea)	5

Function	Return Type	Description	Example	Result
encode(*string* bytea, *type* text)	text	Encode binary string to ASCII-only representation. Supported types are: base64, hex, escape.	encode('123\\000456'::bytea, 'escape')	123\000456
decode(*string* text, *type* text)	bytea	Decode binary string from *string* previously encoded with encode(). Parameter type is same as in encode().	decode('123\\000456', 'escape')	123\000456

6.6. Pattern Matching

There are three separate approaches to pattern matching provided by PostgreSQL: the traditional SQL LIKE operator, the more recent SQL99 SIMILAR TO operator, and POSIX-style regular expressions. Additionally, a pattern matching function, SUBSTRING, is available, using either SQL99-style or POSIX-style regular expressions.

> **Tip:** If you have pattern matching needs that go beyond this, consider writing a user-defined function in Perl or Tcl.

6.6.1. 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:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

LIKE pattern matches always cover the entire string. To match a pattern 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 may 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 the query. Thus, writing a pattern that actually matches a literal backslash means writing four backslashes in the query. You can avoid this by selecting a different escape character with ESCAPE; then 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 keyword 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. All of these operators are PostgreSQL-specific.

6.6.2. SIMILAR TO and SQL99 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 SQL99's definition of a regular expression. SQL99's 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 may match any part of the string. Also like LIKE, SIMILAR TO uses % and _ as wildcard characters denoting any string and any single character, 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 ( ) may 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, 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`*`)`, provides extraction of a substring that matches a SQL99 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, SQL99 specifies that the pattern must contain two occurrences of the escape character followed by 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
```

6.6.3. POSIX Regular Expressions

Table 6-11 lists the available operators for pattern matching using POSIX regular expressions.

Table 6-11. Regular Expression Match Operators

Operator	Description	Example
`~`	Matches regular expression, case sensitive	`'thomas' ~ '.*thomas.*'`
`~*`	Matches regular expression, case insensitive	`'thomas' ~* '.*Thomas.*'`
`!~`	Does not match regular expression, case sensitive	`'thomas' !~ '.*Thomas.*'`
`!~*`	Does not match regular expression, case insensitive	`'thomas' !~* '.*vadim.*'`

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 `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 always put parentheses around the whole expression if you want to use parentheses within it without triggering this exception.

Some examples:

```
SUBSTRING('foobar' FROM 'o.b')     oob
SUBSTRING('foobar' FROM 'o(.)b')   o
```

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of `egrep`; 1003.2 calls these “extended” REs) and obsolete REs (roughly those of `ed`; 1003.2 “basic” REs). PostgreSQL implements the modern form.

A (modern) RE is one or more non-empty *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is one or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by a single `*`, `+`, `?`, or *bound*. An atom followed by `*` matches a sequence of 0 or more matches of the atom. An atom followed by `+` matches a sequence of 1 or more matches of the atom. An atom followed by `?` matches a sequence of 0 or 1 matches of the atom.

A *bound* is `{` followed by an unsigned decimal integer, possibly followed by `,` possibly followed by another unsigned decimal integer, always followed by `}`. The integers must lie between 0 and `RE_DUP_MAX` (255) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer *`i`* and no comma matches a sequence of exactly *`i`* matches of the atom. An atom followed by a bound containing one integer *`i`* and a comma matches a sequence of *`i`* or more matches of the atom. An atom followed by a bound containing two integers *`i`* and *`j`* matches a sequence of *`i`* through *`j`* (inclusive) matches of the atom.

> **Note:** A repetition operator (`?`, `*`, `+`, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow `^` or `|`.

An *atom* is a regular expression enclosed in `()` (matching a match for the regular expression), an empty set of `()` (matching the null string), a *bracket expression* (see below), `.` (matching any single character), `^` (matching the null string at the beginning of the input string), `$` (matching the null string at the end of the input string), a `\` followed by one of the characters `^.[$()|*+?{\` (matching that character taken as an ordinary character), a `\` followed by any other character (matching that character taken as an ordinary character, as if the `\` had not been present), or a single character with no other significance (matching that character). A `{` followed by a character other than a digit is an ordinary character, not the beginning of a bound. It is illegal to end an RE with `\`.

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 the query.

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 (but see below) 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, and 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 and some combinations using `[` (see next paragraphs), all other special characters, including `\`, lose their special significance within a bracket expression.

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.]]*c` matches the first five characters of `chchcc`.

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 may not 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 may provide others. A character class may not be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions `[[:<:]]` and `[[:>:]]` match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which 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.

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, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint

that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, `bb*` matches the three middle characters of `abbbc`, `(wee|week)(knights|nights)` 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 the null 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, so that (e.g.) `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. Bounded repetitions are implemented by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, `((((a{1,100}){1,100}){1,100}){1,100}){1,100}` will (eventually) run almost any existing machine out of swap space. [1]

6.7. 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 6-12 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.

Table 6-12. Formatting Functions

Function	Returns	Description	Example
`to_char(timestamp, text)`	`text`	convert time stamp to string	`to_char(timestamp 'now','HH12:MI:SS')`
`to_char(interval, text)`	`text`	convert interval to string	`to_char(interval '15h 2m 12s','HH24:MI:SS')`
`to_char(int, text)`	`text`	convert integer to string	`to_char(125, '999')`
`to_char(double precision, text)`	`text`	convert real/double precision to string	`to_char(125.8, '999D9')`

1. This was written in 1994, mind you. The numbers have probably changed, but the problem persists.

Function	Returns	Description	Example
to_char(numeric, text)	text	convert numeric to string	to_char(numeric '-125.8', '999D99S')
to_date(text, text)	date	convert string to date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(text, text)	timestamp	convert string to time stamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	convert string to numeric	to_number('12,454.8-', '99G999D9S')

In an output template string, 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, template patterns identify the parts of the input data string to be looked at and the values to be found there.

Table 6-13 shows the template patterns available for formatting date and time values.

Table 6-13. Template patterns for date/time conversions

Pattern	Description
HH	hour of day (01-12)
HH12	hour of day (01-12)
HH24	hour of day (00-23)
MI	minute (00-59)
SS	second (00-59)
MS	millisecond (000-999)
US	microsecond (000000-999999)
SSSS	seconds past midnight (0-86399)
AM or A.M. or PM or P.M.	meridian indicator (upper case)
am or a.m. or pm or p.m.	meridian indicator (lower case)
Y,YYY	year (4 and more digits) with comma
YYYY	year (4 and more digits)
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
BC or B.C. or AD or A.D.	era indicator (upper case)
bc or b.c. or ad or a.d.	era indicator (lower case)
MONTH	full upper case month name (blank-padded to 9 chars)

Pattern	Description
Month	full mixed case month name (blank-padded to 9 chars)
month	full lower case month name (blank-padded to 9 chars)
MON	abbreviated upper case month name (3 chars)
Mon	abbreviated mixed case month name (3 chars)
mon	abbreviated lower case month name (3 chars)
MM	month number (01-12)
DAY	full upper case day name (blank-padded to 9 chars)
Day	full mixed case day name (blank-padded to 9 chars)
day	full lower case day name (blank-padded to 9 chars)
DY	abbreviated upper case day name (3 chars)
Dy	abbreviated mixed case day name (3 chars)
dy	abbreviated lower case day name (3 chars)
DDD	day of year (001-366)
DD	day of month (01-31)
D	day of week (1-7; SUN=1)
W	week of month (1-5) where first week start on the first day of the month
WW	week number of year (1-53) where first week start on the first day of the year
IW	ISO week number of year (The first Thursday of the new year is in week 1.)
CC	century (2 digits)
J	Julian Day (days since January 1, 4712 BC)
Q	quarter
RM	month in Roman Numerals (I-XII; I=January) - upper case
rm	month in Roman Numerals (I-XII; I=January) - lower case
TZ	time-zone name - upper case
tz	time-zone name - lower case

Certain modifiers may be applied to any template pattern to alter its behavior. For example, "FMMonth" is the "Month" pattern with the "FM" prefix. Table 6-14 shows the modifier patterns for date/time formatting.

Table 6-14. Template pattern modifiers for date/time conversions

Modifier	Description	Example

Modifier	Description	Example
FM prefix	fill mode (suppress padding blanks and zeroes)	FMMonth
TH suffix	add upper-case ordinal number suffix	DDTH
th suffix	add lower-case ordinal number suffix	DDth
FX prefix	fixed format global option (see usage notes)	FX Month DD Day
SP suffix	spell mode (not yet implemented)	DDSP

Usage notes for the 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.
- 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 right, but to_timestamp('2000 JUN','FXYYYY MON') returns an error, because to_timestamp expects one blank space only.
- If a backslash ("\") is desired in a string constant, a double backslash ("\\") must be entered; for example '\\HH\\MI\\SS'. This is true for any string constant in PostgreSQL.
- 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 keywords. 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 '\\"YYYY Month\\"'.
- YYYY conversion from string to timestamp or date is restricted 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 year 20000): to_date('200001131', 'YYYYMMDD') will be interpreted as a 4-digit year; better is to use a non-digit separator after the year, like to_date('20000-1131', 'YYYY-MMDD') or to_date('20000Nov31', 'YYYYMonDD').
- Millisecond MS and microsecond US values in a conversion from string to time stamp 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. 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.

Table 6-15 shows the template patterns available for formatting numeric values.

Table 6-15. Template patterns for numeric conversions

Pattern	Description
`9`	value with the specified number of digits
`0`	value with leading zeros
`.` (period)	decimal point
`,` (comma)	group (thousand) separator
`PR`	negative value in angle brackets
`S`	negative value with minus sign (uses locale)
`L`	currency symbol (uses locale)
`D`	decimal point (uses locale)
`G`	group separator (uses locale)
`MI`	minus sign in specified position (if number < 0)
`PL`	plus sign in specified position (if number > 0)
`SG`	plus/minus sign in specified position
`RN`	roman numeral (input between 1 and 3999)
`TH` or `th`	convert to ordinal number
`V`	shift *n* digits (see notes)
`EEEE`	scientific notation (not implemented yet)

Usage notes for the numeric formatting:

- A sign formatted using `SG`, `PL`, or `MI` is not an anchor in 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` specifies 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 decimal 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 6-16 shows some examples of the use of the `to_char` function.

Table 6-16. `to_char` Examples

Input	Output
`to_char(now(),'Day, DD HH12:MI:SS')`	`'Tuesday , 06 05:39:18'`
`to_char(now(),'FMDay, FMDD HH12:MI:SS')`	`'Tuesday, 6 05:39:18'`

Input	Output
to_char(-0.1,'99.99')	' -.10'
to_char(-0.1,'FM9.99')	'-.1'
to_char(0.1,'0.9')	' 0.1'
to_char(12,'9990999.9')	' 0012.0'
to_char(12,'FM9990999.9')	'0012'
to_char(485,'999')	' 485'
to_char(-485,'999')	'-485'
to_char(485,'9 9 9')	' 4 8 5'
to_char(1485,'9,999')	' 1,485'
to_char(1485,'9G999')	' 1 485'
to_char(148.5,'999.999')	' 148.500'
to_char(148.5,'999D999')	' 148,500'
to_char(3148.5,'9G999D999')	' 3 148,500'
to_char(-485,'999S')	'485-'
to_char(-485,'999MI')	'485-'
to_char(485,'999MI')	'485'
to_char(485,'PL999')	'+485'
to_char(485,'SG999')	'+485'
to_char(-485,'SG999')	'-485'
to_char(-485,'9SG99')	'4-85'
to_char(-485,'999PR')	'<485>'
to_char(485,'L999')	'DM 485
to_char(485,'RN')	' CDLXXXV'
to_char(485,'FMRN')	'CDLXXXV'
to_char(5.2,'FMRN')	V
to_char(482,'999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8,'"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12,'99V999')	' 12000'
to_char(12.4,'99V999')	' 12400'
to_char(12.45, '99V9')	' 125'

6.8. Date/Time Functions and Operators

Table 6-18 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 6-17 illustrates the behaviors of the basic arithmetic operators (+, *, etc.). For formatting functions, refer to Section 6.7. You should be familiar with the background information on date/time data types (see Section 5.5).

All the functions and operators described below that take time or timestamp inputs actually come in two variants: one that takes time or timestamp with time zone, and one that takes time or timestamp without time zone. For brevity, these variants are not shown separately.

Table 6-17. Date/Time Operators

Name	Example	Result
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00'
+	time '01:00' + interval '3 hours'	time '04:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00'
-	time '05:00' - interval '2 hours'	time '03:00'
-	interval '2 hours' - time '05:00'	time '03:00:00'
*	interval '1 hour' * int '3'	interval '03:00'
/	interval '1 hour' / int '3'	interval '00:20'

Table 6-18. Date/Time Functions

Name	Return Type	Description	Example	Result
age(timestamp)	interval	Subtract from today	age(timestamp '1957-06-13')	43 years 8 mons 3 days
age(timestamp, timestamp)	interval	Subtract arguments	age('2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
current_date	date	Today's date; see Section 6.8.4		
current_time	time with time zone	Time of day; see Section 6.8.4		
current_timestamp	timestamp with time zone	Date and time; see Section 6.8.4		

Name	Return Type	Description	Example	Result
date_part(text, timestamp)	double precision	Get subfield (equivalent to extract); see also below	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Get subfield (equivalent to extract); see also below	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Truncate to specified precision; see also Section 6.8.2	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00+00
extract(*field* from timestamp)	double precision	Get subfield; see also Section 6.8.1	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(*field* from interval)	double precision	Get subfield; see also Section 6.8.1	extract(month from interval '2 years 3 months')	3
isfinite(timestamp)	boolean	Test for finite time stamp (neither invalid nor infinity)	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	Test for finite interval	isfinite(interval '4 hours')	true
localtime	time	Time of day; see Section 6.8.4		
localtimestamp	timestamp	Date and time; see Section 6.8.4		
now()	timestamp with time zone	Current date and time (equivalent to current_timestamp); see Section 6.8.4		
timeofday()	text	Current date and time; see Section 6.8.4	timeofday()	Wed Feb 21 17:01:13.000126 2001 EST

6.8.1. EXTRACT, date_part

```
EXTRACT (field FROM source)
```

The extract function retrieves subfields from date/time values, such as year or hour. *source* is a value expression that evaluates to type timestamp or interval. (Expressions of type date or time 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 values:

`century`

The year field divided by 100

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

Note that the result for the century field is simply the year field divided by 100, and not the conventional definition which puts most years in the 1900's in the twentieth century.

`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 (0 - 6; Sunday is 0) (for `timestamp` values only)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

`doy`

The day of the year (1 - 365/366) (for `timestamp` values only)

```
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 '2001-02-16 20:38:40');
Result: 982352320

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

`hour`

The hour field (0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

`microseconds`

The seconds field, including fractional parts, multiplied by 1 000 000. Note that this includes full seconds.

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

`millennium`

The year field divided by 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2
```

Note that the result for the millennium field is simply the year field divided by 1000, and not the conventional definition which puts years in the 1900's in the second millennium.

`milliseconds`

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500
```

`minute`

The minutes field (0 - 59)

```
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)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
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 (for `timestamp` values only)

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

`second`

The seconds field, including fractional parts (0 - 59[2])

```
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
```

60 if leap seconds are implemented by the operating system

`timezone_hour`

The hour component of the time zone offset.

`timezone_minute`

The minute component of the time zone offset.

`week`

From a `timestamp` value, calculate 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 week starts on Monday.) In other words, the first Thursday of a year is in week 1 of that year.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

`year`

The year field

```
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 6.7.

The `date_part` function is modeled on the traditional Ingres equivalent to the SQL-standard function `extract`:

```
date_part('field', source)
```

Note that here the *`field`* parameter needs to be a string value, not a name. The valid field values 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
```

6.8.2. `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` (values of type `date` and `time` are cast automatically). *`field`* selects to which precision to truncate the time stamp value. The return value is of type `timestamp` with all fields that are less than the selected one set to zero (or one, for day and month).

Valid values for *`field`* are:

microseconds
milliseconds
second
minute
hour
day
month
year
decade
century
millennium

Examples:

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00+00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00+00
```

6.8.3. `AT TIME ZONE`

The `AT TIME ZONE` construct allows conversions of timestamps to different timezones.

Table 6-19. AT TIME ZONE Variants

Expression	Returns	Description
`timestamp without time zone AT TIME ZONE` *`zone`*	`timestamp with time zone`	Convert local time in given timezone to UTC
`timestamp with time zone AT TIME ZONE` *`zone`*	`timestamp without time zone`	Convert UTC to local time in given timezone
`time with time zone AT TIME ZONE` *`zone`*	`time with time zone`	Convert local time across timezones

In these expressions, the desired time *`zone`* can be specified either as a text string (e.g., `'PST'`) or as an interval (e.g., `INTERVAL '-08:00'`).

Examples (supposing that `TimeZone` 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 zone-less timestamp and interprets it as MST time (GMT-7) to produce a UTC timestamp, which is then rotated to PST (GMT-8) for display. The second example takes a timestamp

specified in EST (GMT-5) and converts it to local time in MST (GMT-7).

The function timezone(*zone*, *timestamp*) is equivalent to the SQL-compliant construct *timestamp* AT TIME ZONE *zone*.

6.8.4. Current Date/Time

The following functions are available to obtain the current date and/or time:

```
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. Without a precision parameter, the result is given to the full available precision.

> **Note:** Prior to PostgreSQL 7.2, the precision parameters were unimplemented, and the result was always given in integer seconds.

Some examples:

```
SELECT CURRENT_TIME;
14:39:53.662522-05

SELECT CURRENT_DATE;
2001-12-23

SELECT CURRENT_TIMESTAMP;
2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
2001-12-23 14:39:53.662522
```

The function now() is the traditional PostgreSQL equivalent to CURRENT_TIMESTAMP.

There is also `timeofday()`, which for historical reasons returns a text string rather than a `timestamp` value:

```
SELECT timeofday();
 Sat Feb 17 19:07:32.000126 2001 EST
```

It is important to realize that `CURRENT_TIMESTAMP` and related functions return the start time of the current transaction; their values do not change during the transaction. `timeofday()` returns the wall clock time and does advance during transactions.

Note: Many other database systems advance these values more frequently.

All the date/time data types also accept the special literal value `now` to specify the current date and time. Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';
```

Note: 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.

6.9. 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 6-20, Table 6-21, and Table 6-22.

Table 6-20. Geometric Operators

Operator	Description	Usage
+	Translation	`box '((0,0),(1,1))' + point '(2.0,0)'`
-	Translation	`box '((0,0),(1,1))' - point '(2.0,0)'`
*	Scaling/rotation	`box '((0,0),(1,1))' * point '(2.0,0)'`

Operator	Description	Usage
/	Scaling/rotation	`box '((0,0),(2,2))' / point '(2.0,0)'`
#	Intersection	`'((1,-1),(-1,1))' # '((1,1),(-1,-1))'`
#	Number of points in path or polygon	`# '((1,0),(0,1),(-1,0))'`
##	Point of closest proximity	`point '(0,0)' ## lseg '((2,0),(0,2))'`
&&	Overlaps?	`box '((0,0),(1,1))' && box '((0,0),(2,2))'`
&<	Overlaps to left?	`box '((0,0),(1,1))' &< box '((0,0),(2,2))'`
&>	Overlaps to right?	`box '((0,0),(3,3))' &> box '((0,0),(2,2))'`
<->	Distance between	`circle '((0,0),1)' <-> circle '((5,0),1)'`
<<	Left of?	`circle '((0,0),1)' << circle '((5,0),1)'`
<^	Is below?	`circle '((0,0),1)' <^ circle '((0,5),1)'`
>>	Is right of?	`circle '((5,0),1)' >> circle '((0,0),1)'`
>^	Is above?	`circle '((0,5),1)' >^ circle '((0,0),1)'`
?#	Intersects or overlaps	`lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'`
?-	Is horizontal?	`point '(1,0)' ?- point '(0,0)'`
?-|	Is perpendicular?	`lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'`
@-@	Length or circumference	`@-@ path '((0,0),(1,0))'`
?|	Is vertical?	`point '(0,1)' ?| point '(0,0)'`
?||	Is parallel?	`lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'`
@	Contained or on	`point '(1,1)' @ circle '((0,0),2)'`
@@	Center of	`@@ circle '((0,0),10)'`
~=	Same as	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'`

Table 6-21. Geometric Functions

Function	Returns	Description	Example
area(object)	double precision	area of item	area(box '((0,0),(1,1))')
box(box, box)	box	intersection box	box(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')
center(object)	point	center of item	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diameter of circle	diameter(circle '((0,0),2.0)')
height(box)	double precision	vertical size of box	height(box '((0,0),(1,1))')
isclosed(path)	boolean	a closed path?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	an open path?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	double precision	length of item	length(path '((-1,0),(1,0))')
npoints(path)	integer	number of points	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	integer	number of points	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	convert path to closed	popen(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	convert path to open path	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	radius of circle	radius(circle '((0,0),2.0)')
width(box)	double precision	horizontal size	width(box '((0,0),(1,1))')

Table 6-22. Geometric Type Conversion Functions

Function	Returns	Description	Example
box(circle)	box	circle to box	box(circle '((0,0),2.0)')

Function	Returns	Description	Example
box(point, point)	box	points to box	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polygon to box	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	to circle	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	point to circle	circle(point '(0,0)', 2.0)
lseg(box)	lseg	box diagonal to lseg	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	points to lseg	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	point	polygon to path	path(polygon '((0,0),(1,1),(2,0))')
point(circle)	point	center	point(circle '((0,0),2.0)')
point(lseg, lseg)	point	intersection	point(lseg '((-1,0),(1,0))', lseg '((-2,-2),(2,2))')
point(polygon)	point	center	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	4-point polygon	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	12-point polygon	polygon(circle '((0,0),2.0)')
polygon(*npts*, circle)	polygon	*npts* polygon	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	path to polygon	polygon(path '((0,0),(1,1),(2,0))')

It is possible to access the two component numbers of a `point` as though it were an array with subscripts 0, 1. For example, if `t.p` is a `point` column then `SELECT p[0] FROM t` retrieves the X coordinate; `UPDATE t SET p[1] = ...` changes the Y coordinate. In the same way, a `box` or an `lseg` may be treated as an array of two `point`s.

6.10. Network Address Type Functions

Table 6-23 shows the operators available for the `inet` and `cidr` types. The operators `<<`, `<<=`, `>>`, `>>=` 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 6-23. `cidr` and `inet` Operators

Operator	Description	Usage
`<`	Less than	`inet '192.168.1.5' < inet '192.168.1.6'`
`<=`	Less than or equal	`inet '192.168.1.5' <= inet '192.168.1.5'`
`=`	Equals	`inet '192.168.1.5' = inet '192.168.1.5'`
`>=`	Greater or equal	`inet '192.168.1.5' >= inet '192.168.1.5'`
`>`	Greater	`inet '192.168.1.5' > inet '192.168.1.4'`
`<>`	Not equal	`inet '192.168.1.5' <> inet '192.168.1.4'`
`<<`	is contained within	`inet '192.168.1.5' << inet '192.168.1/24'`
`<<=`	is contained within or equals	`inet '192.168.1/24' <<= inet '192.168.1/24'`
`>>`	contains	`inet'192.168.1/24' >> inet '192.168.1.5'`
`>>=`	contains or equals	`inet '192.168.1/24' >>= inet '192.168.1/24'`

Table 6-24 shows the functions available for use with the `inet` and `cidr` types. The `host()`, `text()`, and `abbrev()` functions are primarily intended to offer alternative display formats. You can cast a text field to inet using normal casting syntax: `inet(expression)` or `colname::inet`.

Table 6-24. `cidr` and `inet` Functions

Function	Returns	Description	Example	Result
`broadcast(inet)`	`inet`	broadcast address for network	`broadcast('192.168.1.5/24')`	`192.168.1.255/24`
`host(inet)`	`text`	extract IP address as text	`host('192.168.1.5/24')`	`192.168.1.5`
`masklen(inet)`	`integer`	extract netmask length	`masklen('192.168.1.5/24')`	`24`
`set_masklen(inet,integer)`	`inet`	set netmask length for `inet` value	`set_masklen('192.168.1.5/24',16)`	`192.168.1.5/16`

Function	Returns	Description	Example	Result
netmask(inet)	inet	construct netmask for network	netmask('192.168.1.2/24')	255.255.255.0
network(inet)	cidr	extract network part of address	network('192.168.1.2/24')	192.168.1.0/24
text(inet)	text	extract IP address and masklen as text	text(inet '192.168.1.5')	192.168.1.5/32
abbrev(inet)	text	extract abbreviated display as text	abbrev(cidr '10.1.0.0/16')	10.1/16

Table 6-25 shows the functions available for use with the mac type. The function trunc(macaddr) returns a MAC address with the last 3 bytes set to 0. This can be used to associate the remaining prefix with a manufacturer. The directory contrib/mac in the source distribution contains some utilities to create and maintain such an association table.

Table 6-25. macaddr Functions

Function	Returns	Description	Example	Result
trunc(macaddr)	macaddr	set last 3 bytes to zero	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

The macaddr type also supports the standard relational operators (>, <=, etc.) for lexicographical ordering.

6.11. 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 6-26, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

Table 6-26. Sequence Functions

Function	Returns	Description
nextval(text)	bigint	Advance sequence and return new value
currval(text)	bigint	Return value most recently obtained with nextval
setval(text,bigint)	bigint	Set sequence's current value
setval(text,bigint,boolean)	bigint	Set sequence's current value and is_called flag

For largely historical reasons, the sequence to be operated on by a sequence-function call is specified

by a text-string argument. To achieve some compatibility with the handling of ordinary SQL names, the sequence functions convert their argument to lower case unless the string is double-quoted. 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
```

Of course, the text argument can be the result of an expression, not only a simple literal, which is occasionally useful.

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 even if other sessions are executing `nextval` meanwhile.

`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. In the three-parameter form, `is_called` may be set either `true` or `false`. If it's set to `false`, the next `nextval` will return exactly the specified value, and sequence advancement commences with the following `nextval`. 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.

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 may leave unused "holes" in the sequence of assigned values. `setval` operations are never rolled back, either.

If a sequence object has been created with default parameters, `nextval()` calls on it will return successive values beginning with one. Other behaviors can be obtained by using special parameters in the `CREATE SEQUENCE` command; see its command reference page for more information.

6.12. 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.

6.12.1. CASE

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

The SQL `CASE` expression is a generic conditional expression, similar to if/else statements in other languages. `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 *`result`*. 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'
           ELSE 'other'
      END
FROM test;
 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

The data types of all the *result* expressions must be coercible to a single output type. See Section 7.5 for more detail.

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

This "simple" CASE expression is a specialized variant of the general form above. The *expression* is computed and compared to all the *value*s 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
---+-------
 1 | one
 2 | two
 3 | other
```

6.12.2. COALESCE

```
COALESCE(value [, ...])
```

The COALESCE function returns the first of its arguments that is not null. This is often useful to substitute a default value for null values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

6.12.3. NULLIF

```
NULLIF(value1, value2)
```

The NULLIF function returns a null value if and only 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)') ...
```

Tip: COALESCE and NULLIF are just shorthand for CASE expressions. They are actually converted into CASE expressions at a very early stage of processing, and subsequent processing thinks it is dealing with CASE. Thus an incorrect COALESCE or NULLIF usage may draw an error message that refers to CASE.

6.13. Miscellaneous Functions

Table 6-27 shows several functions that extract session and system information.

Table 6-27. Session Information Functions

Name	Return Type	Description
current_database()	name	name of current database
current_schema()	name	name of current schema
current_schemas(boolean)	name[]	names of schemas in search path optionally including implicit schemas
current_user	name	user name of current execution context
session_user	name	session user name
user	name	equivalent to current_user
version()	text	PostgreSQL version information

The session_user is the user that initiated a database connection; it is fixed for the duration of that connection. The current_user is the user identifier that is applicable for permission checking. Normally, it is equal to the session user, but it 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.

The search path may be altered by a run-time setting. The command to use is SET SEARCH_PATH 'schema'[,'schema']...

version() returns a string describing the PostgreSQL server's version.

Table 6-28 shows the functions available to query and alter run-time configuration parameters.

Table 6-28. Configuration Settings Information Functions

Name	Return Type	Description
current_setting(*setting_name*)	text	value of current setting
set_config(*setting_name*, *new_value*, *is_local*)	text	new value of current setting

The `current_setting` is used to obtain the current value of the *`setting_name`* setting, as a query result. It is the equivalent to the SQL `SHOW` command. For example:

```
select current_setting('DateStyle');
            current_setting
-------------------------------------
 ISO with US (NonEuropean) conventions
(1 row)
```

`set_config` allows the *`setting_name`* setting to be changed to *`new_value`*. If *`is_local`* is set to `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. It is the equivalent to the SQL `SET` command. For example:

```
select set_config('show_statement_stats','off','f');
 set_config
------------
 off
(1 row)
```

Table 6-29 lists functions that allow the user to query object access privileges programmatically. See Section 2.7 for more information about privileges.

Table 6-29. Access Privilege Inquiry Functions

Name	Return Type	Description
has_table_privilege(*user*, *table*, *access*)	boolean	does user have access to table
has_table_privilege(*table*, *access*)	boolean	does current user have access to table
has_database_privilege(*user*, *database*, *access*)	boolean	does user have access to database
has_database_privilege(*database*, *access*)	boolean	does current user have access to database

Name	Return Type	Description
has_function_privilege(*user*, *function*, *access*)	boolean	does user have access to function
has_function_privilege(*function*, *access*)	boolean	does current user have access to function
has_language_privilege(*user*, *language*, *access*)	boolean	does user have access to language
has_language_privilege(*language*, *access*)	boolean	does current user have access to language
has_schema_privilege(*user*, *schema*, *access*)	boolean	does user have access to schema
has_schema_privilege(*schema*, *access*)	boolean	does current user have access to schema

has_table_privilege checks whether a user can access a table in a particular way. The user can be specified by name or by ID (pg_user.usesysid), 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 type is specified by a text string, which must evaluate to one of the values SELECT, INSERT, UPDATE, DELETE, RULE, REFERENCES, or TRIGGER. (Case of the string is not significant, however.) An example is:

```
SELECT has_table_privilege('myschema.mytable', 'select');
```

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 type must evaluate to CREATE, 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. The desired access type must currently evaluate to 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 type must currently 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 type must evaluate to CREATE or USAGE.

Table 6-30 shows functions that determine whether a certain object is *visible* in the current schema search path. 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. For example, to list the names of all visible tables:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Table 6-30. Schema Visibility Inquiry Functions

Name	Return Type	Description
`pg_table_is_visible(`*`tableOID`*`)`	`boolean`	is table visible in search path
`pg_type_is_visible(`*`typeOID`*`)`	`boolean`	is type visible in search path
`pg_function_is_visible(`*`functionOID`*`)`	`boolean`	is function visible in search path
`pg_operator_is_visible(`*`operatorOID`*`)`	`boolean`	is operator visible in search path
`pg_opclass_is_visible(`*`opclassOID`*`)`	`boolean`	is operator class visible in search path

`pg_table_is_visible` performs the check for tables (or views, or any other kind of `pg_class` entry). `pg_type_is_visible`, `pg_function_is_visible`, `pg_operator_is_visible`, and `pg_opclass_is_visible` perform the same sort of visibility check for types, functions, operators, and operator classes, respectively. 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`, or `regoperator`), for example

```
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 6-31 lists functions that extract information from the system catalogs. `pg_get_viewdef()`, `pg_get_ruledef()`, `pg_get_indexdef()`, and `pg_get_constraintdef()` respectively reconstruct the creating command for a view, rule, index, or constraint. (Note that this is a decompiled reconstruction, not the verbatim text of the command.) At present `pg_get_constraintdef()` only works for foreign-key constraints. `pg_get_userbyid()` extracts a user's name given a `usesysid` value.

Table 6-31. Catalog Information Functions

Name	Return Type	Description
`pg_get_viewdef(`*`viewname`*`)`	`text`	Get `CREATE VIEW` command for view (*deprecated*)
`pg_get_viewdef(`*`viewOID`*`)`	`text`	Get `CREATE VIEW` command for view
`pg_get_ruledef(`*`ruleOID`*`)`	`text`	Get `CREATE RULE` command for rule
`pg_get_indexdef(`*`indexOID`*`)`	`text`	Get `CREATE INDEX` command for index

Name	Return Type	Description
pg_get_constraintdef(*constraintOID*)	text	Get definition of a constraint
pg_get_userbyid(*userid*)	name	Get user name with given ID

The function shown in Table 6-32 extract comments previously stored with the COMMENT command. A null value is returned if no comment can be found matching the specified parameters.

Table 6-32. Comment Information Functions

Name	Return Type	Description
obj_description(*objectOID*, *tablename*)	text	Get comment for a database object
obj_description(*objectOID*)	text	Get comment for a database object (*deprecated*)
col_description(*tableOID*, *columnnumber*)	text	Get comment for a table column

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.

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.

6.14. Aggregate Functions

Aggregate functions compute a single result value from a set of input values. Table 6-33 show the built-in aggregate functions. The special syntax considerations for aggregate functions are explained in Section 1.2.5. Consult the *PostgreSQL Tutorial* for additional introductory information.

Table 6-33. Aggregate Functions

Function	Argument Type	Return Type	Description	

Function	Argument Type	Return Type	Description	
avg(*expression*)	smallint, integer, bigint, real, double precision, numeric, or interval.	numeric for any integer type argument, double precision for a floating-point argument, otherwise the same as the argument data type	the average (arithmetic mean) of all input values	
count(*)		bigint	number of input values	
count(*expression*)	any	bigint	number of input values for which the value of *expression* is not null	
max(*expression*)	any numeric, string, or date/time type	same as argument type	maximum value of *expression* across all input values	
min(*expression*)	any numeric, string, or date/time type	same as argument type	minimum value of *expression* across all input values	
stddev(*expression*)	smallint, integer, bigint, real, double precision, or numeric.	double precision for floating-point arguments, otherwise numeric.	sample standard deviation of the input values	
sum(*expression*)	smallint, integer, bigint, real, double precision, numeric, or interval	bigint for smallint or integer arguments, numeric for bigint arguments, double precision for floating-point arguments, otherwise the same as the argument data type	sum of *expression* across all input values	

Function	Argument Type	Return Type	Description	
vari-ance(*expression*)	smallint, integer, bigint, real, double precision, or numeric.	double precision for floating-point arguments, otherwise numeric.	sample variance of the input values (square of the sample standard deviation)	

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 function coalesce may be used to substitute zero for null when necessary.

6.15. 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.

6.15.1. 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 may 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:

```
SELECT col1 FROM tab1
    WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

6.15.2. IN (scalar form)

```
expression IN (value[, ...])
```

The right-hand side of this form of IN 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.

> **Note:** This form of IN is not truly a subquery expression, but it seems best to document it in the same place as subquery IN.

6.15.3. IN (subquery form)

```
expression IN (subquery)
```

The right-hand side of this form of IN 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.

```
(expression [, expression ...]) IN (subquery)
```

The right-hand side of this form of IN is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. 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 expressions or subquery 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 row results are either unequal or null, with at least one null, then the result of IN is null.

6.15.4. NOT IN (scalar form)

```
expression NOT IN (value[, ...])
```

The right-hand side of this form of `NOT IN` 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

```
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.

6.15.5. NOT IN (subquery form)

```
expression NOT IN (subquery)
```

The right-hand side of this form of `NOT IN` 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.

```
(expression [, expression ...]) NOT IN (subquery)
```

The right-hand side of this form of `NOT IN` is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise 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.

As usual, null values in the expressions or subquery 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 row results are either unequal or null, with at least one null, then the result of NOT IN is null.

6.15.6. ANY/SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

The right-hand side of this form of ANY 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 *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 subquery returns no rows).

SOME is a synonym for ANY. IN is equivalent to = 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 ANY 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.

```
(expression [, expression ...]) operator ANY (subquery)
(expression [, expression ...]) operator SOME (subquery)
```

The right-hand side of this form of ANY is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. Presently, only = and <> operators are allowed in row-wise ANY queries. The result of ANY is "true" if any equal or unequal row is found, respectively. The result is "false" if no such row is found (including the special case where the subquery returns no rows).

As usual, null values in the expressions or subquery 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 there is at least one null row result, then the result of ANY cannot be false; it will be true or null.

6.15.7. ALL

```
expression operator ALL (subquery)
```

The right-hand side of this form of ALL 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 *operator*, which must yield a Boolean result. The result of 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.

NOT IN is equivalent to <> ALL.

Note that if there are no failures but at least one right-hand row yields null for the operator's result, the result of the ALL 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.

```
(expression [, expression ...]) operator ALL (subquery)
```

The right-hand side of this form of ALL is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand list. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. Presently, only = and <> operators are allowed in row-wise ALL queries. The result of ALL is "true" if all subquery rows are equal or unequal, respectively (including the special case where the subquery returns no rows). The result is "false" if any row is found to be unequal or equal, respectively.

As usual, null values in the expressions or subquery 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 there is at least one null row result, then the result of ALL cannot be true; it will be false or null.

6.15.8. Row-wise Comparison

```
(expression [, expression ...]) operator (subquery)
(expression [, expression ...]) operator (expression [, expression ...])
```

The left-hand side is a list of scalar expressions. The right-hand side can be either a list of scalar expressions of the same length, or a parenthesized subquery, which must return exactly as many columns as there are expressions on the left-hand side. 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, or to the right-hand expression list. Presently, only = and <> operators are allowed in row-wise comparisons. The result is "true" if the two rows are equal or unequal, respectively.

As usual, null values in the expressions or subquery 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 the row comparison is unknown (null).

Chapter 7. Type Conversion

SQL queries 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 a user or programmer using *explicit* type coercion.

This chapter introduces the PostgreSQL type conversion mechanisms and conventions. Refer to the relevant sections in Chapter 5 and Chapter 6 for more information on specific data types and allowed functions and operators.

The *PostgreSQL Programmer's Guide* has more details on the exact algorithms used for implicit type conversion and coercion.

7.1. 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 should be governed by general rules rather than by *ad hoc* heuristics, to allow mixed-type expressions to be meaningful even with user-defined types.

The PostgreSQL scanner/parser decodes lexical elements into only five fundamental categories: integers, floating-point numbers, strings, names, and key words. Most extended types are first tokenized into 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

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
 Label  | Value
--------+-------
 Origin | (0,0)
(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:

Operators

PostgreSQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators.

Function calls

Much of the PostgreSQL type system is built around a rich set of functions. Function calls have one or more arguments which, for any specific query, must be matched to the functions available in the system catalog. 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.

Query targets

SQL `INSERT` and `UPDATE` statements place the results of expressions into a table. The expressions in the query must be matched up with, and perhaps converted to, the types of the target columns.

`UNION` and `CASE` constructs

Since all select 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 coerced to a common type so that the `CASE` expression as a whole has a known output type.

Many of the general type conversion rules use simple conventions built on the PostgreSQL function and operator system tables. There are some heuristics included in the conversion rules to better support conventions for the SQL standard native types such as `smallint`, `integer`, and `real`.

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 should not be 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 a *preferred type* which is 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 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 in the catalog.
- 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 functions into the query.

  Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines an explicit 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.

7.2. Operators

The operand types of an operator invocation are resolved following the procedure below. Note that this procedure is indirectly affected by the precedence of the involved operators. See Section 1.1.6 for more information.

Operand Type Resolution

1. Select the operators to be considered from the `pg_operator` system catalog. If an unqualified operator name is 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 2.8.3). 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 is `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 coerced (using an implicit coercion function) to match. `unknown` literals are assumed to be coercible 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. 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 with the most exact or binary-compatible matches on input types. Keep all candidates if none have any exact or binary-compatible matches. If only one candidate remains, use it; else continue to the next step.

    d. Run through all candidates and keep those that accept preferred types at the most positions where type coercion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

    e. 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. Also note whether any of the candidates accept a preferred data type within the selected category. Now discard operator 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.

f. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Examples

Example 7-1. Exponentiation Operator Type Resolution

There is only one exponentiation operator defined in the catalog, and it takes arguments of type `double precision`. The scanner assigns an initial type of `integer` to both arguments of this query expression:

```
tgl=> SELECT 2 ^ 3 AS "Exp";
 Exp
-----
   8
(1 row)
```

So the parser does a type conversion on both operands and the query is equivalent to

```
tgl=> SELECT CAST(2 AS double precision) ^ CAST(3 AS double precision) AS "Exp";
 Exp
-----
   8
(1 row)
```

or

```
tgl=> SELECT 2.0 ^ 3.0 AS "Exp";
 Exp
-----
   8
(1 row)
```

Note: This last form has the least overhead, since no functions are called to do implicit type conversion. This is not an issue for small queries, but may have an impact on the performance of queries involving large tables.

Example 7-2. String Concatenation Operator Type Resolution

A string-like syntax is used for working with string types as well as for working with complex extended types. Strings with unspecified type are matched with likely operator candidates.

An example with one unspecified argument:

```
tgl=> SELECT text 'abc' || 'def' AS "Text and Unknown";
 Text and Unknown
------------------
 abcdef
(1 row)
```

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`.

Concatenation on unspecified types:

```
tgl=> SELECT 'abc' || 'def' AS "Unspecified";
 Unspecified
-------------
 abcdef
(1 row)
```

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 7-3. Absolute-Value and Factorial 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 a non-numeric input:

```
tgl=> select @ text '-4.5' as "abs";
 abs
-----
 4.5
(1 row)
```

Here the system has performed an implicit text-to-float8 conversion before applying the chosen operator. We can verify that float8 and not some other type was used:

```
tgl=> select @ text '-4.5e500' as "abs";
ERROR:  Input '-4.5e500' is out of range for float8
```

On the other hand, the postfix operator `!` (factorial) is defined only for integer data types, not for float8. So, if we try a similar case with `!`, we get:

```
tgl=> select text '20' ! as "factorial";
ERROR:  Unable to identify a postfix operator '!' for type 'text'
        You may need to add parentheses or an explicit cast
```

This happens because the system can't decide which of the several possible `!` operators should be preferred. We can help it out with an explicit cast:

```
tgl=> select cast(text '20' as int8) ! as "factorial";
      factorial
----------------------
 2432902008176640000
(1 row)
```

7.3. Functions

The argument types of function calls are resolved according to the following steps.

Function Argument Type Resolution

1. Select the functions to be considered from the `pg_proc` system catalog. If an unqualified function name is used, the functions considered are those of the right name and argument count that are visible in the current search path (see Section 2.8.3). 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 trivial type coercion 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. When these conditions are met, the function argument is coerced to the named data type without any explicit function call.

4. Look for the best match.

    a. Discard candidate functions for which the input types do not match and cannot be coerced (using an implicit coercion function) to match. `unknown` literals are assumed to be coercible 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. 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 with the most exact or binary-compatible matches on input types. Keep all candidates if none have any exact or binary-compatible matches. If only one candidate remains, use it; else continue to the next step.

    d. Run through all candidates and keep those that accept preferred types at the most positions where type coercion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

    e. 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. Also note whether any of the candidates accept a preferred data type within the selected category. 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.

f. If only one candidate remains, use it. If no candidate or more than one candidate remains, then fail.

Examples

Example 7-4. Factorial Function Argument Type Resolution

There is only one `int4fac` function defined in the `pg_proc` catalog. So the following query automatically converts the `int2` argument to `int4`:

```
tgl=> SELECT int4fac(int2 '4');
 int4fac
---------
      24
(1 row)
```

and is actually transformed by the parser to

```
tgl=> SELECT int4fac(int4(int2 '4'));
 int4fac
---------
      24
(1 row)
```

Example 7-5. Substring Function Type Resolution

There are two `substr` functions declared in `pg_proc`. However, only one takes two arguments, of types `text` and `int4`.

If called with a string constant of unspecified type, the type is matched up directly with the only candidate function type:

```
tgl=> 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 coerce it to become `text`:

```
tgl=> SELECT substr(varchar '1234', 3);
 substr
--------
     34
(1 row)
```

which is transformed by the parser to become

```
tgl=> SELECT substr(text(varchar '1234'), 3);
 substr
--------
     34
(1 row)
```

Note: Actually, the parser is aware 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 explicit type conversion call is really inserted in this case.

And, if the function is called with an `int4`, the parser will try to convert that to `text`:

```
tgl=> SELECT substr(1234, 3);
 substr
--------
     34
(1 row)
```

which actually executes as

```
tgl=> SELECT substr(text(1234), 3);
 substr
--------
     34
(1 row)
```

This succeeds because there is a conversion function text(int4) in the system catalog.

7.4. Query Targets

Values to be inserted into a table are coerced to the destination column's data type according to the following steps.

Query Target Type Resolution

1. Check for an exact match with the target.
2. Otherwise, try to coerce the expression to the target type. This will succeed if the two types are known binary-compatible, or if there is a conversion function. 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. If the target is a fixed-length type (e.g. `char` or `varchar` declared with a length) then try to find a sizing function for the target type. A sizing function is a function of the same name as the type, taking two arguments of which the first is that type and the second is an integer, and returning the same type. If one is found, it is applied, passing the column's declared length as the second parameter.

Example 7-6. `character` Storage Type Conversion

For a target column declared as `character(20)` the following query ensures that the target is sized correctly:

```
tgl=> CREATE TABLE vv (v character(20));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
```

```
tgl=> SELECT v, length(v) FROM vv;
          v           | length
----------------------+--------
 abcdef               |     20
(1 row)
```

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 coerced to `bpchar` ("blank-padded char", the internal name of the character data type) to match the target column type. (Since the parser knows that `text` and `bpchar` are binary-compatible, this coercion is implicit and does not insert any real function call.) Finally, the sizing function `bpchar(bpchar, integer)` is found in the system catalogs 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.

7.5. `UNION` and `CASE` 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`. A `CASE` construct also uses the identical algorithm to match up its component expressions and select a result data type.

`UNION` and `CASE` Type Resolution

1. If all inputs are of type `unknown`, resolve as type `text` (the preferred type for string category). Otherwise, ignore the `unknown` inputs while choosing the type.
2. If the non-unknown inputs are not all of the same type category, fail.
3. Choose the first non-unknown input type which is a preferred type in that category or allows all the non-unknown inputs to be implicitly coerced to it.
4. Coerce all inputs to the selected type.

Examples

Example 7-7. Underspecified Types in a Union

```
tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
 Text
------
 a
 b
(2 rows)
```

Here, the unknown-type literal `'b'` will be resolved as type text.

Example 7-8. Type Conversion in a Simple Union

```
tgl=> SELECT 1.2 AS "Numeric" UNION SELECT 1;
 Numeric
---------
       1
     1.2
(2 rows)
```

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 7-9. Type Conversion in a Transposed Union

```
tgl=> SELECT 1 AS "Real"
tgl-> 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 8. 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.

8.1. Introduction

The classical example for the need of an index is if there is a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

and the application requires a lot of queries of the form

```
SELECT content FROM test1 WHERE id = constant;
```

Ordinarily, 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 (possibly zero or one) returned by the query, then this is clearly an inefficient method. If the system were instructed to maintain an index on the `id` column, then it could 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, and would not have to read the entire book to find the interesting location. As it is the task of the author to anticipate the items that the readers are most likely to look up, it is the task of the database programmer to foresee which indexes would be of advantage.

The following command would be used to create the index on the `id` column, as discussed:

```
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 the index is created, no further intervention is required: the system will use the index when it thinks it would be more efficient than a sequential table scan. But you may have to run the `ANALYZE` command regularly to update statistics to allow the query planner to make educated decisions. Also read Chapter 10 for information about how to find out whether an index is used and when and why the planner may choose to *not* use an index.

Indexes can benefit `UPDATE`s and `DELETE`s with search conditions. Indexes can also be used in join queries. Thus, an index defined on a column that is part of a join condition can significantly speed up queries with joins.

When 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 non-essential or do not get used at all should be removed. Note that a query or data manipulation command can use at most one index per table.

8.2. Index Types

PostgreSQL provides several index types: B-tree, R-tree, GiST, and Hash. Each index type is more appropriate for a particular query type because of the algorithm it uses. By default, the `CREATE INDEX` command will create a B-tree index, which fits the most common situations. In particular, the PostgreSQL query optimizer will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators: <, <=, =, >=, >

R-tree indexes are especially suited for spatial data. To create an R-tree index, use a command of the form

```
CREATE INDEX name ON table USING RTREE (column);
```

The PostgreSQL query optimizer will consider using an R-tree index whenever an indexed column is involved in a comparison using one of these operators: <<, &<, &>, >>, @, ~=, && (Refer to Section 6.9 about the meaning of these operators.)

The query optimizer will consider using a hash index whenever an indexed column is involved in a comparison using the = operator. 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 be similar or slower than B-tree indexes, and the index size and build time for hash indexes is much worse. Hash indexes also suffer poor performance under high concurrency. For these reasons, hash index use is discouraged.

The B-tree index is an implementation of Lehman-Yao high-concurrency B-trees. The R-tree index method implements standard R-trees using Guttman's quadratic split algorithm. The hash index is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

8.3. Multicolumn Indexes

An index can be defined on more than one column. For example, if you have a table of this form:

```
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 may be appropriate to define an index on the columns `major` and `minor` together, e.g.,

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Currently, only the B-tree and GiST implementations support multicolumn indexes. Up to 32 columns may be specified. (This limit can be altered when building PostgreSQL; see the file `pg_config.h`.)

The query optimizer can use a multicolumn index for queries that involve the first *n* consecutive columns in the index (when used with appropriate operators), up to the total number of columns specified in the index definition. For example, an index on `(a, b, c)` can be used in queries involving all of `a`, `b`, and `c`, or in queries involving both `a` and `b`, or in queries involving only `a`, but not in other combinations. (In a query involving `a` and `c` the optimizer might choose to use the index for `a` only and treat `c` like an ordinary unindexed column.)

Multicolumn indexes can only be used if the clauses involving the indexed columns are joined with `AND`. For instance,

```
SELECT name FROM test2 WHERE major = constant OR minor = constant;
```

cannot make use of the index `test2_mm_idx` defined above to look up both columns. (It can be used to look up only the `major` column, however.)

Multicolumn indexes should be used sparingly. Most of the time, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are almost certainly inappropriate.

8.4. Unique Indexes

Indexes may also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.

```
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.

PostgreSQL automatically creates unique indexes when a table is declared with a unique constraint or a primary key, on the columns that make up the primary key or unique columns (a multicolumn index, if appropriate), to enforce that constraint. A unique index can be added to a table at any later time, to add a unique 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.

8.5. Functional Indexes

For a *functional index*, an index is defined on the result of a function applied to one or more columns of a single table. Functional indexes can be used to obtain fast access to data based on the result of function calls.

For example, a common way to do case-insensitive comparisons is to use the `lower` function:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

This query can use an index, if one has been defined on the result of the `lower(column)` operation:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

The function in the index definition can take more than one argument, but they must be table columns, not constants. Functional indexes are always single-column (namely, the function result) even if the function uses more than one input field; there cannot be multicolumn indexes that contain function calls.

> **Tip:** The restrictions mentioned in the previous paragraph can easily be worked around by defining a custom function to use in the index definition that computes any desired result internally.

8.6. Operator Classes

An index definition may specify an *operator class* for each column of an index.

```
CREATE INDEX name ON table (column opclass [, ...]);
```

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. There are also some operator classes with special purposes:

- The operator classes `box_ops` and `bigbox_ops` both support R-tree indexes on the `box` data type. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating-point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. If the field on which your rectangles lie is about 20 000 units square or larger, you should use `bigbox_ops`.

The following query shows all defined operator classes:

```
SELECT am.amname AS acc_method,
```

```
    opc.opcname AS ops_name
FROM pg_am am, pg_opclass opc
WHERE opc.opcamid = am.oid
ORDER BY acc_method, ops_name;
```

It can be extended to show all the operators included in each class:

```
SELECT am.amname AS acc_method,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    FROM pg_am am, pg_opclass opc, pg_amop amop, pg_operator opr
    WHERE opc.opcamid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
    ORDER BY acc_method, ops_name, ops_comp;
```

8.7. 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.

A major motivation for partial indexes 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 8-1 shows a possible application of this idea.

Example 8-1. 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 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:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

To create a partial index that suits our example, use a command such as this:

```
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:

```
SELECT * FROM access_log WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

A query that cannot use this index is:

```
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.

Another possibility is to exclude values from the index that the typical query workload is not interested in; this is shown in Example 8-2. 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 8-2. 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 may be among the billed or among the unbilled orders.

Example 8-2 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 query's WHERE condition mathematically *implies* the index's predicate. PostgreSQL does not have a sophisticated theorem prover that can recognize mathematically equivalent predicates 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 the query's WHERE condition or the index will not be recognized to be usable.

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 8-3. This enforces uniqueness among the rows that satisfy the index predicate, without constraining those that do not.

Example 8-3. 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:

```
CREATE TABLE tests (subject text,
                    target text,
                    success bool,
                    ...);
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 trials and many unsuccessful ones.

Finally, a partial index can also be used to override the system's query plan choices. It may occur that data sets with peculiar distributions will 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*.

8.8. 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 is done with the `EXPLAIN` command; its application for this purpose is illustrated in Section 10.1.

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 proportionally reduced 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 (described in the *PostgreSQL Administrator's Guide*). 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 problem for 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 of the plan nodes can be tuned with run-time parameters (described in the *PostgreSQL Administrator's Guide*). An inaccurate selectivity estimate is due to insufficient statistics. It may be possible to help this by tuning the statistics-gathering parameters (see `ALTER TABLE` reference).

  If you do not succeed in adjusting the costs to be more appropriate, then you may have to resort to forcing index usage explicitly. You may also want to contact the PostgreSQL developers to examine the issue.

Chapter 9. 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.

9.1. Introduction

Unlike traditional database systems which use locks for concurrency control, PostgreSQL maintains data consistency 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.

The main difference between multiversion and lock models is that in MVCC locks acquired for querying (reading) data don't 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.

9.2. 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 9-1.

Table 9-1. SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

PostgreSQL offers the read committed and serializable isolation levels.

9.2.1. 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`s 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`, and `SELECT FOR UPDATE` 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 query start time. However, such a target row may have already been updated (or deleted or marked for update) by another concurrent transaction by the time it is found. In this case, the would-be updater 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 query search condition (`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.

Because of the above rule, it is possible for updating queries to see inconsistent snapshots --- they can see the effects of concurrent updating queries that affected the same rows they are trying to update, but they do not see effects of those queries on other rows in the database. This behavior makes Read Committed mode unsuitable for queries 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;
```

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 query 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 query starts with a new snapshot that includes all transactions committed up to that instant, subsequent queries 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* query 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 may be necessary to guarantee a more rigorously consistent view of the database than the Read Committed mode provides.

9.2.2. Serializable Isolation 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`s within a single transaction always see the same data.

`UPDATE`, `DELETE`, and `SELECT FOR UPDATE` 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 may have already been updated (or deleted or marked for update) 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 selected it for update) then the serializable transaction will be rolled back with the message

```
ERROR:  Can't serialize access due to concurrent update
```

because a serializable transaction cannot modify 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 may 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 may be significant, this mode is recommended only when updating transactions contain logic sufficiently complex that they may give wrong answers in Read Committed mode. Most commonly, Serializable mode is necessary when a transaction performs several successive queries that must see identical views of the database.

9.3. 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 be executed concurrently with other operations on the same table.)

9.3.1. Table-Level Locks

The list below shows the available lock modes and the contexts in which they are used automatically by PostgreSQL. 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. 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 may acquire `ACCESS EXCLUSIVE` lock and later acquire `ACCESS SHARE` lock on the same table.) Non-conflicting lock modes may be held concurrently by many transactions. Notice in particular that some lock modes are self-conflicting (for example, `ACCESS EXCLUSIVE` cannot be held by more than one transaction at a time) while others are not self-conflicting (for example, `ACCESS SHARE` can be held by multiple transactions). Once acquired, a lock mode is held till end of transaction.

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 the *PostgreSQL Administrator's Guide*.

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` command acquires 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`).

`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 query 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).

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.

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, 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 by any PostgreSQL command.

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, and VACUUM FULL commands. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

Note: Only an ACCESS EXCLUSIVE lock blocks a SELECT (without FOR UPDATE) statement.

9.3.2. Row-Level Locks

In addition to table-level locks, there are row-level locks. A row-level lock on a specific row is automatically acquired when the row is updated (or deleted or marked for update). The lock is held until the transaction commits or rolls back. Row-level locks don't affect data querying; they block *writers to the same row* only. To acquire a row-level lock on a row without actually modifying the row, select the row with SELECT FOR UPDATE. Note that once a particular row-level lock is acquired, the transaction may update the row multiple times without fear of conflicts.

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 may cause a disk write; thus, for example, SELECT FOR UPDATE will modify selected rows to mark them 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 tuple is fetched or updated. Application writers normally need not be concerned with page-level locks, but we mention them for completeness.

9.3.3. Deadlocks

Use of explicit locking can cause *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.)

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. 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 may 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).

9.4. Data Consistency Checks at the Application Level

Because readers in PostgreSQL don't 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 may very well see different snapshots. So the whole concept of "now" is somewhat suspect 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 may ensue.

To ensure the current validity of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE` or an appropriate `LOCK TABLE` statement. (`SELECT FOR UPDATE` 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.

> **Note:** Before version 6.5 PostgreSQL used read locks, and so the above consideration is also the case when upgrading from PostgreSQL versions prior to 6.5.

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 locks to prevent concurrent changes, one should use Read Committed mode, or in Serializable mode be careful to obtain the lock(s) before performing queries. An explicit lock obtained in 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 may predate some now-committed changes in the table. A serializable transaction's snapshot is actually frozen at the start of its first query (`SELECT`, `INSERT`, `UPDATE`, or `DELETE`), so it's possible to obtain explicit locks before the snapshot is frozen.

9.5. 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 indexes

> Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index tuple is fetched or inserted. B-tree indexes provide the highest concurrency without deadlock conditions.

GiST and R-tree indexes

> Share/exclusive index-level locks are used for read/write access. Locks are released after the statement (command) is done.

Hash indexes

> Share/exclusive page-level locks are used for read/write access. Locks are released after the page is processed. Page-level locks provide better concurrency than index-level ones but are liable to deadlocks.

In short, B-tree indexes are the recommended index type for concurrent applications.

Chapter 10. 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.

10.1. 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. You can use the `EXPLAIN` command to see what query plan the system 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 numbers that are currently 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 are retrieved, which they may not be --- a query with a `LIMIT` clause will stop short of paying the total cost, for example.)
- 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 units of disk page fetches. (CPU effort estimates are converted into disk-page units using some fairly arbitrary fudge factors. If you want to experiment with these factors, see the list of run-time configuration parameters in the *PostgreSQL Administrator's Guide*.)

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/optimizer cares about. In particular, the cost does not consider the time spent transmitting result rows to the frontend --- which could be a pretty dominant 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/scanned by the query --- it is usually less, reflecting the estimated selectivity of any `WHERE`-clause constraints that are being applied at this node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Here are some examples (using the regress test database after a `VACUUM ANALYZE`, and 7.3 development sources):

```
regression=# EXPLAIN SELECT * FROM tenk1;
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

This is about as straightforward as it gets. If you do

```
SELECT * FROM pg_class WHERE relname = 'tenk1';
```

you will find out that `tenk1` has 233 disk pages and 10000 rows. So the cost is estimated at 233 page reads, defined as costing 1.0 apiece, plus 10000 * `cpu_tuple_cost` which is currently 0.01 (try `SHOW cpu_tuple_cost`).

Now let's modify the query to add a `WHERE` condition:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..358.00 rows=1033 width=148)
   Filter: (unique1 < 1000)
```

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 to reflect the extra CPU time spent checking the `WHERE` condition.

The actual number of rows this query would select is 1000, but the 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.

Modify the query to restrict the condition even more:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50;
                                  QUERY PLAN
------------------------------------------------------------------------------
-----
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.33 rows=49 width=148)
   Index Cond: (unique1 < 50)
```

and you will see that if we make the `WHERE` condition selective enough, the planner will eventually decide that an index scan is cheaper than a sequential scan. This plan will only have to visit 50 rows because of the index, so it wins despite the fact that each individual fetch is more expensive than reading a whole disk page sequentially.

Add another clause to the `WHERE` condition:

```
regression=# EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 50 AND
regression-# stringu1 = 'xxx';
                                  QUERY PLAN
------------------------------------------------------------------------------
-----
 Index Scan using tenk1_unique1 on tenk1  (cost=0.00..179.45 rows=1 width=148)
   Index Cond: (unique1 < 50)
   Filter: (stringu1 = 'xxx'::name)
```

The added clause `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.

Let's try joining two tables, using the fields we have been discussing:

```
    regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50
    regression-# AND t1.unique2 = t2.unique2;
                                  QUERY PLAN
    ------------------------------------------------------------------------
--
     Nested Loop  (cost=0.00..327.02 rows=49 width=296)
       ->  Index Scan using tenk1_unique1 on tenk1 t1
                                       (cost=0.00..179.33 rows=49 width=148)
             Index Cond: (unique1 < 50)
       ->  Index Scan using tenk2_unique2 on tenk2 t2
                                       (cost=0.00..3.01 rows=1 width=148)
             Index Cond: ("outer".unique2 = t2.unique2)
```

In this nested-loop join, the outer scan is the same index scan we had in the example before last, and so its cost and row count are the same because we are applying the `unique1 < 50` `WHERE` clause 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 (49 * 3.01, here), plus a little CPU time for join processing.

In this example the loop'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 relations 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 for each plan type. (This is a crude tool, but useful. See also Section 10.3.)

```
    regression=# SET enable_nestloop = off;
    SET
    regression=# EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 50
    regression-# AND t1.unique2 = t2.unique2;
                                  QUERY PLAN
    ------------------------------------------------------------------------

     Hash Join  (cost=179.45..563.06 rows=49 width=296)
       Hash Cond: ("outer".unique2 = "inner".unique2)
       ->  Seq Scan on tenk2 t2  (cost=0.00..333.00 rows=10000 width=148)
       ->  Hash  (cost=179.33..179.33 rows=49 width=148)
             ->  Index Scan using tenk1_unique1 on tenk1 t1
                                      (cost=0.00..179.33 rows=49 width=148)
                   Index Cond: (unique1 < 50)
```

This plan proposes to extract the 50 interesting rows of `tenk1` using ye same olde 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 179.33; 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:

```
    regression=# EXPLAIN ANALYZE
    regression-# SELECT * FROM tenk1 t1, tenk2 t2
    regression-# WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;
                                       QUERY PLAN
    ------------------------------------------------------------------------
-----
     Nested Loop  (cost=0.00..327.02 rows=49 width=296)
                                       (actual time=1.18..29.82 rows=50 loops=1)
       ->  Index Scan using tenk1_unique1 on tenk1 t1
                     (cost=0.00..179.33 rows=49 width=148)
                                       (actual time=0.63..8.91 rows=50 loops=1)
             Index Cond: (unique1 < 50)
       ->  Index Scan using tenk2_unique2 on tenk2 t2
                     (cost=0.00..3.01 rows=1 width=148)
                                       (actual time=0.29..0.32 rows=1 loops=50)
             Index Cond: ("outer".unique2 = t2.unique2)
     Total runtime: 31.60 msec
```

Note that the "actual time" values are in milliseconds of real time, whereas the "cost" estimates are expressed in arbitrary units of disk fetches; so they are unlikely to match up. The thing to pay attention to is the ratios.

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 may 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 making the changes.

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 can't be assumed to apply to large tables. The planner's cost estimates are not linear and so it may well 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.

10.2. 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 `pg_class`'s `reltuples` and `relpages` columns. We can look at it with queries similar to this one:

```
regression=# SELECT relname, relkind, reltuples, relpages FROM pg_class
regression-# WHERE relname LIKE 'tenk1%';
    relname    | relkind | reltuples | relpages
---------------+---------+-----------+----------
 tenk1         | r       |     10000 |      233
 tenk1_hundred | i       |     10000 |       30
 tenk1_unique1 | i       |     10000 |       30
 tenk1_unique2 | i       |     10000 |       30
(4 rows)
```

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 only approximate values (which is good enough for the planner's purposes). They are initialized with dummy values (presently 1000 and 10 respectively) when a table is created. They are updated by certain commands, presently `VACUUM`, `ANALYZE`, and `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.

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 clause of the `WHERE` condition. The information used for this task is stored in the `pg_statistic` system catalog. Entries in `pg_statistic` are updated by `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 the 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:

```
    regression=# SELECT attname, n_distinct, most_common_vals FROM pg_stats WHERE table-
name = 'road';
     attname | n_distinct |
    ---------+------------+-------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
----------
     name    |  -0.467008 | {"I- 580                        Ramp","I- 880
road                        ","I- 580                            ","I- 680
```

```
80                          Ramp","14th                          St  ","5th
sion                        Blvd","I- 880                           "}
     thepath |         20 | {"[(-122.089,37.71),(-122.0886,37.711)]"}
    (2 rows)
    regression=#
```

Table 10-1 shows the columns that exist in `pg_stats`.

Table 10-1. `pg_stats` Columns

Name	Type	Description
`tablename`	`name`	Name of the table containing the column
`attname`	`name`	Column described by this row
`null_frac`	`real`	Fraction of column's entries that are null
`avg_width`	`integer`	Average width in bytes of the column's entries
`n_distinct`	`real`	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.
`most_common_vals`	`text[]`	A list of the most common values in the column. (Omitted if no values seem to be more common than any others.)
`most_common_freqs`	`real[]`	A list of the frequencies of the most common values, i.e., number of occurrences of each divided by total number of rows.

Name	Type	Description
histogram_bounds	text[]	A list of values that divide the column's values into groups of approximately equal population. The most_common_vals, if present, are omitted from the histogram calculation. (Omitted if column data type does not have a < operator, or if the most_common_vals list accounts for the entire population.)
correlation	real	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. (Omitted if column data type does not have a < operator.)

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. The default limit is presently 10 entries. Raising the limit may 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 may be appropriate for columns with simple data distributions.

10.3. Controlling the Planner with Explicit JOIN Clauses

Beginning with PostgreSQL 7.1 it has been 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

```
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 may 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 may 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 described in the *PostgreSQL Administrator's Guide*.) 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 much less freedom than it does for plain (inner) joins. For example, consider

```
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.

The PostgreSQL query planner treats all explicit `JOIN` syntaxes as constraining the join order, even though it is not logically necessary to make such a constraint for inner joins. Therefore, although all of these queries give the same result:

```
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 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.

You do not need to constrain the join order completely in order to cut search time, because it's OK to use `JOIN` operators in a plain `FROM` list. For example,

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

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.

If you have a mix of outer and inner joins in a complex query, you might not want to constrain the planner's search for a good ordering of inner joins inside an outer join. You can't do that directly in the `JOIN` syntax, but you can get around the syntactic limitation by using subselects. For example,

```
SELECT * FROM d LEFT JOIN
        (SELECT * FROM a, b, c WHERE ...) AS ss
        ON (...);
```

Here, joining D must be the last step in the query plan, but the planner is free to consider various join orders for A, B, C.

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.

10.4. Populating a Database

One may need to do a large number of table insertions when first populating a database. Here are some tips and techniques for making that as efficient as possible.

10.4.1. 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 may 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 record added. An additional benefit of doing all insertions in one transaction is that if the insertion of one record were to fail then the insertion of all records inserted up to that point would be rolled back, so you won't be stuck with partially loaded data.

10.4.2. Use COPY FROM

Use `COPY FROM STDIN` to load all the records in one command, instead of using a series of `INSERT` commands. This reduces parsing, planning, etc. overhead a great deal. If you do this then it is not necessary to turn off autocommit, since it is only one command anyway.

10.4.3. Remove Indexes

If you are loading a freshly created table, the fastest way is to create the table, bulk-load with `COPY`, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each record is loaded.

If you are augmenting an existing table, you can `DROP INDEX`, load the table, then recreate the index. Of course, the database performance for other users may 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.

10.4.4. Run ANALYZE Afterwards

It's a good idea to run `ANALYZE` or `VACUUM ANALYZE` anytime you've added or updated a lot of data, including just after initially populating a table. This ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner may make poor choices of query plans, leading to bad performance on queries that use your table.

Appendix A. 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 may 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.

A.1. Date/Time Input Interpretation

The date/time types are all decoded using a common set of routines.

Date/Time Input Interpretation

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 may have a text month.
    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 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 either a special string (e.g., `today`), day (e.g., `Thursday`), month (e.g., `January`), or noise word (e.g., `at`, `on`).

       Set field values and bit mask for fields. For example, set year, month, day for `today`, and additionally hour, minute, second for `now`.
    b. If not found, do a similar binary-search table lookup to match the token with a time zone.
    c. If not found, throw an error.
3. The token is a number or number field.
    a. If there are more than 4 digits, and if no other date fields have been previously read, then interpret as a “concatenated date” (e.g., `19990118`). 8 and 6 digits are interpreted as year, month, and day, while 7 and 5 digits are interpreted as year, day of year, respectively.
    b. If the token is three digits and a year has already been decoded, then interpret as day of year.
    c. If four or six digits and a year has already been read, then interpret as a time.
    d. If four or more digits, then interpret as a year.

e. If in European date mode, and if the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.

f. If the month field has not yet been read, and if the value is less than or equal to 12, then interpret as a month.

g. If the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.

h. If two digits or four or more digits, then interpret as a year.

i. Otherwise, throw an error.

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 `1BC` becomes year zero.)

5. If BC was not specified, and if the year field was two digits in length, then adjust the year to 4 digits. If the field was less than 70, then add 2000; otherwise, add 1900.

Tip: Gregorian years AD 1-99 may be entered by using 4 digits with leading zeros (e.g., `0099` is AD 99). Previous versions of PostgreSQL accepted years with three digits and with single digits, but as of version 7.0 the rules have been tightened up to reduce the possibility of ambiguity.

A.2. Date/Time Key Words

Table A-1 shows the tokens that are permissible as abbreviations for the names of the month.

Table A-1. Month Abbreviations

Month	Abbreviations
April	Apr
August	Aug
December	Dec
February	Feb
January	Jan
July	Jul
June	Jun
March	Mar
November	Nov
October	Oct
September	Sep, Sept

Note: The month May has no explicit abbreviation, for obvious reasons.

Table A-2 shows the tokens that are permissible as abbreviations for the names of the days of the week.

Table A-2. Day of the Week Abbreviations

Day	Abbreviation
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Table A-3 shows the tokens that serve various modifier purposes.

Table A-3. Date/Time Field Modifiers

Identifier	Description
`ABSTIME`	Key word ignored
`AM`	Time is before 12:00
`AT`	Key word ignored
`JULIAN, JD, J`	Next field is Julian Day
`ON`	Key word ignored
`PM`	Time is on or after after 12:00
`T`	Next field is time

The key word `ABSTIME` is ignored for historical reasons; in very old releases of PostgreSQL invalid fields 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.

Table A-4 shows the time zone abbreviations recognized by PostgreSQL. PostgreSQL contains internal tabular information for time zone decoding, since there is no standard operating system interface to provide access to general, cross-time zone information. The underlying operating system *is* used to provide time zone information for *output*, however.

The table is organized by time zone offset from UTC, rather than alphabetically; this is intended to facilitate matching local usage with recognized abbreviations for cases where these might differ.

Table A-4. Time Zone Abbreviations

Time Zone	Offset from UTC	Description
NZDT	+13:00	New Zealand Daylight Time
IDLE	+12:00	International Date Line, East
NZST	+12:00	New Zealand Standard Time
NZT	+12:00	New Zealand Time

Time Zone	Offset from UTC	Description
AESST	+11:00	Australia Eastern Summer Standard Time
ACSST	+10:30	Central Australia Summer Standard Time
CADT	+10:30	Central Australia Daylight Savings Time
SADT	+10:30	South Australian Daylight Time
AEST	+10:00	Australia Eastern Standard Time
EAST	+10:00	East Australian Standard Time
GST	+10:00	Guam Standard Time, USSR Zone 9
LIGT	+10:00	Melbourne, Australia
SAST	+09:30	South Australia Standard Time
CAST	+09:30	Central Australia Standard Time
AWSST	+09:00	Australia Western Summer Standard Time
JST	+09:00	Japan Standard Time,USSR Zone 8
KST	+09:00	Korea Standard Time
MHT	+09:00	Kwajalein Time
WDT	+09:00	West Australian Daylight Time
MT	+08:30	Moluccas Time
AWST	+08:00	Australia Western Standard Time
CCT	+08:00	China Coastal Time
WADT	+08:00	West Australian Daylight Time
WST	+08:00	West Australian Standard Time
JT	+07:30	Java Time
ALMST	+07:00	Almaty Summer Time
WAST	+07:00	West Australian Standard Time
CXT	+07:00	Christmas (Island) Time
MMT	+06:30	Myannar Time
ALMT	+06:00	Almaty Time
MAWT	+06:00	Mawson (Antarctica) Time
IOT	+05:00	Indian Chagos Time
MVT	+05:00	Maldives Island Time
TFT	+05:00	Kerguelen Time
AFT	+04:30	Afganistan Time
EAST	+04:00	Antananarivo Savings Time
MUT	+04:00	Mauritius Island Time

Time Zone	Offset from UTC	Description
RET	+04:00	Reunion Island Time
SCT	+04:00	Mahe Island Time
IRT, IT	+03:30	Iran Time
EAT	+03:00	Antananarivo, Comoro Time
BT	+03:00	Baghdad Time
EETDST	+03:00	Eastern Europe Daylight Savings Time
HMT	+03:00	Hellas Mediterranean Time (?)
BDST	+02:00	British Double Standard Time
CEST	+02:00	Central European Savings Time
CETDST	+02:00	Central European Daylight Savings Time
EET	+02:00	Eastern Europe, USSR Zone 1
FWT	+02:00	French Winter Time
IST	+02:00	Israel Standard Time
MEST	+02:00	Middle Europe Summer Time
METDST	+02:00	Middle Europe Daylight Time
SST	+02:00	Swedish Summer Time
BST	+01:00	British Summer Time
CET	+01:00	Central European Time
DNT	+01:00	*Dansk Normal Tid*
FST	+01:00	French Summer Time
MET	+01:00	Middle Europe Time
MEWT	+01:00	Middle Europe Winter Time
MEZ	+01:00	Middle Europe Zone
NOR	+01:00	Norway Standard Time
SET	+01:00	Seychelles Time
SWT	+01:00	Swedish Winter Time
WETDST	+01:00	Western Europe Daylight Savings Time
GMT	+00:00	Greenwich Mean Time
UT	+00:00	Universal Time
UTC	+00:00	Universal Time, Coordinated
Z	+00:00	Same as UTC
ZULU	+00:00	Same as UTC
WET	+00:00	Western Europe
WAT	-01:00	West Africa Time
NDT	-02:30	Newfoundland Daylight Time
ADT	-03:00	Atlantic Daylight Time

Time Zone	Offset from UTC	Description
AWT	-03:00	(unknown)
NFT	-03:30	Newfoundland Standard Time
NST	-03:30	Newfoundland Standard Time
AST	-04:00	Atlantic Standard Time (Canada)
ACST	-04:00	Atlantic/Porto Acre Summer Time
ACT	-05:00	Atlantic/Porto Acre Standard Time
EDT	-04:00	Eastern Daylight Time
CDT	-05:00	Central Daylight Time
EST	-05:00	Eastern Standard Time
CST	-06:00	Central Standard Time
MDT	-06:00	Mountain Daylight Time
MST	-07:00	Mountain Standard Time
PDT	-07:00	Pacific Daylight Time
AKDT	-08:00	Alaska Daylight Time
PST	-08:00	Pacific Standard Time
YDT	-08:00	Yukon Daylight Time
AKST	-09:00	Alaska Standard Time
HDT	-09:00	Hawaii/Alaska Daylight Time
YST	-09:00	Yukon Standard Time
MART	-09:30	Marquesas Time
AHST	-10:00	Alaska-Hawaii Standard Time
HST	-10:00	Hawaii Standard Time
CAT	-10:00	Central Alaska Time
NT	-11:00	Nome Time
IDLW	-12:00	International Date Line, West

Australian Time Zones. There are three naming conflicts between Australian time zone names with time zones commonly used in North and South America: `ACST`, `CST`, and `EST`. If the run-time option `AUSTRALIAN_TIMEZONES` is set to true then `ACST`, `CST`, `EST`, and `SAT` are interpreted as Australian time zone names, as shown in Table A-5. If it is false (which is the default), then `ACST`, `CST`, and `EST` are taken as American time zone names, and `SAT` is interpreted as a noise word indicating Saturday.

Table A-5. Australian Time Zone Abbreviations

Time Zone	Offset from UTC	Description
ACST	+09:30	Central Australia Standard Time
CST	+10:30	Australian Central Standard Time
EST	+10:00	Australian Eastern Standard Time

Time Zone	Offset from UTC	Description
SAT	+09:30	South Australian Standard Time

A.3. History of Units

Note: Contributed by José Soares (<jose@sferacarta.com>)

The Julian Day was invented by the French scholar Joseph Justus Scaliger (1540-1609) and probably takes its name from the Scaliger's father, the Italian scholar Julius Caesar Scaliger (1484-1558). Astronomers have used the Julian period to assign a unique number to every day since 1 January 4713 BC. This is the so-called Julian Day (JD). JD 0 designates the 24 hours from noon UTC on 1 January 4713 BC to noon UTC on 2 January 4713 BC.

The "Julian Day" is different from the "Julian Date". The Julian date refers to the Julian calendar, which was introduced by Julius Caesar in 45 BC. It was in common use until the 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.

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 only 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
```

> **Note:** 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 1752-09-03 and 1752-09-13, although eliminated in some countries by Papal fiat, conform to “natural rules” and are hence valid dates.

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 the 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.

Appendix B. SQL Key Words

Table B-1 lists all tokens that are key words in the SQL standard and in PostgreSQL 7.3.2. Background information can be found in Section 1.1.1.

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 B-1 in the column for PostgreSQL we classify as "non-reserved" those key words that are explicitly known to the parser but are allowed in most or all contexts where an identifier is expected. 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 only allowed as "AS" column label names (and perhaps in very few other contexts). Some reserved key words are allowable as names for functions; this is also shown in the table.

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 B-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 B-1. SQL Key Words

Key Word	PostgreSQL	SQL 99	SQL 92
ABORT	non-reserved		
ABS		non-reserved	
ABSOLUTE	non-reserved	reserved	reserved
ACCESS	non-reserved		
ACTION	non-reserved	reserved	reserved
ADA		non-reserved	non-reserved
ADD	non-reserved	reserved	reserved
ADMIN		reserved	
AFTER	non-reserved	reserved	
AGGREGATE	non-reserved	reserved	
ALIAS		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
ALL	reserved	reserved	reserved
ALLOCATE		reserved	reserved
ALTER	non-reserved	reserved	reserved
ANALYSE	reserved		
ANALYZE	reserved		
AND	reserved	reserved	reserved
ANY	reserved	reserved	reserved
ARE		reserved	reserved
ARRAY		reserved	
AS	reserved	reserved	reserved
ASC	reserved	reserved	reserved
ASENSITIVE		non-reserved	
ASSERTION	non-reserved	reserved	reserved
ASSIGNMENT	non-reserved	non-reserved	
ASYMMETRIC		non-reserved	
AT	non-reserved	reserved	reserved
ATOMIC		non-reserved	
AUTHORIZATION	reserved (can be function)	reserved	reserved
AVG		non-reserved	reserved
BACKWARD	non-reserved		
BEFORE	non-reserved	reserved	
BEGIN	non-reserved	reserved	reserved
BETWEEN	reserved (can be function)	non-reserved	reserved
BIGINT	non-reserved (cannot be function or type)		
BINARY	reserved (can be function)	reserved	
BIT	non-reserved (cannot be function or type)	reserved	reserved
BITVAR		non-reserved	
BIT_LENGTH		non-reserved	reserved
BLOB		reserved	
BOOLEAN	non-reserved (cannot be function or type)	reserved	
BOTH	reserved	reserved	reserved
BREADTH		reserved	
BY	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
C		non-reserved	non-reserved
CACHE	non-reserved		
CALL		reserved	
CALLED	non-reserved	non-reserved	
CARDINALITY		non-reserved	
CASCADE	non-reserved	reserved	reserved
CASCADED		reserved	reserved
CASE	reserved	reserved	reserved
CAST	reserved	reserved	reserved
CATALOG		reserved	reserved
CATALOG_NAME		non-reserved	non-reserved
CHAIN	non-reserved	non-reserved	
CHAR	non-reserved (cannot be function or type)	reserved	reserved
CHARACTER	non-reserved (cannot be function or type)	reserved	reserved
CHARACTERISTICS	non-reserved		
CHARACTER_LENGTH		non-reserved	reserved
CHARACTER_SET_CATALOG		non-reserved	non-reserved
CHARACTER_SET_NAME		non-reserved	non-reserved
CHARACTER_SET_SCHEMA		non-reserved	non-reserved
CHAR_LENGTH		non-reserved	reserved
CHECK	reserved	reserved	reserved
CHECKED		non-reserved	
CHECKPOINT	non-reserved		
CLASS	non-reserved	reserved	
CLASS_ORIGIN		non-reserved	non-reserved
CLOB		reserved	
CLOSE	non-reserved	reserved	reserved
CLUSTER	non-reserved		
COALESCE	non-reserved (cannot be function or type)	non-reserved	reserved
COBOL		non-reserved	non-reserved
COLLATE	reserved	reserved	reserved
COLLATION		reserved	reserved
COLLATION_CATALOG		non-reserved	non-reserved
COLLATION_NAME		non-reserved	non-reserved

Key Word	PostgreSQL	SQL 99	SQL 92
COLLATION_SCHEMA		non-reserved	non-reserved
COLUMN	reserved	reserved	reserved
COLUMN_NAME		non-reserved	non-reserved
COMMAND_FUNCTION		non-reserved	non-reserved
COMMAND_FUNCTION_CODE		non-reserved	
COMMENT	non-reserved		
COMMIT	non-reserved	reserved	reserved
COMMITTED	non-reserved	non-reserved	non-reserved
COMPLETION		reserved	
CONDITION_NUMBER		non-reserved	non-reserved
CONNECT		reserved	reserved
CONNECTION		reserved	reserved
CONNECTION_NAME		non-reserved	non-reserved
CONSTRAINT	reserved	reserved	reserved
CONSTRAINTS	non-reserved	reserved	reserved
CONSTRAINT_CATALOG		non-reserved	non-reserved
CONSTRAINT_NAME		non-reserved	non-reserved
CONSTRAINT_SCHEMA		non-reserved	non-reserved
CONSTRUCTOR		reserved	
CONTAINS		non-reserved	
CONTINUE		reserved	reserved
CONVERSION	non-reserved		
CONVERT	non-reserved (cannot be function or type)	non-reserved	reserved
COPY	non-reserved		
CORRESPONDING		reserved	reserved
COUNT		non-reserved	reserved
CREATE	reserved	reserved	reserved
CREATEDB	non-reserved		
CREATEUSER	non-reserved		
CROSS	reserved (can be function)	reserved	reserved
CUBE		reserved	
CURRENT		reserved	reserved
CURRENT_DATE	reserved	reserved	reserved
CURRENT_PATH		reserved	
CURRENT_ROLE		reserved	
CURRENT_TIME	reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
CURRENT_TIMESTAMP	reserved	reserved	reserved
CURRENT_USER	reserved	reserved	reserved
CURSOR	non-reserved	reserved	reserved
CURSOR_NAME		non-reserved	non-reserved
CYCLE	non-reserved	reserved	
DATA		reserved	non-reserved
DATABASE	non-reserved		
DATE		reserved	reserved
DATETIME_INTERVAL_CODE		non-reserved	non-reserved
DATETIME_INTERVAL_PRECISION		non-reserved	non-reserved
DAY	non-reserved	reserved	reserved
DEALLOCATE	non-reserved	reserved	reserved
DEC	non-reserved (cannot be function or type)	reserved	reserved
DECIMAL	non-reserved (cannot be function or type)	reserved	reserved
DECLARE	non-reserved	reserved	reserved
DEFAULT	reserved	reserved	reserved
DEFERRABLE	reserved	reserved	reserved
DEFERRED	non-reserved	reserved	reserved
DEFINED		non-reserved	
DEFINER	non-reserved	non-reserved	
DELETE	non-reserved	reserved	reserved
DELIMITER	non-reserved		
DELIMITERS	non-reserved		
DEPTH		reserved	
DEREF		reserved	
DESC	reserved	reserved	reserved
DESCRIBE		reserved	reserved
DESCRIPTOR		reserved	reserved
DESTROY		reserved	
DESTRUCTOR		reserved	
DETERMINISTIC		reserved	
DIAGNOSTICS		reserved	reserved
DICTIONARY		reserved	
DISCONNECT		reserved	reserved
DISPATCH		non-reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
DISTINCT	reserved	reserved	reserved
DO	reserved		
DOMAIN	non-reserved	reserved	reserved
DOUBLE	non-reserved	reserved	reserved
DROP	non-reserved	reserved	reserved
DYNAMIC		reserved	
DYNAMIC_FUNCTION		non-reserved	non-reserved
DYNAMIC_FUNCTION_CODE		non-reserved	
EACH	non-reserved	reserved	
ELSE	reserved	reserved	reserved
ENCODING	non-reserved		
ENCRYPTED	non-reserved		
END	reserved	reserved	reserved
END-EXEC		reserved	reserved
EQUALS		reserved	
ESCAPE	non-reserved	reserved	reserved
EVERY		reserved	
EXCEPT	reserved	reserved	reserved
EXCEPTION		reserved	reserved
EXCLUSIVE	non-reserved		
EXEC		reserved	reserved
EXECUTE	non-reserved	reserved	reserved
EXISTING		non-reserved	
EXISTS	non-reserved (cannot be function or type)	non-reserved	reserved
EXPLAIN	non-reserved		
EXTERNAL	non-reserved	reserved	reserved
EXTRACT	non-reserved (cannot be function or type)	non-reserved	reserved
FALSE	reserved	reserved	reserved
FETCH	non-reserved	reserved	reserved
FINAL		non-reserved	
FIRST		reserved	reserved
FLOAT	non-reserved (cannot be function or type)	reserved	reserved
FOR	reserved	reserved	reserved
FORCE	non-reserved		
FOREIGN	reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
FORTRAN		non-reserved	non-reserved
FORWARD	non-reserved		
FOUND		reserved	reserved
FREE		reserved	
FREEZE	reserved (can be function)		
FROM	reserved	reserved	reserved
FULL	reserved (can be function)	reserved	reserved
FUNCTION	non-reserved	reserved	
G		non-reserved	
GENERAL		reserved	
GENERATED		non-reserved	
GET	non-reserved	reserved	reserved
GLOBAL	non-reserved	reserved	reserved
GO		reserved	reserved
GOTO		reserved	reserved
GRANT	reserved	reserved	reserved
GRANTED		non-reserved	
GROUP	reserved	reserved	reserved
GROUPING		reserved	
HANDLER	non-reserved		
HAVING	reserved	reserved	reserved
HIERARCHY		non-reserved	
HOLD		non-reserved	
HOST		reserved	
HOUR	non-reserved	reserved	reserved
IDENTITY		reserved	reserved
IGNORE		reserved	
ILIKE	reserved (can be function)		
IMMEDIATE	non-reserved	reserved	reserved
IMMUTABLE	non-reserved		
IMPLEMENTATION		non-reserved	
IMPLICIT	non-reserved		
IN	reserved (can be function)	reserved	reserved
INCREMENT	non-reserved		
INDEX	non-reserved		

Key Word	PostgreSQL	SQL 99	SQL 92
INDICATOR		reserved	reserved
INFIX		non-reserved	
INHERITS	non-reserved		
INITIALIZE		reserved	
INITIALLY	reserved	reserved	reserved
INNER	reserved (can be function)	reserved	reserved
INOUT	non-reserved	reserved	
INPUT	non-reserved	reserved	reserved
INSENSITIVE	non-reserved	non-reserved	reserved
INSERT	non-reserved	reserved	reserved
INSTANCE		non-reserved	
INSTANTIABLE		non-reserved	
INSTEAD	non-reserved		
INT	non-reserved (cannot be function or type)	reserved	reserved
INTEGER	non-reserved (cannot be function or type)	reserved	reserved
INTERSECT	reserved	reserved	reserved
INTERVAL	non-reserved (cannot be function or type)	reserved	reserved
INTO	reserved	reserved	reserved
INVOKER	non-reserved	non-reserved	
IS	reserved (can be function)	reserved	reserved
ISNULL	reserved (can be function)		
ISOLATION	non-reserved	reserved	reserved
ITERATE		reserved	
JOIN	reserved (can be function)	reserved	reserved
K		non-reserved	
KEY	non-reserved	reserved	reserved
KEY_MEMBER		non-reserved	
KEY_TYPE		non-reserved	
LANCOMPILER	non-reserved		
LANGUAGE	non-reserved	reserved	reserved
LARGE		reserved	
LAST		reserved	reserved
LATERAL		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
LEADING	reserved	reserved	reserved
LEFT	reserved (can be function)	reserved	reserved
LENGTH		non-reserved	non-reserved
LESS		reserved	
LEVEL	non-reserved	reserved	reserved
LIKE	reserved (can be function)	reserved	reserved
LIMIT	reserved	reserved	
LISTEN	non-reserved		
LOAD	non-reserved		
LOCAL	non-reserved	reserved	reserved
LOCALTIME	reserved	reserved	
LOCALTIMESTAMP	reserved	reserved	
LOCATION	non-reserved		
LOCATOR		reserved	
LOCK	non-reserved		
LOWER		non-reserved	reserved
M		non-reserved	
MAP		reserved	
MATCH	non-reserved	reserved	reserved
MAX		non-reserved	reserved
MAXVALUE	non-reserved		
MESSAGE_LENGTH		non-reserved	non-reserved
MESSAGE_OCTET_LENGTH		non-reserved	non-reserved
MESSAGE_TEXT		non-reserved	non-reserved
METHOD		non-reserved	
MIN		non-reserved	reserved
MINUTE	non-reserved	reserved	reserved
MINVALUE	non-reserved		
MOD		non-reserved	
MODE	non-reserved		
MODIFIES		reserved	
MODIFY		reserved	
MODULE		reserved	reserved
MONTH	non-reserved	reserved	reserved
MORE		non-reserved	non-reserved
MOVE	non-reserved		

Key Word	PostgreSQL	SQL 99	SQL 92
MUMPS		non-reserved	non-reserved
NAME		non-reserved	non-reserved
NAMES	non-reserved	reserved	reserved
NATIONAL	non-reserved	reserved	reserved
NATURAL	reserved (can be function)	reserved	reserved
NCHAR	non-reserved (cannot be function or type)	reserved	reserved
NCLOB		reserved	
NEW	reserved	reserved	
NEXT	non-reserved	reserved	reserved
NO	non-reserved	reserved	reserved
NOCREATEDB	non-reserved		
NOCREATEUSER	non-reserved		
NONE	non-reserved (cannot be function or type)	reserved	
NOT	reserved	reserved	reserved
NOTHING	non-reserved		
NOTIFY	non-reserved		
NOTNULL	reserved (can be function)		
NULL	reserved	reserved	reserved
NULLABLE		non-reserved	non-reserved
NULLIF	non-reserved (cannot be function or type)	non-reserved	reserved
NUMBER		non-reserved	non-reserved
NUMERIC	non-reserved (cannot be function or type)	reserved	reserved
OBJECT		reserved	
OCTET_LENGTH		non-reserved	reserved
OF	non-reserved	reserved	reserved
OFF	reserved	reserved	
OFFSET	reserved		
OIDS	non-reserved		
OLD	reserved	reserved	
ON	reserved	reserved	reserved
ONLY	reserved	reserved	reserved
OPEN		reserved	reserved
OPERATION		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
OPERATOR	non-reserved		
OPTION	non-reserved	reserved	reserved
OPTIONS		non-reserved	
OR	reserved	reserved	reserved
ORDER	reserved	reserved	reserved
ORDINALITY		reserved	
OUT	non-reserved	reserved	
OUTER	reserved (can be function)	reserved	reserved
OUTPUT		reserved	reserved
OVERLAPS	reserved (can be function)	non-reserved	reserved
OVERLAY	non-reserved (cannot be function or type)	non-reserved	
OVERRIDING		non-reserved	
OWNER	non-reserved		
PAD		reserved	reserved
PARAMETER		reserved	
PARAMETERS		reserved	
PARAMETER_MODE		non-reserved	
PARAMETER_NAME		non-reserved	
PARAMETER_ORDINAL_POSITION		non-reserved	
PARAMETER_SPECIFIC_CATALOG		non-reserved	
PARAMETER_SPECIFIC_NAME		non-reserved	
PARAMETER_SPECIFIC_SCHEMA		non-reserved	
PARTIAL	non-reserved	reserved	reserved
PASCAL		non-reserved	non-reserved
PASSWORD	non-reserved		
PATH	non-reserved	reserved	
PENDANT	non-reserved		
PLACING	reserved		
PLI		non-reserved	non-reserved
POSITION	non-reserved (cannot be function or type)	non-reserved	reserved
POSTFIX		reserved	
PRECISION	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
PREFIX		reserved	
PREORDER		reserved	
PREPARE	non-reserved	reserved	reserved
PRESERVE		reserved	reserved
PRIMARY	reserved	reserved	reserved
PRIOR	non-reserved	reserved	reserved
PRIVILEGES	non-reserved	reserved	reserved
PROCEDURAL	non-reserved		
PROCEDURE	non-reserved	reserved	reserved
PUBLIC		reserved	reserved
READ	non-reserved	reserved	reserved
READS		reserved	
REAL	non-reserved (cannot be function or type)	reserved	reserved
RECHECK	non-reserved		
RECURSIVE		reserved	
REF		reserved	
REFERENCES	reserved	reserved	reserved
REFERENCING		reserved	
REINDEX	non-reserved		
RELATIVE	non-reserved	reserved	reserved
RENAME	non-reserved		
REPEATABLE		non-reserved	non-reserved
REPLACE	non-reserved		
RESET	non-reserved		
RESTRICT	non-reserved	reserved	reserved
RESULT		reserved	
RETURN		reserved	
RETURNED_LENGTH		non-reserved	non-reserved
RETURNED_OCTET_LENGTH		non-reserved	non-reserved
RETURNED_SQLSTATE		non-reserved	non-reserved
RETURNS	non-reserved	reserved	
REVOKE	non-reserved	reserved	reserved
RIGHT	reserved (can be function)	reserved	reserved
ROLE		reserved	
ROLLBACK	non-reserved	reserved	reserved
ROLLUP		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
ROUTINE		reserved	
ROUTINE_CATALOG		non-reserved	
ROUTINE_NAME		non-reserved	
ROUTINE_SCHEMA		non-reserved	
ROW	non-reserved (cannot be function or type)	reserved	
ROWS		reserved	reserved
ROW_COUNT		non-reserved	non-reserved
RULE	non-reserved		
SAVEPOINT		reserved	
SCALE		non-reserved	non-reserved
SCHEMA	non-reserved	reserved	reserved
SCHEMA_NAME		non-reserved	non-reserved
SCOPE		reserved	
SCROLL	non-reserved	reserved	reserved
SEARCH		reserved	
SECOND	non-reserved	reserved	reserved
SECTION		reserved	reserved
SECURITY	non-reserved	non-reserved	
SELECT	reserved	reserved	reserved
SELF		non-reserved	
SENSITIVE		non-reserved	
SEQUENCE	non-reserved	reserved	
SERIALIZABLE	non-reserved	non-reserved	non-reserved
SERVER_NAME		non-reserved	non-reserved
SESSION	non-reserved	reserved	reserved
SESSION_USER	reserved	reserved	reserved
SET	non-reserved	reserved	reserved
SETOF	non-reserved (cannot be function or type)		
SETS		reserved	
SHARE	non-reserved		
SHOW	non-reserved		
SIMILAR	reserved (can be function)	non-reserved	
SIMPLE	non-reserved	non-reserved	
SIZE		reserved	reserved
SMALLINT	non-reserved (cannot be function or type)	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
SOME	reserved	reserved	reserved
SOURCE		non-reserved	
SPACE		reserved	reserved
SPECIFIC		reserved	
SPECIFICTYPE		reserved	
SPECIFIC_NAME		non-reserved	
SQL		reserved	reserved
SQLCODE			reserved
SQLERROR			reserved
SQLEXCEPTION		reserved	
SQLSTATE		reserved	reserved
SQLWARNING		reserved	
STABLE	non-reserved		
START	non-reserved	reserved	
STATE		reserved	
STATEMENT	non-reserved	reserved	
STATIC		reserved	
STATISTICS	non-reserved		
STDIN	non-reserved		
STDOUT	non-reserved		
STORAGE	non-reserved		
STRICT	non-reserved		
STRUCTURE		reserved	
STYLE		non-reserved	
SUBCLASS_ORIGIN		non-reserved	non-reserved
SUBLIST		non-reserved	
SUBSTRING	non-reserved (cannot be function or type)	non-reserved	reserved
SUM		non-reserved	reserved
SYMMETRIC		non-reserved	
SYSID	non-reserved		
SYSTEM		non-reserved	
SYSTEM_USER		reserved	reserved
TABLE	reserved	reserved	reserved
TABLE_NAME		non-reserved	non-reserved
TEMP	non-reserved		
TEMPLATE	non-reserved		
TEMPORARY	non-reserved	reserved	reserved
TERMINATE		reserved	

Key Word	PostgreSQL	SQL 99	SQL 92
THAN		reserved	
THEN	reserved	reserved	reserved
TIME	non-reserved (cannot be function or type)	reserved	reserved
TIMESTAMP	non-reserved (cannot be function or type)	reserved	reserved
TIMEZONE_HOUR		reserved	reserved
TIMEZONE_MINUTE		reserved	reserved
TO	reserved	reserved	reserved
TOAST	non-reserved		
TRAILING	reserved	reserved	reserved
TRANSACTION	non-reserved	reserved	reserved
TRANSACTIONS_COMMITTED		non-reserved	
TRANSACTIONS_ROLLED_BACK		non-reserved	
TRANSACTION_ACTIVE		non-reserved	
TRANSFORM		non-reserved	
TRANSFORMS		non-reserved	
TRANSLATE		non-reserved	reserved
TRANSLATION		reserved	reserved
TREAT	non-reserved (cannot be function or type)	reserved	
TRIGGER	non-reserved	reserved	
TRIGGER_CATALOG		non-reserved	
TRIGGER_NAME		non-reserved	
TRIGGER_SCHEMA		non-reserved	
TRIM	non-reserved (cannot be function or type)	non-reserved	reserved
TRUE	reserved	reserved	reserved
TRUNCATE	non-reserved		
TRUSTED	non-reserved		
TYPE	non-reserved	non-reserved	non-reserved
UNCOMMITTED		non-reserved	non-reserved
UNDER		reserved	
UNENCRYPTED	non-reserved		
UNION	reserved	reserved	reserved
UNIQUE	reserved	reserved	reserved
UNKNOWN	non-reserved	reserved	reserved

Key Word	PostgreSQL	SQL 99	SQL 92
UNLISTEN	non-reserved		
UNNAMED		non-reserved	non-reserved
UNNEST		reserved	
UNTIL	non-reserved		
UPDATE	non-reserved	reserved	reserved
UPPER		non-reserved	reserved
USAGE	non-reserved	reserved	reserved
USER	reserved	reserved	reserved
USER_DEFINED_TYPE_CATALOG		non-reserved	
USER_DEFINED_TYPE_NAME		non-reserved	
USER_DEFINED_TYPE_SCHEMA		non-reserved	
USING	reserved	reserved	reserved
VACUUM	non-reserved		
VALID	non-reserved		
VALIDATOR	non-reserved		
VALUE		reserved	reserved
VALUES	non-reserved	reserved	reserved
VARCHAR	non-reserved (cannot be function or type)	reserved	reserved
VARIABLE		reserved	
VARYING	non-reserved	reserved	reserved
VERBOSE	reserved (can be function)		
VERSION	non-reserved		
VIEW	non-reserved	reserved	reserved
VOLATILE	non-reserved		
WHEN	reserved	reserved	reserved
WHENEVER		reserved	reserved
WHERE	reserved	reserved	reserved
WITH	non-reserved	reserved	reserved
WITHOUT	non-reserved	reserved	
WORK	non-reserved	reserved	reserved
WRITE	non-reserved	reserved	reserved
YEAR	non-reserved	reserved	reserved
ZONE	non-reserved	reserved	reserved

Appendix C. SQL Conformance

This section attempts to outline to what extent PostgreSQL conforms to the SQL standard. Full compliance to the standard or a complete statement about the compliance to the standard is complicated and not particularly useful, so this section can only give an overview.

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 one appearing in 1999. That version is refered to as ISO/IEC 9075:1999, or informally as SQL99. The version prior to that was SQL92. PostgreSQL development tends to aim for conformance with the latest official version of the standard where such conformance does not contradict traditional features or common sense. At the time of this writing, ballotting is under way for a new revision of the standard, which, if approved, will eventually become the conformance target for future PostgreSQL development.

SQL92 defined three feature sets for conformance: Entry, Intermediate, and Full. Most database products 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.

SQL99 defines a large set of individual features rather than the ineffectively broad three levels found in SQL92. 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 SQL99 standard is also split into 5 parts: Framework, Foundation, Call Level Interface, Persistent Stored Modules, and Host Language Bindings. PostgreSQL only covers parts 1, 2, and 5. 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 in either case.

In the following two sections, we provide a list of those features that PostgreSQL supports, followed by a list of the features defined in SQL99 which are not yet supported in PostgreSQL. Both of these lists are approximate: There may be minor details that are nonconforming for a feature that is listed as supported, and large parts of an unsupported feature may 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.

C.1. Supported Features

Identifier	Package	Description	Comment
B012	Core	Embedded C	
B021		Direct SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	

Identifier	Package	Description	Comment
E011-02	Core	REAL, DOUBLE PRECISON, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	
E021-08	Core	UPPER and LOWER functions	
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character data types	
E021-11	Core	POSITION function	
E011-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	
E051	Core	Basic query specification	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	AS is required
E051-06	Core	HAVING clause	

Identifier	Package	Description	Comment
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	
E061-05	Core	LIKE predicate ESCAPE clause	
E061-06	Core	NULL predicate	
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in comparison predicate	
E061-11	Core	Subqueries in IN predicate	
E061-12	Core	Subqueries in quantified comparison predicate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expressions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	

Identifier	Package	Description	Comment
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-06	Core	REFERENCES privilege at the table level	
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	
E091-05	Core	SUM	
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipulation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-08	Core	CLOSE statement	(cursor)
E121-10	Core	FETCH statement implicit NEXT	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constraints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constraints	

Identifier	Package	Description	Comment
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	
E141-08	Core	NOT NULL inferred on PRIMARY KEY	
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	
E151-02	Core	ROLLBACK statement	
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E161	Core	SQL comments using leading double minus	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	

Identifier	Package	Description	Comment
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join	
F041-08	Core	All comparison operators are supported (rather than just =)	
F051	Core	Basic date and time	
F051-01	Core	DATE data type (including support of DATE literal)	
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST between datetime types and character types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052	Enhanced datetime facilities	Intervals and datetime arithmetic	

Identifier	Package	Description	Comment
F081	Core	UNION and EXCEPT in views	
F111-02		READ COMMITTED isolation level	
F131	Core	Grouped operations	
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables supported in queries with grouped views	
F131-03	Core	Set functions supported in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F191	Enhanced integrity management	Referential delete actions	
F201	Core	CAST function	
F221	Core	Explicit defaults	
F222		INSERT statement: DEFAULT VALUES clause	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F271		Compound character literals	
F281		LIKE enhancements	

Identifier	Package	Description	Comment
F302	OLAP facilities	INTERSECT table operator	
F302-01	OLAP facilities	INTERSECT DISTINCT table operator	
F302-02	OLAP facilities	INTERSECT ALL table operator	
F304	OLAP facilities	EXCEPT ALL table operator	
F311	Core	Schema definition statement	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persistent base tables	
F311-03	Core	CREATE VIEW	
F311-05	Core	GRANT statement	
F321		User authorization	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statement: ALTER COLUMN clause	
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause	
F391		Long identifiers	
F401	OLAP facilities	Extended joined table	
F401-01	OLAP facilities	NATURAL JOIN	
F401-02	OLAP facilities	FULL OUTER JOIN	
F401-03	OLAP facilities	UNION JOIN	
F401-04	OLAP facilities	CROSS JOIN	
F411	Enhanced datetime facilities	Time zone specification	
F421		National character	
F431-01		FETCH with explicit NEXT	
F431-04		FETCH PRIOR	
F431-06		FETCH RELATIVE	

Identifier	Package	Description	Comment
F441		Extended set function support	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491	Enhanced integrity management	Constraint management	
F511		BIT data type	
F531		Temporary tables	
F555	Enhanced datetime facilities	Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	
F591	OLAP facilities	Derived tables	
F611		Indicator data types	
F651		Catalog name qualifiers	
F701	Enhanced integrity management	Referential update actions	
F761		Session management	
F791		Insensitive cursors	
F801		Full set function	
S071	Enhanced object support	SQL paths in function and type name resolution	
S111	Enhanced object support	ONLY in query expressions	
S211	Enhanced object support, SQL/MM support	User-defined cast functions	
T031		BOOLEAN data type	
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T191	Enhanced integrity management	Referential action RESTRICT	
T201	Enhanced integrity management	Comparable data types for referential constraints	
T211-01	Enhanced integrity management	Triggers activated on UPDATE, INSERT, or DELETE of one base table	
T211-02	Enhanced integrity management	BEFORE triggers	

Identifier	Package	Description	Comment
T211-03	Enhanced integrity management	AFTER triggers	
T211-04	Enhanced integrity management	FOR EACH ROW triggers	
T211-07	Enhanced integrity management	TRIGGER privilege	
T231		SENSITIVE cursors	
T241		START TRANSACTION statement	
T312		OVERLAY function	
T321-01	Core	User-defined functions with no overloading	
T321-03	Core	Function invocation	
T322	PSM, SQL/MM support	Overloading of SQL-invoked functions and procedures	
T323		Explicit security for external routines	
T351		Bracketed SQL comments (/*...*/ comments)	
T441		ABS and MOD functions	
T501		Enhanced EXISTS predicate	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	

C.2. Unsupported Features

The following features defined in SQL99 are not implemented in the current release of PostgreSQL. In a few cases, equivalent functionality is available.

Identifier	Package	Description	Comment
B011	Core	Embedded Ada	

Identifier	Package	Description	Comment
B013	Core	Embedded COBOL	
B014	Core	Embedded Fortran	
B015	Core	Embedded MUMPS	
B016	Core	Embedded Pascal	
B017	Core	Embedded PL/I	
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-1		<describe input> statement	
B041		Extensions to embedded SQL exception declarations	
B051		Enhanced execution rights	
E081	Core	Basic Privileges	
E081-05	Core	UPDATE privilege at the column level	
E081-07	Core	REFERENCES privilege at the column level	
E081-08	Core	WITH GRANT OPTION	
E121	Core	Basic cursor support	
E121-04	Core	OPEN statement	(cursor)
E121-06	Core	Positioned UPDATE statement	(cursor)
E121-07	Core	Positioned DELETE statement	(cursor)
E121-17	Core	WITH HOLD cursors	Cursor to stay open across transactions
E152	Core	Basic SET TRANSACTION statement	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	Syntax accepted; READ ONLY not supported
E153	Core	Updatable queries with subqueries	
E171	Core	SQLSTATE support	
F181		Multiple module support	
E182	Core	Module language	

Identifier	Package	Description	Comment
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	
F021-06	Core	CHECK_CONSTRAINTS view	
F031-19	Core	REVOKE statement: RESTRICT clause	
F034		Extended REVOKE statement	
F034-01		REVOKE statement performed by other than the owner of a schema object	
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F111		Isolation levels other than SERIALIZABLE	
F111-01		READ UNCOMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F231		Privilege Tables	

Identifier	Package	Description	Comment
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	
F231-03		USAGE_PRIVILEGES view	
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F341		Usage tables	
F431		Read-only scrollable cursors	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-05		FETCH ABSOLUTE	
F451		Character set definition	
F461		Named character sets	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F501-03	Core	SQL_LANGUAGES view	
F502		Enhanced documentation tables	
F502-01		SQL_SIZING_PROFILES view	
F502-02		SQL_IMPLEMENTATION_INFO view	
F502-03		SQL_PACKAGES view	
F521	Enhanced integrity management	Assertions	
F641	OLAP facilities	Row and table constructors	
F661		Simple tables	
F671	Enhanced integrity management	Subqueries in CHECK	intentionally omitted
F691		Collation and translation	
F711		ALTER domain	

Identifier	Package	Description	Comment
F721		Deferrable constraints	foreign keys only
F731		INSERT column privileges	
F741		Referential MATCH types	no partial match yet
F751		View CHECK enhancements	
F771		Connection management	
F781		Self-referencing operations	
F811		Extended flagging	
F812	Core	Basic flagging	
F813		Extended flagging for "Core SQL Flagging" and "Catalog Lookup" only	
F821		Local table references	
F831		Full cursor update	
F831-01		Updatable scrollable cursors	
F831-02		Updatable ordered cursors	
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023	Basic object support, SQL/MM support	Basic structured types	
S024, SQL/MM support	Enhanced object support	Enhanced structured types	
S041	Basic object support	Basic reference types	
S043	Enhanced object support	Enhanced reference types	
S051	Basic object support	Create table of type	
S081	Enhanced object support	Subtables	
S091	SQL/MM support	Basic array support	PostgreSQL arrays are different
S091-01	SQL/MM support	Arrays of built-in data types	
S091-02	SQL/MM support	Arrays of distinct types	
S091-03	SQL/MM support	Array expressions	
S092	SQL/MM support	Arrays of user-defined types	

Identifier	Package	Description	Comment
S094		Arrays of reference types	
S151	Basic object support	Type predicate	IS OF
S161	Enhanced object support	Subtype treatment	TREAT(expr AS type)
S201		SQL routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	
S231	Enhanced object support	Structured type locators	
S232		Array locators	
S241	Enhanced object support	Transform functions	
S251		User-defined orderings	CREATE ORDERING FOR
S261		Specific type method	
T011		Timestamp in Information Schema	
T041	Basic object support	Basic LOB data type support	
T041-01	Basic object support	BLOB data type	
T041-02	Basic object support	CLOB data type	
T041-03	Basic object support	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	
T041-04	Basic object support	Concatenation of LOB data types	
T041-05	Basic object support	LOB locator: non-holdable	
T042		Extended LOB data type support	
T051		Row types	
T111		Updatable joins, unions, and columns	
T121		WITH (excluding RECURSIVE) in query expression	
T131		Recursive query	
T171		LIKE clause in table definition	CREATE TABLE T1 (LIKE T2)

Identifier	Package	Description	Comment
T211	Enhanced integrity management, Active database	Basic trigger capability	
T211-05	Enhanced integrity management	Ability to specify a search condition that must be true before the trigger is invoked	
T211-06	Enhanced integrity management	Support for run-time rules for the interaction of triggers and constraints	
T211-08	Enhanced integrity management	Multiple triggers for the same the event are executed in the order in which they were created	
T212	Enhanced integrity management	Enhanced trigger capability	
T251		SET TRANSACTION statement: LOCAL option	
T261		Chained transactions	
T271		Savepoints	
T281		SELECT privilege with column granularity	
T301		Functional Dependencies	
T321	Core	Basic SQL-invoked routines	
T321-02	Core	User-defined stored procedures with no overloading	
T321-04	Core	CALL statement	
T321-05	Core	RETURN statement	
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T331		Basic roles	
T332		Extended roles	
T401		INSERT into a cursor	
T411		UPDATE statement: SET ROW option	
T431	OLAP facilities	CUBE and ROLLUP operations	

Identifier	Package	Description	Comment
T461		Symmetric BETWEEN predicate	
T471		Result sets return value	
T491		LATERAL derived table	
T511		Transaction counts	
T541		Updatable table references	
T561		Holdable locators	
T571		Array-returning external SQL-invoked functions	
T601		Local cursor references	

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[1]

SQL Reference Books

Judith Bowman, Sandra Emerson, and Marcy Darnovsky, *The Practical SQL Handbook: Using Structured Query Language*, Third Edition, Addison-Wesley, ISBN 0-201-44787-8, 1996.

C. J. Date and Hugh Darwen, *A Guide to the SQL Standard: A user's guide to the standard database language SQL*, Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.

C. J. Date, *An Introduction to Database Systems*, Volume 1, Sixth Edition, Addison-Wesley, 1994.

Ramez Elmasri and Shamkant Navathe, *Fundamentals of Database Systems*, 3rd Edition, Addison-Wesley, ISBN 0-805-31755-4, August 1999.

Jim Melton and Alan R. Simon, *Understanding the New SQL: A complete guide*, Morgan Kaufmann, ISBN 1-55860-245-3, 1993.

Jeffrey D. Ullman, *Principles of Database and Knowledge: Base Systems*, Volume 1, Computer Science Press, 1988.

PostgreSQL-Specific Documentation

Stefan Simkovics, *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Department of Information Systems, Vienna University of Technology, November 29, 1998.

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.

A. Yu and J. Chen, The POSTGRES Group, *The Postgres95 User Manual*, University of California, Sept. 5, 1995.

Zelaine Fong, *The design and implementation of the POSTGRES query optimizer*[2], University of California, Berkeley, Computer Science Department.

1. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/
2. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/UCB-MS-zfong.pdf

Proceedings and Articles

Nels Olson, *Partial indexing in POSTGRES: research project*, University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong and J. Goh, "A Unified Framework for Version Modeling Using Production Rules in a Database System", *ERL Technical Memorandum M90/33*, University of California, April, 1990.

L. Rowe and M. Stonebraker, "The POSTGRES data model[3]", Proc. VLDB Conference, Sept. 1987.

P. Seshadri and A. Swami, "Generalized Partial Indexes[4] ", Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, p. 420-7.

M. Stonebraker and L. Rowe, "The design of POSTGRES[5]", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, and C. H. Hong, "The design of the POSTGRES rules system", Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, "The design of the POSTGRES storage system[6]", Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, and S. Potamianos, "A commentary on the POSTGRES rules system[7]", *SIGMOD Record 18(3)*, Sept. 1989.

M. Stonebraker, "The case for partial indexes[8]", *SIGMOD Record 18(4)*, Dec. 1989, p. 4-11.

M. Stonebraker, L. A. Rowe, and M. Hirohama, "The implementation of POSTGRES[9]", *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, March 1990.

M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedures, Caching and Views in Database Systems[10]", Proc. ACM-SIGMOD Conference on Management of Data, June 1990.

3. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-13.pdf
4. http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z
5. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M85-95.pdf
6. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-06.pdf
7. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-82.pdf
8. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf
9. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-34.pdf
10. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-36.pdf

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y