PostgreSQL FAQ

Frequently Asked Questions (FAQ) for PostgreSQL

Last updated: Sat Jul 10 00:37:57 EDT 1999

Current maintainer: Bruce Momjian (maillist@candle.pha.pa.us)

The most recent version of this document can be viewed at the postgreSQL Web site, http://www.PostgreSQL.org.

Linux-specific questions are answered in http://www.PostgreSQL.org/docs/faq-linux.html.

Irix-specific questions are answered in http://www.PostgreSQL.org/docs/faq-irix.html.

HPUX-specific questions are answered in http://www.PostgreSQL.org/docs/faq-hpux.shtml.

General Questions

User Client Questions

Administrative Questions

Operational Questions

Extending PostgreSQL

General Questions

1.1) What is PostgreSQL?

PostgreSQL is an enhancement of the POSTGRES database management system, a next-generation DBMS research prototype. While PostgreSQL retains the powerful data model and rich data types of POSTGRES, it replaces the PostQuel query language with an extended subset of SQL. PostgreSQL is free and the complete source is available.

PostgreSQL development is being performed by a team of Internet developers who all subscribe to the PostgreSQL development mailing list. The current coordinator is Marc G. Fournier (scrappy@postgreSQL.org). (See below on how to join). This team is now responsible for all current and future development of PostgreSQL.

The authors of PostgreSQL 1.01 were Andrew Yu and Jolly Chen. Many others have contributed to the porting, testing, debugging and enhancement of the code. The original Postgres code, from which PostgreSQL is derived, was the effort of many graduate students, undergraduate students, and staff programmers working under the direction of Professor Michael Stonebraker at the University of California, Berkeley.

The original name of the software at Berkeley was Postgres. When SQL functionality was added in 1995, its name was changed to Postgres95. The name was changed at the end of 1996 to PostgreSQL.

1.2) What's the copyright on PostgreSQL?

PostgreSQL is subject to the following COPYRIGHT.

PostgreSQL Data Base Management System

Copyright (c) 1994-6 Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

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

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE

SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

1.3) What Unix platforms does PostgreSQL run on?

The authors have compiled and tested PostgreSQL on the following platforms (some of these compiles require gcc):

- aix - IBM on AIX 3.2.5 or 4.x
- alpha - DEC Alpha AXP on Digital Unix 2.0, 3.2, 4.0
- BSD44_derived - OSs derived from 4.4-lite BSD (NetBSD, FreeBSD)
- bsdi - BSD/OS 2.x, 3.x, 4.x
- dgux - DG/UX 5.4R4.11
- hpux - HP PA-RISC on HP-UX 9.*, 10.*
- i386_solaris - i386 Solaris
- irix5 - SGI MIPS on IRIX 5.3
- linux - Intel i86
  Alpha
  SPARC
  PPC
  M68k
- sco - SCO 3.2v5
  Unixware
- sparc_solaris - SUN SPARC on Solaris 2.4, 2.5, 2.5.1
- sunos4 - SUN SPARC on SunOS 4.1.3
- svr4 - Intel x86 on Intel SVR4 and MIPS
- ultrix4 - DEC MIPS on Ultrix 4.4

1.4) What non-unix ports are available?

It is possible to compile the libpq C library, psql, and other interfaces and binaries to run on MS Windows platforms. In this case, the client is running on MS Windows, and communicates via TCP/IP to a server running on one of our supported Unix platforms.

A file *win31.mak* is included in the distribution for making a Win32 libpq library and psql.

The database server is now working on Windows NT using the Cygnus Unix/NT porting library. See pgsql/doc/README.NT in the distribution.

There is another port using U/Win at http://surya.wipro.com/uwin/ported.html.

1.5) Where can I get PostgreSQL?

The primary anonymous ftp site for PostgreSQL is ftp://ftp.postgreSQL.org/pub

For mirror sites, see our main web site.

1.6) Where can I get support for PostgreSQL?

There is no official support for PostgreSQL from the University of California, Berkeley. It is maintained through volunteer effort.

The main mailing list is: pgsql-general@postgreSQL.org. It is available for discussion of matters pertaining to PostgreSQL. To subscribe, send a mail with the lines in the body (not the subject line)

```
subscribe
end
```

to pgsql-general-request@postgreSQL.org.

There is also a digest list available. To subscribe to this list, send email to: pgsql-general-digest-request@postgreSQL.orgwith a BODY of:

```
subscribe
end
```

Digests are sent out to members of this list whenever the main list has received around 30k of messages.

The bugs mailing list is available. To subscribe to this list, send email to bugs-request@postgreSQL.org with a BODY of:

```
subscribe
end
```

There is also a developers discussion mailing list available. To subscribe to this list, send email to hackers-request@postgreSQL.org with a BODY of:

```
subscribe
end
```

Additional mailing lists and information about PostgreSQL can be found via the PostgreSQL WWW home page at:

http://postgreSQL.org

There is also an IRC channel on EFNet, channel #PostgreSQL. I use the unix command

```
irc -c '#PostgreSQL' "$USER"
irc.phoenix.net
```

Commercial support for PostgreSQL is available at http://www.pgsql.com/

1.7) What is the latest release of PostgreSQL?

The latest release of PostgreSQL is version 6.5.

We plan to have major releases every four months.

1.8) What documentation is available for PostgreSQL?

Several manuals, manual pages, and some small test examples are included in the distribution. See the /doc directory.

psql has some nice \d commands to show information about types, operators, functions, aggregates, etc.

The web site contains even more documentation.

1.9) How do I find out about known bugs or missing features?

PostgreSQL supports an extended subset of SQL-92. See our TODOfor a list of known bugs, missing features, and future plans.

1.10) How can I learn SQL?

There is a nice tutorial at http://w3.one.net/~jhoffman/sqltut.htmand at http://our-world.compuserve.com/homepages/Graeme_Birchall/DB2_COOK.HTM.

Many of our users like *The Practical SQL Handbook*, Bowman et al., Addison Wesley.

1.11) Is PostgreSQL Y2K compliant?

Yes, we easily handle dates past the year 2000AD, and before 2000BC.

1.12) How do I join the development team?

First, download the latest sources and read the PostgreSQL Developers documentation on our web site, or in the distribution. Second, subscribe to the pgsql-hackers and pgsql-patches mailing lists. Third, submit high-quality patches to pgsql-patches.

There are about a dozen people who have *commit* privileges to the PostgreSQL CVS archive. All of them have submitted so many high-quality patches that it was a pain for the existing committers to keep up, and we had confidence that patches they committed were likely to be of high quality.

1.13) How do I submit a bug report?

Fill out the "bug-template" file and send it to: bugs@postgreSQL.org

Also check out our ftp site ftp://ftp.postgreSQL.org/pubto see if there is a more recent PostgreSQL version or patches.

1.14) How does PostgreSQL compare to other DBMS's?

There are several ways of measuring software: features, performance, reliability, support, and price.

- **Features**
  PostgreSQL has most features present in large commercial DBMS's, like transactions, subselects, triggers, views, and sophisticated locking. We have some features they don't have, like user-defined types, inheritance, rules, and multi-version concurrency control to reduce lock contention. We don't have foreign key referential integrity or outer joins, but are working on them for our next release.

- **Performance**
  PostgreSQL runs in two modes. Normal *fsync* mode flushes every completed transaction to disk, guaranteeing that if the OS crashes or looses power in the next few seconds, all your data is safely stored on disk. In this mode, we are slower than most commercial databases, partly because few of them do such conservative flushing to disk in their default modes. In *no-fsync* mode, we are usually faster than commercial databases, though in this mode, an OS crash could cause data corruption. We are working to provide an intermediate mode that suffers from less performance overhead than full fsync mode, and will allow data integrity within 30 seconds of an OS crash. The mode is select-able by the database administrator.
  In comparison to MySQL or leaner database systems, we are slower on inserts/updates because we have transaction overhead. Of course, MySQL doesn't have any of the features mentioned in the *Features* section above. We are built for flexibility and features, though we continue to improve performance through profiling and source code analysis.

- **Reliability**
  We realize that a DBMS must be reliable, or it is worthless. We strive to release well-tested, stable code that has a minimum of bugs. Each release has at least one month of beta testing, and our release history shows that we can provide stable, solid releases that are ready for production use. We believe we compare favorably to other database software in this area.

- **Support**
  Our mailing list provides a large group of developers and users to help resolve any problems encountered. While we can not guarantee a fix, commercial DBMS's don't always supply a fix either. Direct access to developers, the user community, manuals, and the source code often make PostgreSQL support superior to other DBMS's. There is commercial per-incident support available for those who need it. (See support FAQ item.)

- **Price**
  We are free for all use, both commercial and non-commercial. You can add our code to your product with no limitations, except those outlined in our BSD-style license stated above.

User Client Questions

2.1) Are there ODBC drivers for PostgreSQL?

There are two ODBC drivers available, PostODBC and OpenLink ODBC.

PostODBC is included in the distribution. More information about it can be gotten from: http://www.insightdist.com/psqlodbc

OpenLink ODBC can be gotten from http://www.openlinksw.com. It works with their standard ODBC client software so you'll have PostgreSQL ODBC available on every client platform they support (Win, Mac, Unix, VMS).

They will probably be selling this product to people who need commercial-quality support, but a freeware version will always be available. Questions to postgres95@openlink.co.uk.

2.2) What tools are available for hooking PostgreSQL to Web pages?

A nice introduction to Database-backed Web pages can be seen at: http://www.webtools.com

There is also one at http://www.phone.net/home/mwm/hotlist/.

For web integration, PHP is an excellent interface. It is at: http://www.php.net

PHP is great for simple stuff, but for more complex cases, many use the perl interface and CGI.pm.

A WWW gateway based on WDB using perl can be downloaded from http://www.eol.ists.ca/~dunlop/wdb-p95

2.3) Does PostgreSQL have a graphical user interface? A report generator? An embedded query language interface?

We have a nice graphical user interface called *pgaccess*, which is shipped as part of the distribution. *Pgaccess* also has a report generator. The web page is http://www.flex.ro/pgaccess

We also include *ecpg*, which is an embedded SQL query language interface for C.

2.4) What languages are available to communicate with PostgreSQL?

We have:

- C(libpq)
- C++(libpq++)
- Embedded C(ecpg)
- Java(jdbc)
- Perl(perl5)
- ODBC(odbc)
- Python(PyGreSQL)
- TCL(libpgtcl)
- A crude C/4GL(contrib/pginterface)
- Embedded HTML(PHP from http://www.php.net)

Administrative Questions

3.1) Why does initdb fail?

- check that you don't have any of the previous version's binaries in your path (If you see the message

```
WARN:heap_modifytuple: repl is
\  9
```

, this is the problem.)

- check to see that you have the proper paths set
- check that the *postgres* user owns the proper files

3.2) How do I install PostgreSQL somewhere other than /usr/local/pgsql?

The simplest way is to specify the --prefix option when running configure. If you forgot to do that, you can edit Makefile.global and change POSTGRESDIR accordingly, or create a Makefile.custom and define POSTGRESDIR there.

3.3) When I start the postmaster, I get a *Bad System Call* or core dumped message. Why?

It could be a variety of problems, but first check to see that you have system V extensions installed on your kernel. PostgreSQL requires kernel support for shared memory and semaphores.

3.4) When I try to start the postmaster, I get *IpcMemoryCreate* errors. Why?

You either do not have shared memory configured properly in kernel or you need to enlarge the shared memory available in the kernel. The exact amount you need depends on your architecture and how many buffers and backend processes you configure postmaster to run with. For most systems, with default numbers of buffers and processes, you need a minimum of ˜1MB.

3.5) When I try to start the postmaster, I get *IpcSemaphoreCreate* errors. Why?

If the error message is *IpcSemaphoreCreate: semget failed (No space left on device)* then your kernel is not configured with enough semaphores. Postgres needs one semaphore per potential backend process. A temporary solution is to start the postmaster with a smaller limit on the number of backend processes. Use *-N* with a parameter less than the default of 32. A more permanent solution is to increase your kernel's *SEMMNS* and *SEMMNI* parameters.

If the error message is something else, you might not have semaphore support configured in your kernel at all.

3.6) How do I prevent other hosts from accessing my PostgreSQL database?

By default, PostgreSQL only allows connections from the local machine using unix domain sockets. Other machines will not be able to connect unless you add the *-i* flag to the *postmaster,* **and** enable host-based authentication by modifying the file *$PGDATA/pg_hba.conf* accordingly. This will allow TCP/IP connections.

3.7) Why can't I connect to my database from another machine?

The default configuration allows only unix domain socket connections from the local machine. To enable TCP/IP connections, make sure the postmaster has been started with the *-i* option, and add an appropriate host entry to the file *pgsql/data/pg_hba.conf*. See the *pg_hba.conf* manual page.

3.8) Why can't I access the database as the *root* user?

You should not create database users with user id 0 (root). They will be unable to access the database. This is a security precaution because of the ability of any user to dynamically link object modules into the database engine.

3.9) All my servers crash under concurrent table access. Why?

This problem can be caused by a kernel that is not configured to support semaphores.

3.10) How do I tune the database engine for better performance?

Certainly, indices can speed up queries. The *explain* command allows you to see how PostgreSQL is interpreting your query, and which indices are being used.

If you are doing a lot of *inserts*, consider doing them in a large batch using the *copy* command. This is much faster than single individual *inserts*. Second, statements not in a *begin work/commit* transaction block are considered to be in their own transaction. Consider performing several statements in a single transaction block. This reduces the transaction overhead. Also consider dropping and recreating indices when making large data changes.

There are several tuning things that can be done. You can disable fsync() by starting the postmaster with a *-o -F* option. This will prevent *fsync()'s* from flushing to disk after every transaction.

You can also use the postmaster -B option to increase the number of shared memory buffers used by the backend processes. If you make this parameter too high, the postmaster may not start up because you've exceeded your kernel's limit on shared memory space. Each buffer is 8K and the default is 64 buffers.

You can also use the backend -S option to increase the maximum amount of memory used by each backend process for temporary sorts. The -S value is measured in kilobytes, and the default is 512 (ie, 512K). It is unwise to make this value too large, or you may run out of memory when a query invokes several concurrent sorts.

You can also use the *cluster* command to group data in base tables to match an index. See the cluster(l) manual page for more details.

3.11) What debugging features are available in PostgreSQL?

PostgreSQL has several features that report status information that can be valuable for debugging purposes.

First, by running configure with the --enable-cassert option, many *assert()'s* monitor the progress of the backend and halt the program when something unexpected occurs.

Both postmaster and postgres have several debug options available. First, whenever you start the postmaster, make sure you send the standard output and error to a log file, like:

```
cd /usr/local/pgsql
./bin/postmaster >server.log 2>&1 &
```

This will put a server.log file in the top-level PostgreSQL directory. This file contains useful information about problems or errors encountered by the server. Postmaster has a -d option that allows even more detailed information to be reported. The -d option takes

a number that specifies the debug level. Be warned that high debug level values generate large log files.

You can actually run the postgres backend from the command line, and type your SQL statement directly. This is recommended **only** for debugging purposes. Note that a newline terminates the query, not a semicolon. If you have compiled with debugging symbols, you can use a debugger to see what is happening. Because the backend was not started from the postmaster, it is not running in an identical environment and locking/backend interaction problems may not be duplicated. Some operating system can attach to a running backend directly to diagnose problems.

The postgres program has -s, -A, and -t options that can be very useful for debugging and performance measurements.

You can also compile with profiling to see what functions are taking execution time. The backend profile files will be deposited in the pgsql/data/base/dbname directory. The client profile file will be put in the current directory.

3.12) I get 'Sorry, too many clients' when trying to connect. Why?

You need to increase the postmaster's limit on how many concurrent backend processes it can start.

In Postgres 6.5, the default limit is 32 processes. You can increase it by restarting the postmaster with a suitable *-N* value. With the default configuration you can set *-N* as large as 1024; if you need more, increase *MAXBACKENDS* in *include/config.h* and rebuild. You can set the default value of *-N* at configuration time, if you like, using configure's *--with-maxbackends* switch.

Note that if you make *-N* larger than 32, you should consider increasing *-B* beyond its default of 64. For large numbers of backend processes, you are also likely to find that you need to increase various Unix kernel configuration parameters. Things to check include the maximum size of shared memory blocks, *SHMMAX,* the maximum number of semaphores, *SEMMNS* and *SEMMNI,* the maximum number of processes, *NPROC,* the maximum number of processes per user, *MAXUPRC,* and the maximum number of open files, *NFILE* and *NINODE.* The reason that Postgres has a limit on the number of allowed backend processes is so that you can ensure that your system won't run out of resources.

In Postgres versions prior to 6.5, the maximum number of backends was 64, and changing it required a rebuild after altering the MaxBackendId constant in *include/storage/sinvaladt.h.*

3.13) What are the pg_tempNNN.NN files in my database directory?

They are temporary files generated by the query executor. For example, if a sort needs to be done to satisfy an *order by,* and the sort requires more space than the backend's -S parameter allows, then temp files are created to hold the extra data.

The temp files should go away automatically, but might not if a backend crashes during a sort. If you have no transactions running at the time, it is safe to delete the pg_tempNNN.NN files.

3.14) How do I set up a pg_group?

Currently, there is no easy interface to set up user groups. You have to explicitly insert/update the pg_group table. For example:

```
jolly=> insert into pg_group (groname, grosysid, grolist)
jolly=>     values ('posthackers', '1234', '{5443, 8261}');
INSERT 548224
jolly=> grant insert on foo to group posthackers;
CHANGE
jolly=>
```

The fields in pg_group are:

- groname: the group name. This a name and should be purely alphanumeric. Do not include underscores or other punctuation.
- grosysid: the group id. This is an int4. This should be unique for each group.
- grolist: the list of pg_user id's that belong in the group. This is an int4[].

Operational Questions

4.1) The system seems to be confused about commas, decimal points, and date formats.

Check your locale configuration. PostgreSQL uses the locale settings of the user that ran the postmaster process. There are postgres and psql SET commands to control the date format. Set those accordingly for your operating environment.

4.2) What is the exact difference between binary cursors and normal cursors?

See the *declare* manual page for a description.

4.3) How do I *select* only the first few rows of a query?

See the *fetch* manual page, or use SELECT ... LIMIT....

This only prevents all row results from being transferred to the client. The entire query must be evaluated, even if you only want just the first few rows. Consider a query that has an *order by.* There is no way to return any rows until the entire query is evaluated and sorted.

4.4) How do I get a list of tables, or other information I see in *psql?*

You can read the source code for *psql*, file pgsql/src/bin/psql/psql.c. It contains SQL commands that generate the output for psql's backslash commands. Beginning in Postgres 6.5, you can also start *psql* with the *-E* option so that it will print out the queries it uses to execute the commands you give.

4.5) How do you remove a column from a table?

We do not support *alter table drop column*, but do this:

```
     SELECT ...
-- select all columns but the one you want to remove
     INTO TABLE new_table
     FROM old_table;
     DROP TABLE old_table;
     ALTER TABLE new_table RENAME TO old_table;
```

4.6) What is the maximum size for a row, table, database?

Rows are limited to 8K bytes, but this can be changed by editing *include/config.h* and changing *BLCKSZ*. To use attributes larger than 8K, you can also use the large object interface.

Rows do not cross 8k boundaries so a 5k row will require 8k of storage.

Table and database sizes are unlimited. There are many databases that are tens of gigabytes, and probably some that are hundreds.

4.7)How much database disk space is required to store data from a typical flat file?

A Postgres database can require about six and a half times the disk space required to store the data in a flat file.

Consider a file of 300,000 lines with two integers on each line. The flat file is 2.4MB. The size of the PostgreSQL database file containing this data can be estimated at 14MB:

```
 36 bytes: each row header (approximate)
+ 8 bytes: two int fields @ 4 bytes each
+ 4 bytes: pointer on page to tuple
----------------------------------------
 48 bytes per row
The data page size in PostgreSQL is 8192 bytes (8 KB), so:
8192 bytes per page
------------------   =  171 rows per database page (rounded up)
  48 bytes per row
300000 data rows
------------------  =  1755 database pages
   171 rows per page
1755 database pages * 8192 bytes per page  =  14,376,960 bytes (14MB)
```

Indexes do not contain as much overhead, but do contain the data that is being indexed, so they can be large also.

4.8) How do I find out what indices or operations are defined in the database?

psql has a variety of backslash commands to show such information. Use \? to see them.

Also try the file *pgsql/src/tutorial/syscat.source.* It illustrates many of the *select*s needed to get information from the database system tables.

4.9) My queries are slow or don't make use of the indexes. Why?

PostgreSQL does not automatically maintain statistics. One has to make an explicit *vacuum* call to update the statistics.
After statistics are updated, the optimizer knows how many rows in the table, and can better decide if it should use indices. Note that the optimizer does not use indices in cases when the table is small because a sequential scan would be faster.

For column-specific optimization statistics, use *vacuum analyze. Vacuum analyze* is important for complex multi-join queries, so the optimizer can estimate the number of rows returned from each table, and choose the proper join order. The backend does not keep track of column statistics on its own, so *vacuum analyze* must be run to collect them periodically.

Indexes are not used for *order by* operations.

When using wild-card operators such as *LIKE* or ~, indices can only be used if the beginning of the search is anchored to the start of the string. So, to use indices, *LIKE* searches can should not begin with *%*, and ~(regular expression searches) should start with ^.

4.10) How do I see how the query optimizer is evaluating my query?

See the *explain* manual page.

4.11) What is an R-tree index?

An r-tree index is used for indexing spatial data. A hash index can't handle range searches. A B-tree index only handles range searches in a single dimension. R-tree's can handle multi-dimensional data. For example, if an R-tree index can be built on an attribute of type *point*, the system can more efficient answer queries like select all points within a bounding rectangle.

The canonical paper that describes the original R-Tree design is:

Guttman, A. "R-Trees: A Dynamic Index Structure for Spatial Searching." Proc of the 1984 ACM SIGMOD Int'l Conf on Mgmt of Data, 45-57.

You can also find this paper in Stonebraker's "Readings in Database Systems"

Builtin R-Trees can handle polygons and boxes. In theory, R-trees can be extended to handle higher number of dimensions. In practice, extending R-trees require a bit of work and we don't currently have any documentation on how to do it.

4.12) What is Genetic Query Optimization?

The GEQO module in PostgreSQL is intended to solve the query optimization problem of joining many tables by means of a Genetic Algorithm (GA). It allows the handling of large join queries through non-exhaustive search.

For further information see the documentation.

4.13) How do I do regular expression searches and case-insensitive regexp searching?

~ and ~* are probably what you want. See psql's *\do* command.

4.14) In a query, how do I detect if a field is NULL?

You test the column with IS NULL and IS NOT NULL.

4.15) What is the difference between the various character types?

```
Type             Internal Name   Notes
-------------------------------------------------
CHAR             char            1 character
CHAR(#)          bpchar          blank padded to the specified fixed length
VARCHAR(#)       varchar         size specifies maximum length, no padding
TEXT             text            length limited only by maximum row length
BYTEA            bytea           variable-length array of bytes
```

You need to use the internal name when doing internal operations.

The last four types above are "varlena" types (i.e. the first four bytes are the length, followed by the data). *char(#)* allocates the maximum number of bytes no matter how much data is stored in the field. *text*, *varchar(#)*, and *bytea* all have variable length on the disk, and because of this, there is a small performance penalty for using them. Specifically, the penalty is for access to all columns after the first column of this type.

4.16) How do I create a serial/auto-incrementing field?

PostgreSQL supports a *serial* data type. It auto-creates a sequence and index on the column. See the *create_sequence* manual page for more information about sequences. You can also use each row's *oid* field as a unique value. However, if you need to dump and reload the database, you need to use *pg_dump's -o* option or *copy with oids* option to preserve the oids.

4.17) What is an oid? What is a tid?

Oids are PostgreSQL's answer to unique row ids. Every row that is created in PostgreSQL gets a unique oid. All oids generated during initdb are less than 16384 (from *backend/access/transam.h*). All user-created oids are equal or greater that this. By default, all these oids are unique not only within a table, or database, but unique within the entire PostgreSQL installation.

PostgreSQL uses oids in its internal system tables to link rows between tables. These oids can be used to identify specific user rows and used in joins. It is recommended you use column type oid to store oid values. See the *sql(l)* manual page to see the other internal columns. You can create an index on the oid field for faster access.

Oids are assigned to all new rows from a central area that is used by all databases. If you want to change the oid to something else, or if you want to make a copy of the table, with the original oid's, there is no reason you can't do it:

```
CREATE TABLE new_table(old_oid oid, mycol int);
SELECT INTO new SELECT old_oid, mycol FROM old;
COPY new TO '/tmp/pgtable';
DELETE FROM new;
COPY new WITH OIDS FROM '/tmp/pgtable';
```

Tids are used to identify specific physical rows with block and offset values. Tids change after rows are modified or reloaded. They are used by index entries to point to physical rows.

4.18) What is the meaning of some of the terms used in PostgreSQL?

Some of the source code and older documentation use terms that have more common usage. Here are some:

- row, record, tuple
- attribute, field, column
- table, class
- retrieve, select
- replace, update
- append, insert
- oid, serial value
- portal, cursor
- range variable, table name, table alias

4.19) Why do I get the error "FATAL: palloc failure: memory exhausted?"

It is possible you have run out of virtual memory on your system, or your kernel has a low limit for certain resources. Try this before starting the postmaster:

```
ulimit -d 65536
limit datasize 64m
```

Depending on your shell, only one of these may succeed, but it will set your process data segment limit much higher and perhaps allow the query to complete. This command applies to the current process, and all subprocesses created after the command is run. If you are having a problem with the SQL client because the backend is returning too much data, try it before starting the client.

4.20) How do I tell what PostgreSQL version I am running?

From *psql*, type

```
select version();
```

4.21) My large-object operations get *invalid large obj descriptor.* **Why?**

You need to put

```
BEGIN WORK
```

and

```
COMMIT
```

around any use of a large object handle, that is, surrounding

```
lo_open
```

...

```
lo_close.
```

The documentation has always stated that *lo_open* must be wrapped in a transaction, but PostgreSQL versions prior to 6.5 didn't enforce that rule. Instead, they'd just fail occasionally if you broke it.

Current PostgreSQL enforces the rule by closing large object handles at transaction commit, which will be instantly upon completion of the *lo_open* command if you are not inside a transaction. So the first attempt to do anything with the handle will draw *invalid large obj descriptor.* So code that used to work (at least most of the time) will now generate that error message if you fail to use a transaction.

If you are using a client interface like ODBC you may need to set

```
auto-commit off.
```

5.1) I wrote a user-defined function. When I run it in *psql*, why does it dump core?

The problem could be a number of things. Try testing your user-defined function in a stand alone test program first. Also, make sure you are not sending elog NOTICES when the front-end is expecting data, such as during a type_in() or type_out() functions

5.2) What does the message: *NOTICE:PortalHeapMemoryFree: 0x402251d0 not in alloc set!* mean?

You are *pfree'ing* something that was not *palloc'ed.* Beware of mixing *malloc/free* and *palloc/pfree.*

5.3) How can I contribute some nifty new types and functions for PostgreSQL?

Send your extensions to the pgsql-hackers mailing list, and they will eventually end up in the *contrib/* subdirectory.

5.4) How do I write a C function to return a tuple?

This requires wizardry so extreme that the authors have never tried it, though in principle it can be done.

5.5) I have changed a source file. Why does the recompile does not see the change?

The Makefiles do not have the proper dependencies for include files. You have to do a *make clean* and then another *make.*