PostgreSQL 6.4.2 Documentation

The PostgreSQL Development Team

PostgreSQL 6.4.2 Documentation

by The PostgreSQL Development Team

Covering v6.4 for general release

Table of Contents

List of Figures

List of Tables

List of Examples

Summary

Postgres, developed originally in the UC Berkeley Computer Science Department, pioneered many of the object-relational concepts now becoming available in some commercial databases. It provides SQL92/ SQL3 language support, transaction integrity, and type extensibility. PostgreSQL is a public-domain, open source descendant of this original Berkeley code.

Part I. Tutorial

Introduction for new users.

Table of Contents

Chapter 1. Introduction

This document is the user manual for the PostgreSQL[1] database management system, originally developed at the University of California at Berkeley. PostgreSQL is based on Postgres release 4.2[2]. 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.

1.1. What is Postgres?

Traditional relational database management systems (DBMSs) 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, as mentioned, this simplicity often makes the implementation of certain applications very difficult. Postgres offers substantial additional power by incorporating the following four additional basic concepts in such a way that users can easily extend the system:

classes
inheritance
types
functions

Other features provide additional power and flexibility:

constraints
triggers
rules
transaction integrity

These features put Postgres 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 the traditional relational database languages. So, although Postgres has some object-oriented features, it is firmly in the relational database world. In fact, some commercial databases have recently incorporated features pioneered by Postgres.

1.2. A Short History of Postgres

1.2.1. 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 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. We released Version 1, described in The

[1] http://postgresql.org/
[2] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

Implementation of Postgres, 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, releases since then have 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[3] (since merged into Informix[4]) picked up the code and commercialized it. Postgres became the primary data manager for the Sequoia 2000[5] scientific computing project in late 1992. Furthermore, 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 project officially ended with Version 4.2.

1.2.2. Postgres95

In 1994, Andrew Yu[6] and Jolly Chen[7] added a SQL language interpreter to Postgres, and the code was subsequently released to the Web to find its own way in the world. Postgres95 was a public-domain, open source descendant of this original Berkeley code.

Postgres95 is a derivative of the last official release of Postgres (version 4.2). The code is now completely ANSI C and the code size has been trimmed by 25%. There are a lot of internal changes that improve performance and code maintainability. Postgres95 v1.0.x runs about 30-50% faster on the Wisconsin Benchmark compared to v4.2. Apart from bug fixes, these are the major enhancements:

- The query language Postquel has been replaced with SQL (implemented in the server). We do not yet support subqueries (which can be imitated with user defined SQL functions). Aggregates have been re-implemented. We also added support for ``GROUP BY''. The `libpq` interface is still available for C programs.

- In addition to the monitor program, we provide a new program (psql) which supports GNU `readline`.

- We added a new front-end library, `libpgtcl`, that supports Tcl-based clients. A sample shell, pgtclsh, provides new Tcl commands to interface tcl programs with the Postgres95 backend.

- The large object interface has been overhauled. We kept Inversion large objects as the only mechanism for storing large objects. (This is not to be confused with the Inversion file system which has been removed.)

- The instance-level rule system has been removed. Rules are still available as rewrite rules.

- A short tutorial introducing regular SQL features as well as those of ours is distributed with the source code.

- GNU make (instead of BSD make) is used for the build. Also, Postgres95 can be compiled with an unpatched gcc (data alignment of doubles has been fixed).

[3] http://www.illustra.com/
[4] http://www.informix.com/
[5] http://www.sdsc.edu/0/Parts_Collabs/S2K/s2k_home.html
[6] mailto:ayu@informix.com
[7] http://http.cs.berkeley.edu/~jolly/

1.2.3. PostgreSQL

By 1996, it became clear that the name “Postgres95” would not stand the test of time. A new name, PostgreSQL, was chosen to reflect the relationship between original Postgres and the more recent versions with SQL capability. At the same time, the version numbering was reset to start at 6.0, putting the numbers back into the sequence originally begun by the Postgres Project.

The emphasis on development for the v1.0.x releases of Postgres95 was on stabilizing the backend code. With the v6.x series of PostgreSQL, the emphasis has shifted from identifying and understanding existing problems in the backend to augmenting features and capabilities, although work continues in all areas.

Major enhancements include:

- 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 coersion, 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 startup time has decreased 80% since v6.0 was released.

1.3. About This Release

PostgreSQL is available without cost. This manual describes version 6.4 of PostgreSQL.

We will use Postgres to mean the version distributed as PostgreSQL.

Check the Administrator's Guide for a list of currently supported machines. In general, Postgres is portable to any Unix/Posix-compatible system with full libc library support.

1.4. Resources

This manual set is organized into several parts:

Tutorial

An introduction for new users. Does not cover advanced features.

User's Guide

General information for users, including available commands and data types.

Programmer's Guide

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

Administrator's Guide

Installation and management information. List of supported machines.

Developer's Guide

Information for Postgres developers. This is intended for those who are contributing to the Postgres project; application development information should appear in the *Programmer's Guide*. Currently included in the *Programmer's Guide*.

Reference Manual

Detailed reference information on command syntax. Currently included in the *User's Guide*.

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

man pages

The man pages have general information on command syntax.

FAQs

The Frequently Asked Questions (FAQ) documents address both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

Web Site

The Postgres[8] web site has some information not appearing in the distribution. There is a mhonarc catalog of mailing list traffic which is a rich resource for many topics.

Mailing Lists

The Postgres Questions[9] mailing list is a good place to have user questions answered. Other mailing lists are available; consult the web page for details.

Yourself!

Postgres is an open source product. As such, it depends on the user community for ongoing support. As you begin to use Postgres, 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 it. 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 Postgres Documentation[10] mailing list is the place to get going.

1.5. Terminology

In the following documentation, *site* may be interpreted as the host machine on which Postgres is installed. Since it is possible to install more than one set of Postgres databases on a single host, this term more precisely denotes any particular set of installed Postgres binaries and databases.

The Postgres *superuser* is the user named `postgres` who owns the Postgres binaries and database files. As the database superuser, all protection mechanisms may be bypassed and any data accessed arbitrarily.

[8] postgresql.org
[9] mailto:questions@postgresql.org
[10] mailto:docs@postgresql.org

In addition, the Postgres superuser is allowed to execute some support programs which are generally not available to all users. Note that the Postgres superuser is *not* the same as the Unix superuser (which will be referred to as *root*). The superuser should have a non-zero user identifier (*UID*) for security reasons.

The *database administrator* or DBA, is the person who is responsible for installing Postgres with mechanisms to enforce a security policy for a site. The DBA can add new users by the method described below and maintain a set of template databases for use by createdb.

The postmaster is the process that acts as a clearing-house for requests to the Postgres system. Frontend applications connect to the postmaster, which keeps tracks of any system errors and communication between the backend processes. The postmaster can take several command-line arguments to tune its behavior. However, supplying arguments is necessary only if you intend to run multiple sites or a non-default site.

The Postgres backend (the actual executable program postgres) may be executed directly from the user shell by the Postgres super-user (with the database name as an argument). However, doing this bypasses the shared buffer pool and lock table associated with a postmaster/site, therefore this is not recommended in a multiuser site.

1.6. Notation

"..." or `/usr/local/pgsql/` at the front of a file name is used to represent the path to the Postgres superuser's home directory.

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.

In examples, parentheses ("(" and ")") are used to group boolean expressions. "|" is the boolean operator OR.

Examples will show commands executed from various accounts and programs. Commands executed from the root account will be preceeded with ">". Commands executed from the Postgres superuser account will be preceeded with "%", while commands executed from an unprivileged user's account will be preceeded with "$". SQL commands will be preceeded with "=>" or will have no leading prompt, depending on the context.

Note

At the time of writing (Postgres v6.4) the notation for flagging commands is not universally consistant throughout the documentation set. Please report problems to the Documentation Mailing List[11].

1.7. Y2K Statement

Author

Written by Thomas Lockhart[12] on 1998-10-22.

The PostgreSQL Global Development Team provides the Postgres software code tree as a public service, without warranty and without liability for it's behavior or performance. However, at the time of writing:

[11] mailto:docs@postgresql.org
[12] mailto:lockhart@alumni.caltech.edu

- The author of this statement, a volunteer on the Postgres support team since November, 1996, is not aware of any problems in the Postgres code base related to time transitions around Jan 1, 2000 (Y2K).

- The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of Postgres. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.

- To the best of the author's knowledge, the assumptions Postgres makes about dates specified with a two-digit year are documented in the current User's Guide[13] in the chapter on data types. For two-digit years, the significant transition year is 1970, not 2000; e.g. “70-01-01” is interpreted as “1970-01-01”, whereas “69-01-01” is interpreted as “2069-01-01”.

- Any Y2K problems in the underlying OS related to obtaining "the current time" may propagate into apparent Y2K problems in Postgres.

Refer to The Gnu Project[14] and The Perl Institute[15] for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

1.8. Copyrights and Trademarks

[13] http://www.postgresql.org/docs/user/datatype.htm
[14] http://www.gnu.org/software/year2000.html
[15] http://language.perl.com/news/y2k.html

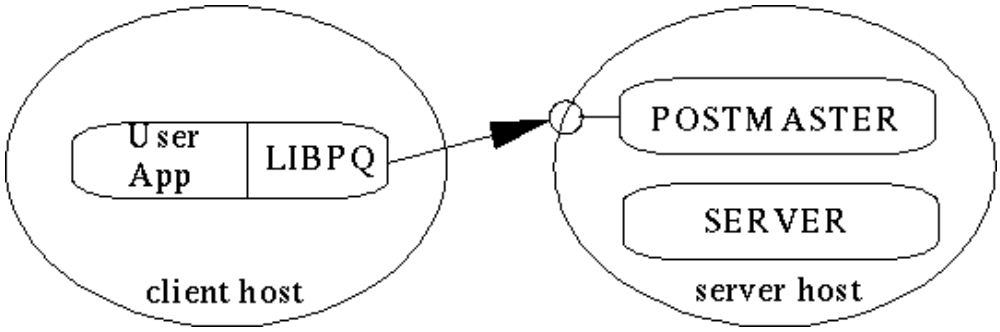
Chapter 2. Architecture

2.1. Postgres Architectural Concepts

Before we begin, you should understand the basic Postgres system architecture. Understanding how the parts of Postgres interact will make the next chapter somewhat clearer. In database jargon, Postgres uses a simple "process per-user" client/server model. A Postgres session consists of the following cooperating UNIX processes (programs):

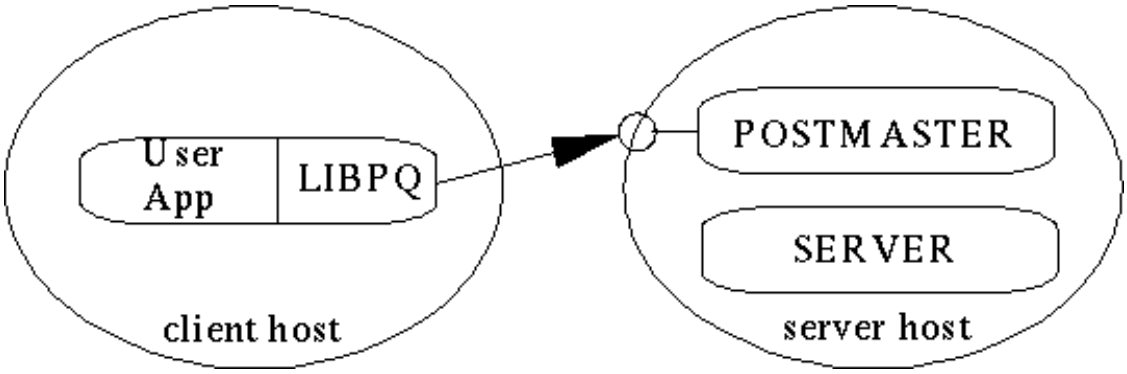
- A supervisory daemon process (postmaster),
- the user's frontend application (e.g., the psql program), and
- the one or more backend database servers (the postgres process itself).

A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called an installation or site. Frontend applications that wish to access a given database within an installation make calls to the library. The library sends user requests over the network to the postmaster (



), which in turn starts a new backend server process

Figure 2.1. How a connection is established



and connects the frontend process to the new server. From that point on, the frontend process and the backend server communicate without intervention by the postmaster. Hence, the postmaster is always running, waiting for requests, whereas frontend and backend processes come and go.

The `libpq` library allows a single frontend to make multiple connections to backend processes. However, the frontend application is still a single-threaded process. Multithreaded frontend/backend connections are not currently supported in `libpq`. One implication of this architecture is that the postmaster and the backend always run on the same machine (the database server), while the frontend application may run

anywhere. You should keep this in mind, because the files that can be accessed on a client machine may not be accessible (or may only be accessed using a different filename) on the database server machine.

You should also be aware that the postmaster and postgres servers run with the user-id of the Postgres "superuser." Note that the Postgres superuser does not have to be a special user (e.g., a user named "postgres"). Furthermore, the Postgres superuser should definitely not be the UNIX superuser ("root")! In any case, all files relating to a database should belong to this Postgres superuser.

Chapter 3. Getting Started

How to begin work with Postgres for a new user.

Some of the steps required to use Postgres can be performed by any Postgres user, and some must be done by the site database administrator. This site administrator is the person who installed the software, created the database directories and started the postmaster process. This person does not have to be the UNIX superuser ("root") or the computer system administrator; a person can install and use Postgres without any special accounts or privileges.

If you are installing Postgres yourself, then refer to the Administrator's Guide for instructions on installation, and return to this guide when the installation is complete.

Throughout this manual, any examples that begin with the character "%" are commands that should be typed at the UNIX shell prompt. Examples that begin with the character "*" are commands in the Postgres query language, Postgres SQL.

3.1. Setting Up Your Environment

This section discusses how to set up your own environment so that you can use frontend applications. We assume Postgres has already been successfully installed and started; refer to the Administrator's Guide and the installation notes for how to install Postgres.

Postgres is a client/server application. As a user, you only need access to the client portions of the installation (an example of a client application is the interactive monitor psql). For simplicity, we will assume that Postgres has been installed in the directory `/usr/local/pgsql`. Therefore, wherever you see the directory `/usr/local/pgsql` you should substitute the name of the directory where Postgres is actually installed. All Postgres commands are installed in the directory `/usr/local/pgsql/bin`. Therefore, you should add this directory to your shell command path. If you use a variant of the Berkeley C shell, such as csh or tcsh, you would add

```
% set path = ( /usr/local/pgsql/bin path )
```

in the `.login` file in your home directory. If you use a variant of the Bourne shell, such as sh, ksh, or bash, then you would add

```
% PATH=/usr/local/pgsql/bin PATH
% export PATH
```

to the .profile file in your home directory. From now on, we will assume that you have added the Postgres bin directory to your path. In addition, we will make frequent reference to "setting a shell variable" or "setting an environment variable" throughout this document. If you did not fully understand the last paragraph on modifying your search path, you should consult the UNIX manual pages that describe your shell before going any further.

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

3.2. Starting the Interactive Monitor (psql)

Assuming that your site administrator has properly started the postmaster process and authorized you to use the database, you (as a user) may begin to start up applications. As previously mentioned, you should add `/usr/local/pgsql/bin` to your shell search path. In most cases, this is all you should have to do in terms of preparation.

As of Postgres v6.3, two different styles of connections are supported. The site administrator will have chosen to allow TCP/IP network connections or will have restricted database access to local (same-machine) socket connections only. These choices become significant if you encounter problems in connecting to a database.

If you get the following error message from a Postgres command (such as psql or createdb):

```
% psql template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting
 connections
    at 'UNIX Socket' on port '5432'?
```

or

```
% psql -h localhost template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP
    (with -i) connections at 'localhost' on port '5432'?
```

it is usually because (1) the postmaster is not running, or (2) you are attempting to connect to the wrong server host. If you get the following error message:

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

it means that the site administrator started the postmaster as the wrong user. Tell him to restart it as the Postgres superuser.

3.3. Managing a Database

Now that Postgres is up and running we can create some databases to experiment with. Here, we describe the basic commands for managing a database.

Most Postgres applications assume that the database name, if not specified, is the same as the name on your computer account.

If your database administrator has set up your account without database creation privileges, then she should have told you what the name of your database is. If this is the case, then you can skip the sections on creating and destroying databases.

3.3.1. Creating a Database

Let's say you want to create a database named mydb. You can do this with the following command:

```
% createdb mydb
```

If you do not have the privileges required to create a database, you will see the following:

```
% createdb mydb
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 32 characters in length. Not every user has authorization to become a database administrator. If Postgres refuses to create databases for you, then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs.

3.3.2. Accessing a Database

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

- running the Postgres terminal monitor programs (e.g. psql) which allows you to interactively enter, edit, and execute SQL commands.
- writing a C program using the LIBPQ subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in libpq.

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

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: template1

mydb=>
```

This prompt indicates that the terminal monitor is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, “\” For example, you can get help on the syntax of various Postgres SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the “\g” is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say myFile, instead of entering them interactively, type:

```
mydb=> \i fileName
```

To get out of psql and return to UNIX, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type `\h` at the monitor prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by "--". Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by "/* ... */"

3.3.3. Destroying a Database

If you are the database administrator for the database mydb, you can destroy it using the following UNIX command:

```
% destroydb mydb
```

This action physically removes all of the UNIX files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 4. The Query Language

The Postgres query language is a variant of the SQL3 draft next-generation standard. It has many extensions such as an extensible type system, inheritance, functions and production rules. These are features carried over from the original Postgres query language, PostQuel. This section provides an overview of how to use Postgres SQL to perform simple operations. This manual is only intended to give you an idea of our flavor of SQL and is in no way a complete tutorial on SQL. Numerous books have been written on SQL, including [MELT93] and [DATE97]. You should be aware that some language features are not part of the ANSI standard.

4.1. Interactive Monitor

In the examples that follow, we assume that you have created the mydb database as described in the previous subsection and have started psql. Examples in this manual can also be found in `/usr/local/pgsql/src/tutorial/`. Refer to the `README` file in that directory for how to use them. To start the tutorial, do the following:

```
% cd /usr/local/pgsql/src/tutorial
% psql -s mydb
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: postgres

mydb=> \i basics.sql
```

The `\i` command read in queries from the specified files. The `-s` option puts you in single step mode which pauses before sending a query to the backend. Queries in this section are in the file `basics.sql`.

psql has a variety of `\d` commands for showing system information. Consult these commands for more details; for a listing, type `\?` at the psql prompt.

4.2. Concepts

The fundamental notion in Postgres is that of a class, which is a named collection of object instances. Each instance has the same collection of named attributes, and each attribute is of a specific type. Furthermore, each instance has a permanent *object identifier* (OID) that is unique throughout the installation. Because SQL syntax refers to tables, we will use the terms *table* and *class* interchangeably. Likewise, an SQL *row* is an *instance* and SQL *columns* are *attributes*. As previously discussed, classes are grouped into databases, and a collection of databases managed by a single postmaster process constitutes an installation or site.

4.3. Creating a New Class

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

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
```

```
    temp_hi         int,            -- high temperature
    prcp            real,           -- precipitation
    date            date
);
```

Note that both keywords and identifiers are case-insensitive; identifiers can become case-sensitive by surrounding them with double-quotes as allowed by SQL92. Postgres SQL supports the usual SQL types `int`, `float`, `real`, `smallint`, `char(N)`, `varchar(N)`, `date`, `time`, and `timestamp`, as well as other types of general utility and a rich set of geometric types. As we will see later, Postgres can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical keywords, except where required to support special cases in the SQL92 standard. So far, the Postgres create command looks exactly like the command used to create a table in a traditional relational system. However, we will presently see that classes have properties that are extensions of the relational model.

4.4. Populating a Class with Instances

The `insert` statement is used to populate a class with instances:

```
INSERT INTO weather
    VALUES ('San Francisco', 46, 50, 0.25, '11/27/1994')
```

You can also use the `copy` command to perform load large amounts of data from flat (ASCII) files.

4.5. Querying a Class

The weather class can be queried with normal relational selection and projection queries. A SQL `select` statement is used to do this. The statement is divided into a target list (the part that lists the attributes to be returned) and a qualification (the part that specifies any restrictions). For example, to retrieve all the rows of weather, type:

```
SELECT * FROM WEATHER;
```

and the output should be:

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
|San Francisco | 43      | 57      | 0    | 11-29-1994 |
+--------------+---------+---------+------+------------+
|Hayward       | 37      | 54      |      | 11-29-1994 |
+--------------+---------+---------+------+------------+
```

You may specify any arbitrary expressions in the target list. For example, you can do:

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

Arbitrary Boolean operators (`and`, `or` and `not`) are allowed in the qualification of any query. For example,

```
SELECT * FROM weather
    WHERE city = 'San Francisco'
```

```
    AND prcp > 0.0;

+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
```

As a final note, you can specify that the results of a select can be returned in a *sorted order* or with *duplicate instances* removed.

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

4.6. Redirecting SELECT Queries

Any select query can be redirected to a new class

```
SELECT * INTO TABLE temp FROM weather;
```

This forms an implicit `create` command, creating a new class temp with the attribute names and types specified in the target list of the `select into` command. We can then, of course, perform any operations on the resulting class that we can perform on other classes.

4.7. Joins Between Classes

Thus far, our queries have only accessed one class at a time. Queries can access multiple classes at once, or access the same class in such a way that multiple instances of the class are being processed at the same time. A query that accesses multiple instances of the same or different classes at one time is called a join query. As an example, say we wish to find all the records that are in the temperature range of other records. In effect, we need to compare the temp_lo and temp_hi attributes of each EMP instance to the temp_lo and temp_hi attributes of all other EMP instances.

Note

This is only a conceptual model. The actual join may be performed in a more efficient manner, but this is invisible to the user.

We can do this with the following query:

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

+--------------+-----+------+--------------+-----+------+
|city          | low | high | city         | low | high |
+--------------+-----+------+--------------+-----+------+
|San Francisco | 43  | 57   | San Francisco | 46  | 50   |
+--------------+-----+------+--------------+-----+------+
|San Francisco | 37  | 54   | San Francisco | 46  | 50   |
+--------------+-----+------+--------------+-----+------+
```

Note

The semantics of such a join are that the qualification is a truth expression defined for the Cartesian product of the classes indicated in the query. For those instances in the Cartesian product for which the qualification is true, Postgres computes and returns the values specified in the target list. Postgres SQL does not assign any meaning to duplicate values in such expressions. This means that Postgres sometimes recomputes the same target list several times; this frequently happens when Boolean expressions are connected with an "or". To remove such duplicates, you must use the `select distinct` statement.

In this case, both W1 and W2 are surrogates for an instance of the class weather, and both range over all instances of the class. (In the terminology of most database systems, W1 and W2 are known as *range variables*.) A query can contain an arbitrary number of class names and surrogates.

4.8. Updates

You can update existing instances using the update command. Suppose you discover the temperature readings are all off by 2 degrees as of Nov 28, you may update the data as follow:

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

4.9. Deletions

Deletions are performed using the `delete` command:

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

All weather recording belongs to Hayward is removed. One should be wary of queries of the form

```
DELETE FROM classname;
```

Without a qualification, `delete` will simply remove all instances of the given class, leaving it empty. The system will not request confirmation before doing this.

4.10. Using Aggregate Functions

Like most other query languages, PostgreSQL supports aggregate functions. The current implementation of Postgres aggregate functions have some limitations. Specifically, while there are aggregates to compute such functions as the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of instances, aggregates can only appear in the target list of a query and not directly in the qualification (the where clause). As an example,

```
SELECT max(temp_lo) FROM weather;
```

is allowed, while

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

is not. However, as is often the case the query can be restated to accomplish the intended result; here by using a *subselect*:

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

Aggregates may also have *group by* clauses:

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

Chapter 5. Advanced Postgres SQL Features

Having covered the basics of using Postgres SQL to access your data, we will now discuss those features of Postgres that distinguish it from conventional data managers. These features include inheritance, time travel and non-atomic data values (array- and set-valued attributes). Examples in this section can also be found in `advance.sql` in the tutorial directory. (Refer to Chapter 4, *The Query Language* for how to use it.)

5.1. Inheritance

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

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

CREATE TABLE capitals (
    state           char2
) INHERITS (cities);
```

In this case, an instance of capitals *inherits* all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is `text`, a native Postgres type for variable length ASCII strings. The type of the attribute population is `float`, a native Postgres type for double precision floating point numbers. State capitals have an extra attribute, state, that shows their state. In Postgres, a class can inherit from zero or more other classes, and a query can reference either all instances of a class or all instances of a class plus all of its descendants.

Note

The inheritance hierarchy is a directed acyclic graph.

For example, the following query finds all the cities that are situated at an attitude of 500ft or higher:

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

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

On the other hand, to find the names of all cities, including state capitals, that are located at an altitude over 500ft, the query is:

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

which returns:

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

Here the "*" after cities indicates that the query should be run over cities and all classes below cities in the inheritance hierarchy. Many of the commands that we have already discussed (`select`, `update` and `delete`) support this "*" notation, as do others, like `alter`.

5.2. Non-Atomic Values

One of the tenets of the relational model is that the attributes of a relation are atomic. Postgres does not have this restriction; attributes can themselves contain sub-values that can be accessed from the query language. For example, you can create attributes that are arrays of base types.

5.2.1. Arrays

Postgres allows attributes of an instance to be defined as fixed-length or variable-length multi-dimensional arrays. Arrays of any base type or user-defined type can be created. To illustrate their use, we first create a class with arrays of base types.

```
CREATE TABLE SAL_EMP (
    name            text,
    pay_by_quarter  int4[],
    schedule        text[][]
);
```

The above query will create a class named SAL_EMP with a *text* string (name), a one-dimensional array of *int4* (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 *INSERTS*s; note that when appending to an array, we enclose the values within braces and separate them by commas. If you know *C*, this is not unlike the syntax for initializing structures.

```
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"}}');
```

By default, Postgres uses the "one-based" numbering convention for arrays -- that is, an array of n elements starts with array[1] and ends with array[n]. 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 SAL_EMP.pay_by_quarter[1] <>
    SAL_EMP.pay_by_quarter[2];

+------+
|name  |
+------+
|Carol |
+------+
```

This query retrieves the third quarter pay of all employees:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;


+---------------+
|pay_by_quarter |
+---------------+
|10000          |
+---------------+
|25000          |
+---------------+
```

We can also access arbitrary slices of an array, or subarrays. This query retrieves the first item on Bill's schedule for the first two days of the week.

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

+-------------------+
|schedule           |
+-------------------+
|{{"meeting"},{""}} |
+-------------------+
```

5.3. Time Travel

As of Postgres v6.2, *time travel is no longer supported*. There are several reasons for this: performance impact, storage size, and a pg_time file which grows toward infinite size in a short period of time.

New features such as triggers allow one to mimic the behavior of time travel when desired, without incurring the overhead when it is not needed (for most users, this is most of the time). See examples in the `contrib` directory for more information.

Time travel is deprecated

The remaining text in this section is retained only until it can be rewritten in the context of new techniques to accomplish the same purpose. Volunteers? - thomas 1998-01-12

Postgres supports the notion of time travel. This feature allows a user to run historical queries. For example, to find the current population of Mariposa city, one would query:

```
SELECT * FROM cities WHERE name = 'Mariposa';

+---------+------------+----------+
|name     | population | altitude |
+---------+------------+----------+
|Mariposa | 1320       | 1953     |
+---------+------------+----------+
```

Postgres will automatically find the version of Mariposa's record valid at the current time. One can also give a time range. For example to see the past and present populations of Mariposa, one would query:

```
SELECT name, population
    FROM cities['epoch', 'now']
    WHERE name = 'Mariposa';
```

where "epoch" indicates the beginning of the system clock.

Note

On UNIX systems, this is always midnight, January 1, 1970 GMT.

If you have executed all of the examples so far, then the above query returns:

```
+---------+------------+
|name     | population |
+---------+------------+
|Mariposa | 1200       |
+---------+------------+
|Mariposa | 1320       |
+---------+------------+
```

The default beginning of a time range is the earliest time representable by the system and the default end is the current time; thus, the above time range can be abbreviated as ``[,]."

5.4. More Advanced Features

Postgres has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These are discussed in more detail in both the User's and Programmer's Guides.

Part II. User's Guide

Information for users.

Table of Contents

Chapter 6. Setting Up Your Environment

This section discusses how to set up your own environment so that you can use frontend applications. We assume Postgres has already been successfully installed and started; refer to the Administrator's Guide and the installation notes for how to install Postgres.

Postgres is a client/server application. As a user, you only need access to the client portions of the installation (an example of a client application is the interactive monitor psql). For simplicity, we will assume that Postgres has been installed in the directory `/usr/local/pgsql`. Therefore, wherever you see the directory `/usr/local/pgsql` you should substitute the name of the directory where Postgres is actually installed. All Postgres commands are installed in the directory `/usr/local/pgsql/bin`. Therefore, you should add this directory to your shell command path. If you use a variant of the Berkeley C shell, such as csh or tcsh, you would add

```
set path = ( /usr/local/pgsql/bin path )
```

in the `.login` file in your home directory. If you use a variant of the Bourne shell, such as sh, ksh, or bash, then you would add

```
PATH=/usr/local/pgsql/bin PATH
export PATH
```

to the `.profile` file in your home directory. From now on, we will assume that you have added the Postgres bin directory to your path. In addition, we will make frequent reference to “setting a shell variable” or “setting an environment variable” throughout this document. If you did not fully understand the last paragraph on modifying your search path, you should consult the UNIX manual pages that describe your shell before going any further.

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

Chapter 7. Managing a Database

Note

This section is currently a thinly disguised copy of the Tutorial. Needs to be augmented. - thomas 1998-01-12

Although the *site administrator* is responsible for overall management of the Postgres installation, some databases within the installation may be managed by another person, designated the *database administrator*. This assignment of responsibilities occurs when a database is created. A user may be assigned explicit privileges to create databases and/or to create new users. A user assigned both privileges can perform most administrative task within Postgres, but will not by default have the same operating system privileges as the site administrator.

The Database Administrator's Guide covers these topics in more detail.

7.1. Database Creation

Databases are created by the `create database` issued from within Postgres. createdb is a command-line utility provided to give the same functionality from outside Postgres.

The Postgres backend must be running for either method to succeed, and the user issuing the command must be the Postgres *superuser* or have been assigned database creation privileges by the superuser.

To create a new database named “mydb” from the command line, type

```
% createdb mydb
```

and to do the same from within psql type

```
* CREATE DATABASE mydb;
```

If you do not have the privileges required to create a database, you will see the following:

```
% createdb mydb
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 32 characters in length.

7.2. Alternate Database Locations

It is possible to create a database in a location other than the default location for the installation. Remember that all database access actually occurs through the database backend, so that any location specified must be accessible by the backend.

Either an absolute path name or an environment variable may be specified as a location. Any environment variable specifying an alternate location must have been defined before the backend was started. Consult with the site administrator regarding preconfigured alternate database locations.

Note

The environment variable style of specification is to be preferred since it allows the site administrator more flexibility in managing disk storage.

For security and integrity reasons, any path or environment variable specified has some additional path fields appended.

Alternate database locations must be prepared by running initlocation.

To create a data storage area in `/alt/postgres/data`, ensure that `/alt/postgres` already exists. From the command line, type

```
% initlocation /alt/postgres/data
Creating Postgres database system directory /alt/postgres/data

Creating Postgres database system directory /alt/postgres/data/base
```

To do the same using an environment variable PGDATA2, type

```
% initlocation $PGDATA2
Creating Postgres database system directory /alt/postgres/data

Creating Postgres database system directory /alt/postgres/data/base
```

To create a database in the alternate storage area `/alt/postgres/data` from the command line, type

```
% createdb -D /alt/postgres/data mydb
```

or

```
% createdb -D PGDATA2 mydb
```

and to do the same from within psql type

```
* CREATE DATABASE mydb WITH LOCATION = 'PGDATA2';
```

If you do not have the privileges required to create a database, you will see the following:

```
% createdb mydb
WARN:user "your username" is not allowed to create/destroy databases
createdb: database creation failed on mydb.
```

If the specified location does not exist or the database backend does not have permission to access it or to write to directories under it, you will see the following:

```
% createdb -D /alt/postgres/data mydb
ERROR:  Unable to create database directory /alt/postgres/data/base/
mydb
createdb: database creation failed on mydb.
```

7.3. Accessing a Database

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

- running the Postgres terminal monitor programs (e.g. psql) which allows you to interactively enter, edit, and execute SQL commands.

- writing a C program using the LIBPQ subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in section ??.

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

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: template1

mydb=>
```

This prompt indicates that the terminal monitor is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, "\" For example, you can get help on the syntax of various Postgres SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the "\g" is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say myFile, instead of entering them interactively, type:

```
mydb=> \i fileName
```

To get out of psql and return to UNIX, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type `\h` at the monitor prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by "--". Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by "/* ... */"

7.3.1. Database Privileges

7.3.2. Table Privileges

TBD

7.4. Destroying a Database

If you are the database administrator for the database mydb, you can destroy it using the following UNIX command:

```
% destroydb mydb
```

This action physically removes all of the UNIX files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 8. Data Types

Describes the built-in data types available in Postgres.

Postgres has a rich set of native data types available to users. Users may add new types to Postgres using the `define type` command described elsewhere.

In the context of data types, the following sections will discuss SQL standards compliance, porting issues, and usage. Some Postgres types correspond directly to SQL92-compatible types. In other cases, data types defined by SQL92 syntax are mapped directly into native Postgres types. Many of the built-in types have obvious external formats. However, several types are either unique to Postgres, such as open and closed paths, or have several possibilities for formats, such as date and time types.

Table 8.1. Postgres Data Types

Postgres Type	SQL92 or SQL3 Type	Description
bool	boolean	logical boolean (true/false)
box		rectangular box in 2D plane
char(n)	character(n)	fixed-length character string
cidr		IP version 4 network or host address
circle		circle in 2D plane
date	date	calendar date without time of day
float4/8	float(p)	floating-point number with precision p
float8	real, double precision	double-precision floating-point number
inet		IP version 4 network or host address
int2	smallint	signed two-byte integer
int4	int, integer	signed 4-byte integer
int4	decimal(p,s)	exact numeric for p <= 9, s = 0
int4	numeric(p,s)	exact numeric for p == 9, s = 0
int8		signed 8-byte integer
line		infinite line in 2D plane
lseg		line segment in 2D plane
money	decimal(9,2)	US-style currency
path		open and closed geometric path in 2D plane
point		geometric point in 2D plane
polygon		closed geometric path in 2D plane
serial		unique id for indexing and cross-reference
time	time	time of day

Postgres Type	SQL92 or SQL3 Type	Description
timespan	interval	general-use time span
timestamp	timestamp with time zone	date/time
varchar(n)	character varying(n)	variable-length character string

Note

The `cidr` and `inet` types are designed to handle any IP type but only ipv4 is handled in the current implementation. Everything here that talks about ipv4 will apply to ipv6 in a future release.

Table 8.2. Postgres Function Constants

Postgres Function	SQL92 Constant	Description
getpgusername()	current_user	user name in current session
date('now')	current_date	date of current transaction
time('now')	current_time	time of current transaction
timestamp('now')	current_timestamp	date and time of current transaction

Postgres has features at the forefront of ORDBMS development. In addition to SQL3 conformance, substantial portions of SQL92 are also supported. Although we strive for SQL92 compliance, there are some aspects of the standard which are ill considered and which should not live through subsequent standards. Postgres will not make great efforts to conform to these features; however, these tend to apply in little-used or obsure cases, and a typical user is not likely to run into them.

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 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.

Note that 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.

Note

The original Postgres v4.2 code received from Berkeley rounded all double precision floating point results to six digits for output. Starting with v6.1, floating point numbers are allowed to retain most of the intrinsic precision of the type (typically 15 digits for doubles, 6 digits for 4-byte floats). Other types with underlying floating point fields (e.g. geometric types) carry similar precision.

8.1. Numeric Types

Numeric types consist of two- and four-byte integers and four- and eight-byte floating point numbers.

Table 8.3. Postgres Numeric Types

Numeric Type	Storage	Description	Range
float4	4 bytes	Variable-precision	6 decimal places
float8	8 bytes	Variable-precision	15 decimal places

Numeric Type	Storage	Description	Range
int2	2 bytes	Fixed-precision	-32768 to +32767
int4	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
int8	8 bytes	Very large range fixed-precision	+/- > 18 decimal places
serial	4 bytes	Identifer or cross-reference	0 to +2147483647

The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Numerical Operators and Mathematical Functions for more information.

The `serial` type is a special-case type constructed by Postgres from other existing components. It is typically used to create unique identifiers for table entries. In the current implementation, specifying

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

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename
    (colname INT4 DEFAULT nextval('tablename_colname_seq');
CREATE UNIQUE INDEX tablename_colname_key on tablename (colname);
```

Caution

The implicit sequence created for the `serial` type will *not* be automatically removed when the table is dropped. So, the following commands executed in order will likely fail:

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

The sequence will remain in the database until explicitly dropped using `DROP SEQUENCE`.

The *exact numerics* `decimal` and `numeric` have fully implemented syntax but currently (Postgres v6.4) support only a small range of precision and/or range values. The `int8` type may not be available on all platforms since it relies on compiler support for this.

8.2. Monetary Type

The `money` type supports US-style currency with fixed decimal point representation. If Postgres is compiled with USE_LOCALE then the money type should use the monetary conventions defined for *locale(7)*.

Table 8.4. Postgres Monetary Types

Monetary Type	Storage	Description	Range
money	4 bytes	Fixed-precision	-21474836.48 to +21474836.47

`numeric` should eventually replace the money type. It has a fully implemented syntax but currently (Postgres v6.4) support only a small range of precision and/or range values and cannot adequately substitute for the money type.

8.3. Character Types

SQL92 defines two primary character types: `char` and `varchar`. Postgres supports these types, in addition to the more general `text` type, which unlike `varchar` does not require an upper limit to be declared on the size of the field.

Table 8.5. Postgres Character Types

Character Type	Storage	Recommendation	Description
char	1 byte	SQL92-compatible	Single character
char(n)	(4+n) bytes	SQL92-compatible	Fixed-length blank padded
text	(4+x) bytes	Best choice	Variable-length
varchar(n)	(4+n) bytes	SQL92-compatible	Variable-length with limit

There is one other fixed-length character type. The `name` type only has one purpose and that is to provide Postgres with a special type to use for internal names. It is not intended for use by the general user. It's length is currently defined as 32 chars but should be reference using NAMEDATALEN. This is set at compile time and may change in a future release.

Table 8.6. Postgres Specialty Character Type

Character Type	Storage	Description
name	32 bytes	Thirty-two character internal type

8.4. Date/Time Types

There are two fundamental kinds of date and time measurements: absolute clock times and relative time intervals. Both quantities should demonstrate continuity and smoothness, as does time itself. Postgres supplies two primary user-oriented date and time types, `datetime` and `timespan`, as well as the related SQL92 types `timestamp`, `interval`, `date` and `time`.

In a future release, `datetime` and `timespan` are likely to merge with the SQL92 types `timestamp`, `interval`. Other date and time types are also available, mostly for historical reasons.

Table 8.7. Postgres Date/Time Types

Date/Time Type	Storage	Recommendation	Description
abstime	4 bytes	original date and time	limited range
date	4 bytes	SQL92 type	wide range
datetime	8 bytes	best general date and time	wide range, high precision
interval	12 bytes	SQL92 type	equivalent to timespan
reltime	4 bytes	original time interval	limited range, low precision
time	4 bytes	SQL92 type	wide range
timespan	12 bytes	best general time interval	wide range, high precision

Date/Time Type	Storage	Recommendation	Description
timestamp	4 bytes	SQL92 type	limited range

`timestamp` is currently implemented separately from `datetime`, although they share input and output routines.

Table 8.8. Postgres Date/Time Ranges

Date/Time Type	Earliest	Latest	Resolution
abstime	1901-12-14	2038-01-19	1 sec
date	4713 BC	32767 AD	1 day
datetime	4713 BC	1465001 AD	1 microsec to 14 digits
interval	-178000000 years	178000000 years	1 microsec
reltime	-68 years	+68 years	1 sec
time	00:00:00.00	23:59:59.99	1 microsec
timespan	-178000000 years	178000000 years	1 microsec (14 digits)
timestamp	1901-12-14	2038-01-19	1 sec

Postgres endevours to be compatible with SQL92 definitions for typical usage. The SQL92 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 or does.
- The default time zone is specified as a constant integer offset from GMT/UTC.

However, 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 savings time boundaries.

To address these difficulties, Postgres associates time zones only with date and time types which contain both date and time, and assumes local time 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 savings time and other expected behavior.

In future releases, the number of date/time types will decrease, with the current implementation of `datetime` becoming `timestamp`, `timespan` becoming `interval`, and (possibly) `abstime` and `reltime` being deprecated in favor of `timestamp` and `interval`. The more arcane features of the date/time definitions from the SQL92 standard are not likely to be pursued.

8.4.1. Date/Time Styles

Output formats can be set to one of four styles: ISO-8601, SQL (Ingres), traditional Postgres, and German.

Table 8.9. Postgres Date Styles

Style Specification	Description	Example
ISO	ISO-8601 standard	1997-12-17 07:37:16-08
SQL	Traditional style	12/17/1997 07:37:16.00 PST
Postgres	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 (US) variants, which determines whether month follows day or vica versa.

Table 8.10. Postgres Date Order Conventions

Style Specification	Description	Example
European	Regional convention	17/12/1997 15:37:16.00 MET
NonEuropean	Regional convention	12/17/1997 07:37:16.00 PST
US	Regional convention	12/17/1997 07:37:16.00 PST

There are several ways to affect the appearance of date/time types:

- The PGDATESTYLE environment variable used by the backend directly on postmaster startup.
- The PGDATESTYLE environment variable used by the frontend libpq on session startup.
- SET DateStyle SQL command.

For Postgres v6.4 (and earlier) the default date/time style is "non-European traditional Postgres". In future releases, the default may become ISO-8601, which alleviates date specification ambiguities and Y2K collation problems.

8.4.2. Time Zones

Postgres obtains time zone support from the underlying operating system. All dates and times are stored internally in Universal Coordinated Time (UTC), alternately 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 affect the time zone behavior:

- The TZ environment variable used by the backend directly on postmaster startup as the default time zone.
- The PGTZ environment variable set at the client used by libpq to send time zone information to the backend upon connection.
- The SQL command `SET TIME ZONE` sets the time zone for the session.

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

8.4.3. Date/Time Input

General-use date and time is input using a wide range of styles, including ISO-compatible, SQL-compatible, traditional Postgres and other permutations of date and time. In cases where interpretation can be ambiguous (quite possible with many traditional styles of date specification) Postgres uses a style setting to resolve the ambiguity.

Most date and time types share code for data input. For those types the input can have any of a wide variety of styles. For numeric date representations, European and US conventions can differ, and the proper interpretation is obtained by using the `set datestyle` command before entering data. Note that the style setting does not preclude use of various styles for input; it is used primarily to determine the output style and to resolve ambiguities.

The special values `current', `infinity' and `-infinity' are provided. `infinity' specifies a time later than any other valid time, and `-infinity' specifies a time earlier than any other valid time. `current' indicates that the current time should be substituted whenever this value appears in a computation. The strings `now', `today', `yesterday', `tomorrow', and `epoch' can be used to specify time values. `now' means the current

transaction time, and differs from `current' in that the current time is immediately substituted for it. `epoch' means Jan 1 00:00:00 1970 GMT.

Table 8.11. Postgres Date/Time Special Constants

Constant	Description
current	Current transaction time, deferred
epoch	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	Later than other valid times
-infinity	Earlier than other valid times
invalid	Illegal entry
now	Current transaction time
today	Midnight today
tomorrow	Midnight tomorrow
yesterday	Midnight yesterday

8.4.4. datetime

General-use date and time is input using a wide range of styles, including ISO-compatible, SQL-compatible, traditional Postgres (see section on "absolute time") and other permutations of date and time. Output styles can be ISO-compatible, SQL-compatible, or traditional Postgres, with the default set to be compatible with Postgres v6.0.

`datetime` is specified using the following syntax:

```
Year-Month-Day [ Hour : Minute : Second ]      [AD,BC] [ Timezone ]
  YearMonthDay [ Hour : Minute : Second ]      [AD,BC] [ Timezone ]
     Month Day [ Hour : Minute : Second ] Year [AD,BC] [ Timezone ]
where
    Year is 4013 BC, ..., very large
    Month is Jan, Feb, ..., Dec or 1, 2, ..., 12
    Day is 1, 2, ..., 31
    Hour is 00, 02, ..., 23
    Minute is 00, 01, ..., 59
    Second is 00, 01, ..., 59 (60 for leap second)
    Timezone is 3 characters or ISO offset to GMT
```

Valid dates are from Nov 13 00:00:00 4013 BC GMT to far into the future. Timezones are either three characters (e.g. "GMT" or "PST") or ISO-compatible offsets to GMT (e.g. "-08" or "-08:00" when in Pacific Standard Time). Dates are stored internally in Greenwich Mean Time. Input and output routines translate time to the local time zone of the server.

8.4.5. `timespan`

General-use time span is input using a wide range of syntaxes, including ISO-compatible, SQL-compatible, traditional Postgres (see section on "relative time") and other permutations of time span. Output formats can be ISO-compatible, SQL-compatible, or traditional Postgres, with the default set to be Postgres-compatible. Months and years are a "qualitative" time interval, and are stored separately from the other "quantitative" time intervals such as day or hour. For date arithmetic, the qualitative time units are instantiated in the context of the relevant date or time.

Time span is specified with the following syntax:

```
  Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Direction]
where
    Quantity is ..., `-1', `0', `1', `2', ...
    Unit is `second', `minute', `hour', `day', `week', `month',
 `year',
    'decade', 'century', millenium', or abbreviations or plurals of
 these units.
    Direction is `ago'.
```

8.4.6. abstime

Absolute time (`abstime`) is a limited-range (+/- 68 years) and limited-precision (1 sec) date data type. `datetime` may be preferred, since it covers a larger range with greater precision.

Absolute time is specified using the following syntax:

```
Month  Day [ Hour : Minute : Second ]  Year [ Timezone ]
where
    Month is Jan, Feb, ..., Dec
    Day is 1, 2, ..., 31
    Hour is 01, 02, ..., 24
    Minute is 00, 01, ..., 59
    Second is 00, 01, ..., 59
    Year is 1901, 1902, ..., 2038
```

Valid dates are from Dec 13 20:45:53 1901 GMT to Jan 19 03:14:04 2038 GMT.

Historical Note

As of Version 3.0, times are no longer read and written using Greenwich Mean Time; the input and output routines default to the local time zone.

All special values allowed for `datetime` are also allowed for "absolute time".

8.4.7. reltime

Relative time `reltime` is a limited-range (+/- 68 years) and limited-precision (1 sec) time span data type. `timespan` should be preferred, since it covers a larger range with greater precision and, more importantly, can distinguish between relative units (months and years) and quantitative units (days, hours, etc). Instead, reltime must force months to be exactly 30 days, so time arithmetic does not always work as expected. For example, adding one reltime year to abstime today does not produce today's date one year from now, but rather a date 360 days from today.

`reltime` shares input and output routines with the other time span types. The section on `timespan` covers this in more detail.

8.4.8. `timestamp`

This is currently a limited-range absolute time which closely resembles the abstime data type. It shares the general input parser with the other date/time types. In future releases this type will absorb the capabilities of the `datetime` type and will move toward SQL92 compliance.

`timestamp` is specified using the same syntax as for `datetime`.

8.4.9. `interval`

`interval` is an SQL92 data type which is currently mapped to the `timespan` Postgres data type.

8.4.10. tinterval

Time ranges are specified as:

```
[ 'abstime' 'abstime']
where
    abstime is a time in the absolute time format.
```

Special abstime values such as `current', `infinity' and `-infinity' can be used.

8.5. Boolean Type

Postgres supports `bool` as the SQL3 boolean type. `bool` can have one of only two states: 'true' or 'false'. A third state, 'unknown', is not implemented and is not suggested in SQL3; NULL is an effective substitute. `bool` can be used in any boolean expression, and boolean expressions always evaluate to a result compatible with this type.

`bool` uses 4 bytes of storage.

Table 8.12. Postgres Boolean Type

State	Output	Input
True	't'	TRUE, 't', 'true', 'y', 'yes', '1'
False	'f'	FALSE, 'f', 'false', 'n', 'no', '0'

8.6. Geometric Types

Geometric types represent two-dimensional spatial objects. The most fundamental type, the point, forms the basis for all of the other types.

Table 8.13. Postgres Geometric Types

Geometric Type	Storage	Representation	Description
point	16 bytes	(x,y)	Point in space
line	32 bytes	((x1,y1),(x2,y2))	Infinite line
lseg	32 bytes	((x1,y1),(x2,y2))	Finite line segment
box	32 bytes	((x1,y1),(x2,y2))	Rectangular box
path	4+32n bytes	((x1,y1),...)	Closed path (similar to polygon)
path	4+32n bytes	[(x1,y1),...]	Open path
polygon	4+32n bytes	((x1,y1),...)	Polygon (similar to closed path)

Geometric Type	Storage	Representation	Description
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.

8.6.1. Point

Points are specified using the following syntax:

```
( x , y )
  x , y
where
    x is the x-axis coordinate as a floating point number
    y is the y-axis coordinate as a floating point number
```

8.6.2. 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
    (x1,y1) and (x2,y2) are the endpoints of the segment
```

8.6.3. Box

Boxes are represented by pairs of points which 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
    (x1,y1) and (x2,y2) are opposite corners
```

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

8.6.4. 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 select 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
    (x1,y1),...,(xn,yn) are points 1 through n
    a leading "[" indicates an open path
    a leading "(" indicates a closed path
```

Paths are output using the first syntax. Note that Postgres versions prior to v6.1 used a format for paths which had a single leading parenthesis, a "closed" flag, an integer count of the number of points, then the list of points followed by a closing parenthesis. The built-in function `upgradepath` is supplied to convert paths dumped and reloaded from pre-v6.1 databases.

8.6.5. 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
    (x1,y1),...,(xn,yn) are points 1 through n
```

Polygons are output using the first syntax. Note that Postgres versions prior to v6.1 used a format for polygons which had a single leading parenthesis, the list of x-axis coordinates, the list of y-axis coordinates, followed by a closing parenthesis. The built-in function `upgradepoly` is supplied to convert polygons dumped and reloaded from pre-v6.1 databases.

8.6.6. 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
    (x,y) is the center of the circle
    r is the radius of the circle
```

Circles are output using the first syntax.

8.7. IP Version 4 Networks and Host Addresses

The `cidr` type stores networks specified in CIDR notation. The `inet` type stores hosts and networks in CIDR notation.

Table 8.14. PostgresIP Version 4 Types

IPV4 Type	Storage	Description	Range
cidr	variable	CIDR networks	Valid IPV4 CIDR blocks
inet	variable	nets and hosts	Valid IPV4 CIDR blocks

8.7.1. `inet` for IP Networks

The cidr type holds a CIDR network. The format for specifying 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 the "/y" part is left off, it is calculated using assumptions from the old class system except that it is extended to include at least all of the octets in the input. Here are some examples:

Table 8.15. PostgresIP Types Examples

CIDR Input	CIDR Displayed
192.168.1	192.168.1/24
192.168	192.168.0/24
128.1	128.1/16
128	128.0/16
128.1.2	128.1.2/24
10.1.2	10.1.2/24
10.1	10.1/16
10	10/8

8.7.2. `inet` for IP Networks

The inet type is designed to hold, in one field, all of the information about a host including the CIDR style subnet that it is in. Note that if you want to store proper CIDR networks, see the cidr type. The inet type is similar to the cidr type except that the bits in the host part can be non-zero. Functions exist to extract the various elements of the field.

The input format for this function is "x.x.x.x/y" where "x.x.x.x" is an internet host and y is the number of bits in the netmask. If the "/y" part is left off, it is treated as "/32." On output, the "/y" part is not printed if it is /32. This allows the type to be used as a straight host type by just leaving of the bits part.

Chapter 9. Operators

Describes the built-in operators available in Postgres.

Postgres provides a large number of built-in operators on system types. These operators are declared in the system catalog pg_operator. Every entry in pg_operator includes the name of the procedure that implements the operator and the class OIDs of the input and output types.

To view all variations of the "||" string concatenation operator, try

```
    SELECT oprleft, oprright, oprresult, oprcode
    FROM pg_operator WHERE oprname = '||';

oprleft|oprright|oprresult|oprcode
-------+--------+---------+-------
     25|      25|       25|textcat
   1042|    1042|     1042|textcat
   1043|    1043|     1043|textcat
(3 rows)
```

Users may invoke operators using the operator name, as in:

```
select * from emp where salary < 40000;
```

Alternatively, users may call the functions that implement the operators directly. In this case, the query above would be expressed as:

```
select * from emp where int4lt(salary, 40000);
```

psql has a command (`\dd`) to show these operators.

9.1. Lexical Precedence

Operators have a precedence which is currently hardcoded into the parser. Most operators have the same precedence and are non-associative. This may lead to non-intuitive behavior; for example the boolean operators "<" and ">" have a different precedence that the boolean operators "<=" and ">=".

Table 9.1. Operator Ordering (decreasing precedence)

Element	Precedence	Description
UNION	left	SQL select construct
::		Postgres typecasting
[ ]	left	array delimiters
.	left	table/column delimiter
-	right	unary minus
;	left	statement termination, logarithm
:	right	exponentiation
|	left	start of interval

Element	Precedence	Description
* /	left	multiplication, division
+ -	left	addition, subtraction
IS		test for TRUE, FALSE, NULL
ISNULL		test for NULL
NOTNULL		test for NOT NULL
(all other operators)		native and user-defined
IN		set membership
BETWEEN		containment
LIKE		string pattern matching
< >		boolean inequality
=	right	equality
NOT	right	negation
AND	left	logical intersection
OR	left	logical union

9.2. General Operators

The operators listed here are defined for a number of native data types, ranging from numeric types to data/time types.

Table 9.2. Postgres Operators

Operator	Description	Usage
<	Less than?	1 < 2
<=	Less than or equal to?	1 <= 2
<>	Not equal?	1 <> 2
=	Equal?	1 = 1
>	Greater than?	2 > 1
>=	Greater than or equal to?	2 >= 1
||	Concatenate strings	'Postgre' || 'SQL'
!!=	NOT IN	3 !!= i
~~	LIKE	'scrappy,marc,hermit' ~~ '%scrappy%'
!~~	NOT LIKE	'bruce' !~~ '%al%'
~	Match (regex), case sensitive	'thomas' ~ '.*thomas.*'
~*	Match (regex), case insensitive	'thomas' ~* '.*Thomas.*'
!~	Does not match (regex), case sensitive	'thomas' !~ '.*Thomas.*'
!~*	Does not match (regex), case insensitive	'thomas' !~ '.*vadim.*'

9.3. Numerical Operators

Table 9.3. Postgres Numerical Operators

Operator	Description	Usage
!	Factorial	3 !
!!	Factorial (left operator)	!! 3
%	Modulo	5 % 4
%	Truncate	% 4.5
*	Multiplication	2 * 3
+	Addition	2 + 3
-	Subtraction	2 - 3
/	Division	4 / 2
:	Natural Exponentiation	: 3.0
;	Natural Logarithm	(; 5.0)
@	Absolute value	@ -5.0
^	Exponentiation	2.0 ^ 3.0
|/	Square root	|/ 25.0
||/	Cube root	||/ 27.0

9.4. Geometric Operators

Table 9.4. Postgres Geometric Operators

Operator	Description	Usage
+	Translation	'((0,0),(1,1))'::box + '(2.0,0)'::point
-	Translation	'((0,0),(1,1))'::box - '(2.0,0)'::point
*	Scaling/rotation	'((0,0),(1,1))'::box * '(2.0,0)'::point
/	Scaling/rotation	'((0,0),(2,2))'::box / '(2.0,0)'::point
#	Intersection	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Number of points in polygon	# '((1,0),(0,1),(-1,0))'
##	Point of closest proximity	'(0,0)'::point ## '((2,0),(0,2))'::lseg
&&	Overlaps?	'((0,0),(1,1))'::box && '((0,0),(2,2))'::box
&<	Overlaps to left?	'((0,0),(1,1))'::box &< '((0,0),(2,2))'::box
&>	Overlaps to right?	'((0,0),(3,3))'::box &> '((0,0),(2,2))'::box
<->	Distance between	'((0,0),1)'::circle <-> '((5,0),1)'::circle

Operator	Description	Usage
<<	Left of?	'((0,0),1)'::circle << '((5,0),1)'::circle
<^	Is below?	'((0,0),1)'::circle <^ '((0,5),1)'::circle
>>	Is right of?	'((5,0),1)'::circle >> '((0,0),1)'::circle
>^	Is above?	'((0,5),1)'::circle >^ '((0,0),1)'::circle
?#	Intersects or overlaps	'((-1,0),(1,0))'::lseg ?# '((-2,-2),(2,2))'::box;
?-	Is horizontal?	'(1,0)'::point ?- '(0,0)'::point
?-|	Is perpendicular?	'((0,0),(0,1))'::lseg ?-| '((0,0),(1,0))'::lseg
@-@	Length or circumference	@-@ '((0,0),(1,0))'::path
?|	Is vertical?	'(0,1)'::point ?| '(0,0)'::point
?||	Is parallel?	'((-1,0),(1,0))'::lseg ?|| '((-1,2),(1,2))'::lseg
@	Contained or on	'(1,1)'::point @ '((0,0),2)'::circle
@@	Center of	@@ '((0,0),10)'::circle
~=	Same as	'((0,0),(1,1))'::polygon ~= '((1,1),(0,0))'::polygon

9.5. Time Interval Operators

The time interval data type `tinterval` is a legacy from the original date/time types and is not as well supported as the more modern types. There are several operators for this type.

Table 9.5. Postgres Time Interval Operators

Operator	Description	Usage
#<	Interval less than?	
#<=	Interval less than or equal to?	
#<>	Interval not equal?	
#=	Interval equal?	
#>	Interval greater than?	
#>=	Interval greater than or equal to?	
<#>	Convert to time interval	
<<	Interval less than?	
|	Start of interval	
~=	Same as	
<?>	Time inside interval?	

9.6. IP V4 Operators

Table 9.6. PostgresIP V4 Operators

Operator	Description	Usage
<	Less than	'192.168.1.5'::cidr < '192.168.1.6'::cidr
<=	Less than or equal	'192.168.1.5'::cidr <= '192.168.1.5'::cidr
=	Equals	'192.168.1.5'::cidr = '192.168.1.5'::cidr
%gt;=	Greater or equal	'192.168.1.5'::cidr >= '192.168.1.5'::cidr
%gt;	Greater	'192.168.1.5'::cidr %gt; '192.168.1.4'::cidr
<>	Not equal	'192.168.1.5'::cidr <> '192.168.1.4'::cidr
<<	is contained within	'192.168.1.5'::cidr << '192.168.1/24'::cidr
<<=	is contained within or equals	'192.168.1/24'::cidr <<= '192.168.1/24'::cidr
>>	contains	'192.168.1/24'::cidr >> '192.168.1.5'::cidr
>>=	contains or equals	'192.168.1/24'::cidr >>= '192.168.1/24'::cidr

9.7. IP V4 Operators

Table 9.7. PostgresIP V4 Operators

Operator	Description	Usage
<	Less than	'192.168.1.5'::inet < '192.168.1.6'::inet
<=	Less than or equal	'192.168.1.5'::inet <= '192.168.1.5'::inet
=	Equals	'192.168.1.5'::inet = '192.168.1.5'::inet
%gt;=	Greater or equal	'192.168.1.5'::inet >= '192.168.1.5'::inet
%gt;	Greater	'192.168.1.5'::inet %gt; '192.168.1.4'::inet
<>	Not equal	'192.168.1.5'::inet <> '192.168.1.4'::inet
<<	is contained within	'192.168.1.5'::inet << '192.168.1/24'::inet

Operator	Description	Usage
<<=	is contained within or equals	'192.168.1/24'::inet <<= '192.168.1/24'::inet
>>	contains	'192.168.1/24'::inet >> '192.168.1.5'::inet
>>=	contains or equals	'192.168.1/24'::inet >>= '192.168.1/24'::inet

Chapter 10. Functions

Describes the built-in functions available in Postgres.

Many data types have functions available for conversion to other related types. In addition, there are some type-specific functions. Some functions are also available through operators and may be documented as operators only.

10.1. Mathematical Functions

Table 10.1. Mathematical Functions

Function	Returns	Description	Example
dexp(float8)	float8	raise e to the specified exponent	dexp(2.0)
dpow(float8,float8)	float8	raise a number to the specified exponent	dpow(2.0, 16.0)
float(int)	float8	convert integer to floating point	float(2)
float4(int)	float4	convert integer to floating point	float4(2)
integer(float)	int	convert floating point to integer	integer(2.0)

10.2. String Functions

SQL92 defines string functions with specific syntax. Some of these are implemented using other Postgres functions.

Table 10.2. SQL92 String Functions

Function	Returns	Description	Example
position(text in text)	int4	location of specified substring	position('o' in 'Tom')
substring(text [from int] [for int])	text	extract specified substring	substring('Tom' from 2 for 2)
trim([leading|trailing| both] [text] from text)	text	trim characters from text	trim(both 'x' from 'xTomx')

Many string functions are available for text, varchar(), and char() types. Some are used internally to implement the SQL92 string functions listed above.

Table 10.3. String Functions

Function	Returns	Description	Example
char(text)	char	convert text to char type	char('text string')
char(varchar)	char	convert varchar to char type	char(varchar 'varchar string')

Function	Returns	Description	Example
initcap(text)	text	first letter of each word to upper case	initcap('thomas')
lower(text)	text	convert text to lower case	lower('TOM')
lpad(text,int,text)	text	left pad string to specified length	lpad('hi',4,'??')
ltrim(text,text)	text	left trim characters from text	ltrim('xxxxtrim','x')
position(text,text)	text	extract specified substring	position('high','ig')
rpad(text,int,text)	text	right pad string to specified length	rpad('hi',4,'x')
rtrim(text,text)	text	right trim characters from text	rtrim('trimxxxx','x')
substr(text,int[,int])	text	extract specified substring	substr('hi there',3,5)
text(char)	text	convert char to text type	text('char string')
text(varchar)	text	convert varchar to text type	text(varchar 'varchar string')
translate(text,from,to)	text	convert character in string	translate('12345', '1', 'a')
varchar(char)	varchar	convert char to varchar type	varchar('char string')
varchar(text)	varchar	convert text to varchar type	varchar('text string')
upper(text)	text	convert text to upper case	upper('tom')

Most functions explicitly defined for text will work for char() and varchar() arguments.

10.3. Date/Time Functions

The date/time functions provide a powerful set of tools for manipulating various date/time types.

Table 10.4. Date/Time Functions

Function	Returns	Description	Example
abstime(datetime)	abstime	convert to abstime	abstime('now'::datetime)
age(datetime,datetime)	timespan	span preserving months and years	age('now','1957-06-13'::datetime)
datetime(abstime)	datetime	convert to datetime	datetime('now'::abstime)
datetime(date)	datetime	convert to datetime	datetime('today'::date)
datetime(date,time)	datetime	convert to datetime	datetime('1998-02-24'::datetime, '23:07'::time);
date_part(text,datetime)	float8	specified portion of date field	date_part('dow','now'::datetime)

Function	Returns	Description	Example
date_part(text,timespan)	float8	specified portion of time field	date_part('hour','4 hrs 3 mins'::timespan)
date_trunc(text,datetime)	datetime	truncate date at specified units	date_trunc('month','now'::abstime)
isfinite(abstime)	bool	TRUE if this is a finite time	isfinite('now'::abstime)
isfinite(datetime)	bool	TRUE if this is a finite time	isfinite('now'::datetime)
isfinite(timespan)	bool	TRUE if this is a finite time	isfinite('4 hrs'::timespan)
reltime(timespan)	reltime	convert to reltime	reltime('4 hrs'::timespan)
timespan(reltime)	timespan	convert to timespan	timespan('4 hours'::reltime)

For the `date_part` and `date_trunc` functions, arguments can be `year', `month', `day', `hour', `minute', and `second', as well as the more specialized quantities `decade', `century', `millenium', `millisecond', and `microsecond'. `date_part` allows `dow' to return day of week and `epoch' to return seconds since 1970 (for `datetime`) or 'epoch' to return total elapsed seconds (for `timespan`).

10.4. Geometric Functions

The geometric types point, box, lseg, line, path, polygon, and circle have a large set of native support functions.

Table 10.5. Geometric Functions

Function	Returns	Description	Example
area(box)	float8	area of box	area('((0,0),(1,1))'::box)
area(circle)	float8	area of circle	area('((0,0),2.0)'::circle)
box(box,box)	box	boxes to intersection box	box('((0,0),(1,1))','((0.5,0.5),(2,2))')
center(box)	point	center of object	center('((0,0),(1,2))'::box)
center(circle)	point	center of object	center('((0,0),2.0)'::circle)
diameter(circle)	float8	diameter of circle	diameter('((0,0),2.0)'::circle)
height(box)	float8	vertical size of box	height('((0,0),(1,1))'::box)
isclosed(path)	bool	TRUE if this is a closed path	isclosed('((0,0),(1,1),(2,0))'::path)
isopen(path)	bool	TRUE if this is an open path	isopen('[(0,0),(1,1),(2,0)]'::path)
length(lseg)	float8	length of line segment	length('((-1,0),(1,0))'::lseg)
length(path)	float8	length of path	length('((0,0),(1,1),(2,0))'::path)

Function	Returns	Description	Example
pclose(path)	path	convert path to closed variant	popen('[(0,0),(1,1),(2,0)]'::path)
point(lseg,lseg)	point	convert to point (intersection)	point('((-1,0),(1,0))'::lseg,'((-2,-2),(2,2))'::lseg)
points(path)	int4	number of points in path	points('[(0,0),(1,1),(2,0)]'::path)
popen(path)	path	convert path to open variant	popen('((0,0),(1,1),(2,0))'::path)
radius(circle)	float8	radius of circle	radius('((0,0),2.0)'::circle)
width(box)	float8	horizontal size of box	width('((0,0),(1,1))'::box)

Table 10.6. Geometric Type Conversion Functions

Function	Returns	Description	Example
box(circle)	box	convert circle to box	box('((0,0),2.0)'::circle)
box(point,point)	box	convert points to box	box('(0,0)'::point,'(1,1)'::point)
box(polygon)	box	convert polygon to box	box('((0,0),(1,1),(2,0))'::polygon)
circle(box)	circle	convert to circle	circle('((0,0),(1,1))'::box)
circle(point,float8)	circle	convert to circle	circle('(0,0)'::point,2.0)
lseg(box)	lseg	convert diagonal to lseg	lseg('((-1,0),(1,0))'::box)
lseg(point,point)	lseg	convert to lseg	lseg('(-1,0)'::point,'(1,0)'::point)
path(polygon)	point	convert to path	path('((0,0),(1,1),(2,0))'::polygon)
point(circle)	point	convert to point (center)	point('((0,0),2.0)'::circle)
point(lseg,lseg)	point	convert to point (intersection)	point('((-1,0),(1,0))'::lseg,'((-2,-2),(2,2))'::lseg)
point(polygon)	point	center of polygon	point('((0,0),(1,1),(2,0))'::polygon)
polygon(box)	polygon	convert to polygon with 12 points	polygon('((0,0),(1,1))'::box)
polygon(circle)	polygon	convert to polygon with 12 points	polygon('((0,0),2.0)'::circle)
polygon(npts,circle)	polygon	convert to polygon with npts points	polygon(12,'((0,0),2.0)'::circle)
polygon(path)	polygon	convert to polygon	polygon('((0,0),(1,1),(2,0))'::path)

Table 10.7. Geometric Upgrade Functions

Function	Returns	Description	Example
isoldpath(path)	path	test path for pre-v6.1 form	isoldpath('(1,3,0,0,1,1,2,0)'::path)
revertpoly(polygon)	polygon	convert pre-v6.1 polygon	revertpoly('((0,0),(1,1),(2,0))'::polygon)
upgradepath(path)	path	convert pre-v6.1 path	upgradepath('(1,3,0,0,1,1,2,0)'::path)
upgradepoly(polygon)	polygon	convert pre-v6.1 polygon	upgradepoly('(0,1,2,0,1,0)'::polygon)

10.5. IP V4 Functions

Table 10.8. PostgresIP V4 Functions

Function	Returns	Description	Example
broadcast(cidr)	text	construct broadcast address as text	broadcast('192.168.1.5/24') ==> '192.168.1.255'
broadcast(inet)	text	construct broadcast address as text	broadcast('192.168.1.5/24') ==> '192.168.1.255'
host(inet)	text	extract host address as text	host('192.168.1.5/24') ==> '192.168.1.5'
masklen(cidr)	int4	calculate netmask length	masklen('192.168.1.5/24') ==> 24
masklen(inet)	int4	calculate netmask length	masklen('192.168.1.5/24') ==> 24
netmask(inet)	text	construct netmask as text	netmask('192.168.1.5/24') ==> '255.255.255.0'

Chapter 11. Type Conversion

SQL queries can, intentionally or not, require mixing of different data types in the same expression. Postgres 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 Postgres can affect the apparent results of a query, and these results can be tailored by a user or programmer using *explicit* type coersion.

This chapter introduces the Postgres type conversion mechanisms and conventions. Refer to the relevant sections in the User's Guide and Programmer's Guide for more information on specific data types and allowed functions and operators.

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

11.1. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. Postgres has an extensible type system which is much more general and flexible than other RDBMS implementations. Hence, most type conversion behavior in Postgres 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 Postgres scanner/parser decodes lexical elements into only five fundamental categories: integers, floats, strings, names, and keywords. Most extended types are first tokenized into strings. The SQL language definition allows specifying type names with strings, and this mechanism is used by Postgres 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 strings, of type `text` and `point`. If a type is not specified, 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 Postgres parser:

Operators

Postgres allows expressions with left- and right-unary (one argument) operators, as well as binary (two argument) operators.

Function calls

Much of the Postgres 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.

Query targets

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

UNION queries

Since all select results from a UNION SELECT statement must appear in a single set of columns, the types of each SELECT clause must be matched up and converted to a uniform set.

Many of the general type conversion rules use simple conventions built on the Postgres function and operator system tables. There are some heuristics included in the conversion rules to better support conventions for the SQL92 standard native types such as `smallint`, `integer`, and `float`.

The Postgres parser uses the convention that all type conversion functions take a single argument of the source type and are named with the same name as the target type. Any function meeting this criteria is considered to be a valid conversion function, and may be used by the parser as such. This simple assumption gives the parser the power to explore type conversion possibilities without hardcoding, allowing extended user-defined types to use these same features transparently.

An additional heuristic is provided in the parser to allow better guesses at proper behavior for SQL standard types. There are five categories of types defined: boolean, string, numeric, geometric, and user-defined. Each category, with the exception of user-defined, has a "preferred type" which is used to resolve ambiguities in candidates. Each "user-defined" type is its own "preferred type", so ambiguous expressions (those with multiple candidate parsing solutions) with only one user-defined type can resolve to a single best choice, while those with multiple user-defined types will remain ambiguous and throw an error.

Ambiguous expressions which have candidate solutions within only one type category are likely to resolve, while ambiguous expressions with candidates spanning multiple categories are likely to throw an error and ask for clarification from the user.

11.1.1. Guidelines

All type conversion rules are designed with several principles in mind:

- Implicit conversions should never have suprising or unpredictable outcomes.
- User-defined types, of which the parser has no apriori knowledge, should be "higher" in the type heirarchy. 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, Postgres 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.

11.2. Operators

11.2.1. Conversion Procedure

Operator Evaluation

1. Check for an exact match in the pg_operator system catalog.

    a. (Optional) If one argument of a binary operator is `unknown`, then assume it is the same type as the other argument.

    b. Reverse the arguments, and look for an exact match with an operator which points to itself as being commutative. If found, then reverse the arguments in the parse tree and use this operator.

2. Look for the best match.

    a. (Optional) Make a list of all operators of the same name.

    b. If only one operator is in the list, use it if the input type can be coerced, and throw an error if the type cannot be coerced.

    c. Keep all operators with the most explicit matches for types. Keep all if there are no explicit matches and move to the next step. If only one candidate remains, use it if the type can be coerced.

    d. If any input arguments are "unknown", categorize the input candidates as boolean, numeric, string, geometric, or user-defined. If there is a mix of categories, or more than one user-defined type, throw an error because the correct choice cannot be deduced without more clues. If only one category is present, then assign the "preferred type" to the input column which had been previously "unknown".

    e. Choose the candidate with the most exact type matches, and which matches the "preferred type" for each column category from the previous step. If there is still more than one candidate, or if there are none, then throw an error.

11.2.2. Examples

11.2.2.1. Exponentiation Operator

There is only one exponentiation operator defined in the catalog, and it takes `float8` arguments. The scanner assigns an initial type of `int4` 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 float8(2) ^ float8(3) 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.

11.2.2.2. String Concatenation

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.

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 all arguments for all the candidates are string types. It chooses the "preferred type" for strings, `text`, for this query.

Note

If a user defines a new type and defines an operator “||” to work with it, then this query would no longer succeed as written. The parser would now have candidate types from two categories, and could not decide which to use.

11.2.2.3. Factorial

This example illustrates an interesting result. Traditionally, the factorial operator is defined for integers only. The Postgres operator catalog has only one entry for factorial, taking an integer operand. If given a non-integer numeric argument, Postgres will try to convert that argument to an integer for evaluation of the factorial.

```
tgl=> select (4.3 !);
?column?
--------
      24
(1 row)
```

Note

Of course, this leads to a mathematically suspect result, since in principle the factorial of a non-integer is not defined. However, the role of a database is not to teach mathematics, but to be a tool for data manipulation. If a user chooses to take the factorial of a floating point number, Postgres will try to oblige.

11.3. Functions

Function Evaluation

1. Check for an exact match in the pg_proc system catalog.
2. Look for the best match.
   a. Make a list of all functions of the same name with the same number of arguments.
   b. If only one function is in the list, use it if the input types can be coerced, and throw an error if the types cannot be coerced.
   c. Keep all functions with the most explicit matches for types. Keep all if there are no explicit matches and move to the next step. If only one candidate remains, use it if the type can be coerced.
   d. If any input arguments are "unknown", categorize the input candidate arguments as boolean, numeric, string, geometric, or user-defined. If there is a mix of categories, or more than one user-defined type, throw an error because the correct choice cannot be deduced without more clues. If only one category is present, then assign the "preferred type" to the input column which had been previously "unknown".
   e. Choose the candidate with the most exact type matches, and which matches the "preferred type" for each column category from the previous step. If there is still more than one candidate, or if there are none, then throw an error.

11.3.1. Examples

11.3.1.1. Factorial Function

There is only one factorial 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)
```

11.3.1.2. Substring Function

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

There are some heuristics in the parser to optimize the relationship between the `char`, `varchar`, and `text` types. For this case, `substr` is called directly with the `varchar` string rather than inserting an explicit conversion call.

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)
```

actually executes as

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

11.4. Query Targets

Target Evaluation

1. Check for an exact match with the target.
2. Try to coerce the expression directly to the target type if necessary.
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 of the same name as the type taking two arguments, the first the type name and the second an integer length.

11.4.1. Examples

11.4.1.1. `varchar` Storage

For a target column declared as `varchar(4)` the following query ensures that the target is sized correctly:

```
tgl=> CREATE TABLE vv (v varchar(4));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> select * from vv;
v
----
abcd
(1 row)
```

11.5. UNION Queries

The UNION construct is somewhat different in that it must match up possibly dissimilar types to become a single result set.

UNION Evaluation

1. Check for identical types for all results.
2. Coerce each result from the UNION clauses to match the type of the first SELECT clause or the target column.

11.5.1. Examples

11.5.1.1. Underspecified Types

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

11.5.1.2. Simple UNION

```
tgl=> SELECT 1.2 AS Float8 UNION SELECT 1;
```

```
Float8
------
     1
   1.2
(2 rows)
```

11.5.1.3. Transposed UNION

The types of the union are forced to match the types of the first/top clause in the union:

```
tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
All integers
------------
           1
           2
           3
(3 rows)
```

An alternate parser strategy could be to choose the "best" type of the bunch, but this is more difficult because of the nice recursion technique used in the parser. However, the "best" type is used when selecting *into* a table:

```
tgl=> CREATE TABLE ff (f float);
CREATE
tgl=> INSERT INTO ff
tgl-> SELECT 1
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
INSERT 0 3
tgl=> SELECT f AS "Floating point" from ff;
  Floating point
----------------
               1
2.20000004768372
             3.3
(3 rows)
```

Chapter 12. Indices and Keys

Herouth Maoz

Author

Written by Herouth Maoz[1]

Editor's Note

This originally appeared on the mailing list in response to the question: "What is the difference between PRIMARY KEY and UNIQUE constraints?".

```
Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE

        What's the difference between:

              PRIMARY KEY(fields,...) and
              UNIQUE (fields,...)

       - Is this an alias?
       - If PRIMARY KEY is already unique, then why
         is there another kind of key named UNIQUE?
```

A primary key is the field(s) used to identify a specific row. For example, Social Security numbers identifying a person.

A simply UNIQUE combination of fields has nothing to do with identifying the row. It's simply an integrity constraint. For example, I have collections of links. Each collection is identified by a unique number, which is the primary key. This key is used in relations.

However, my application requires that each collection will also have a unique name. Why? So that a human being who wants to modify a collection will be able to identify it. It's much harder to know, if you have two collections named "Life Science", the the one tagged 24433 is the one you need, and the one tagged 29882 is not.

So, the user selects the collection by its name. We therefore make sure, withing the database, that names are unique. However, no other table in the database relates to the collections table by the collection Name. That would be very inefficient.

Moreover, despite being unique, the collection name does not actually define the collection! For example, if somebody decided to change the name of the collection from "Life Science" to "Biology", it will still be the same collection, only with a different name. As long as the name is unique, that's OK.

So:

- Primary key:
  - Is used for identifying the row and relating to it.
  - Is impossible (or hard) to update.
  - Should not allow NULLs.

- Unique field(s):

[1] herouth@oumail.openu.ac.il

- Are used as an alternative access to the row.
- Are updateable, so long as they are kept unique.
- NULLs are acceptable.

As for why no non-unique keys are defined explicitly in standard SQL syntax? Well, you must understand that indices are implementation-dependent. SQL does not define the implementation, merely the relations between data in the database. Postgres does allow non-unique indices, but indices used to enforce SQL keys are always unique.

Thus, you may query a table by any combination of its columns, despite the fact that you don't have an index on these columns. The indexes are merely an implementational aid which each RDBMS offers you, in order to cause commonly used queries to be done more efficiently. Some RDBMS may give you additional measures, such as keeping a key stored in main memory. They will have a special command, for example

```
CREATE MEMSTORE ON <table> COLUMNS <cols>
```

(this is not an existing command, just an example).

In fact, when you create a primary key or a unique combination of fields, nowhere in the SQL specification does it say that an index is created, nor that the retrieval of data by the key is going to be more efficient than a sequential scan!

So, if you want to use a combination of fields which is not unique as a secondary key, you really don't have to specify anything - just start retrieving by that combination! However, if you want to make the retrieval efficient, you'll have to resort to the means your RDBMS provider gives you - be it an index, my imaginary MEMSTORE command, or an intelligent RDBMS which creates indices without your knowledge based on the fact that you have sent it many queries based on a specific combination of keys... (It learns from experience).

Chapter 13. Arrays

Note

This must become a chapter on array behavior. Volunteers? - thomas 1998-01-12

Postgres allows attributes of an instance to be defined as fixed-length or variable-length multi-dimensional arrays. Arrays of any base type or user-defined type can be created. To illustrate their use, we first create a class with arrays of base types.

```
CREATE TABLE SAL_EMP (
    name            text,
    pay_by_quarter  int4[],
    schedule        text[][]
);
```

The above query will create a class named SAL_EMP with a *text* string (name), a one-dimensional array of *int4* (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 *INSERTS*s; note that when appending to an array, we enclose the values within braces and separate them by commas. If you know *C*, this is not unlike the syntax for initializing structures.

```
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"}}');
```

By default, Postgres uses the "one-based" numbering convention for arrays -- that is, an array of n elements starts with array[1] and ends with array[n]. 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 SAL_EMP.pay_by_quarter[1] <>
    SAL_EMP.pay_by_quarter[2];

+------+
|name  |
+------+
|Carol |
+------+
```

This query retrieves the third quarter pay of all employees:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;


+---------------+
```

```
|pay_by_quarter |
+---------------+
|10000          |
+---------------+
|25000          |
+---------------+
```

We can also access arbitrary slices of an array, or subarrays. This query retrieves the first item on Bill's schedule for the first two days of the week.

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

```
+-------------------+
|schedule           |
+-------------------+
|{{"meeting"},{""}} |
+-------------------+
```

Chapter 14. Inheritance

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

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

CREATE TABLE capitals (
    state           char2
) INHERITS (cities);
```

In this case, an instance of capitals *inherits* all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is `text`, a native Postgres type for variable length ASCII strings. The type of the attribute population is `float`, a native Postgres type for double precision floating point numbers. State capitals have an extra attribute, state, that shows their state. In Postgres, a class can inherit from zero or more other classes, and a query can reference either all instances of a class or all instances of a class plus all of its descendants.

Note

The inheritance hierarchy is a actually a directed acyclic graph.

For example, the following query finds all the cities that are situated at an attitude of 500ft or higher:

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

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

On the other hand, to find the names of all cities, including state capitals, that are located at an altitude over 500ft, the query is:

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

which returns:

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

```
+----------+----------+
|Madison   | 845      |
+----------+----------+
```

Here the "*" after cities indicates that the query should be run over cities and all classes below cities in the inheritance hierarchy. Many of the commands that we have already discussed -- `select`, `update` and `delete` -- support this "*" notation, as do others, like `alter`.

Chapter 15. The Query Language

Note

This chapter must go into depth on each area of the query language. Currently a copy of the tutorial. - thomas 1998-01-12

The Postgres query language is a variant of SQL3. It has many extensions such as an extensible type system, inheritance, functions and production rules. Those are features carried over from the original Postgres query language, PostQuel. This section provides an overview of how to use Postgres SQL to perform simple operations. This manual is only intended to give you an idea of our flavor of SQL and is in no way a complete tutorial on SQL. Numerous books have been written on SQL. For instance, consult Understanding the New SQL or A Guide to the SQL Standard. You should also be aware that some features of Postgres are not part of the ANSI standard.

15.1. Concepts

The fundamental notion in Postgres is that of a class, which is a named collection of object instances. Each instance has the same collection of named attributes, and each attribute is of a specific type. Furthermore, each instance has a permanent *object identifier* (OID) that is unique throughout the installation. Because SQL syntax refers to tables, we will use the terms *table* and *class* interchangeably. Likewise, an SQL *row* is an *instance* and SQL *columns* are *attributes*. As previously discussed, classes are grouped into databases, and a collection of databases managed by a single `postmaster` process constitutes an installation or site.

15.2. Creating a New Class

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

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

Note that keywords are case-insensitive and identifiers are usually case-insensitive. Postgres allows SQL92 *delimited identifiers* (identifiers surrounded by double-quotes) to include mixed-case and spaces, tabs, etc.

Postgres SQL supports the usual SQL types `int`, `float`, `real`, `smallint`, `char(N)`, `varchar(N)`, `date`, `time`, and `timestamp`, as well as other types of general utility and a rich set of geometric types. As we will see later, Postgres can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical keywords, except where required to support special cases in the SQL92 standard. So far, the Postgres create command looks exactly like the command used to create a table in a traditional relational system. However, we will presently see that classes have properties that are extensions of the relational model.

15.3. Populating a Class with Instances

The `insert` statement is used to populate a class with instances:

```
INSERT INTO weather
    VALUES ('San Francisco', 46, 50, 0.25, '11/27/1994')
```

You can also use the `copy` command to perform load large amounts of data from flat (ASCII) files. This is usually faster because the data is read (or written) as a single atomic transaction directly to or from the target table. An example would be:

```
COPY INTO weather FROM '/home/user/weather.txt'
    USING DELIMITERS '|';
```

where the path name for the source file must be available to the backend server machine, not just the client.

15.4. Querying a Class

The weather class can be queried with normal relational selection and projection queries. A SQL `select` statement is used to do this. The statement is divided into a target list (the part that lists the attributes to be returned) and a qualification (the part that specifies any restrictions). For example, to retrieve all the rows of weather, type:

```
SELECT * FROM WEATHER;
```

and the output should be:

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
|San Francisco | 43      | 57      | 0    | 11-29-1994 |
+--------------+---------+---------+------+------------+
|Hayward       | 37      | 54      |      | 11-29-1994 |
+--------------+---------+---------+------+------------+
```

You may specify any arbitrary expressions in the target list. For example, you can do:

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

Arbitrary Boolean operators (`and`, `or` and `not`) are allowed in the qualification of any query. For example,

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

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
```

As a final note, you can specify that the results of a select can be returned in a *sorted order* or with *duplicate instances* removed.

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

15.5. Redirecting SELECT Queries

Any select query can be redirected to a new class

```
SELECT * INTO TABLE temp FROM weather;
```

This forms an implicit `create` command, creating a new class temp with the attribute names and types specified in the target list of the `select into` command. We can then, of course, perform any operations on the resulting class that we can perform on other classes.

15.6. Joins Between Classes

Thus far, our queries have only accessed one class at a time. Queries can access multiple classes at once, or access the same class in such a way that multiple instances of the class are being processed at the same time. A query that accesses multiple instances of the same or different classes at one time is called a join query. As an example, say we wish to find all the records that are in the temperature range of other records. In effect, we need to compare the temp_lo and temp_hi attributes of each EMP instance to the temp_lo and temp_hi attributes of all other EMP instances.

Note

This is only a conceptual model. The actual join may be performed in a more efficient manner, but this is invisible to the user.

We can do this with the following query:

```
SELECT W1.city, W1.temp_lo, W1.temp_hi,
    W2.city, W2.temp_lo, W2.temp_hi
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

+--------------+---------+---------+---------------+---------
+---------+
|city          | temp_lo | temp_hi | city          | temp_lo | temp_hi
 |
+--------------+---------+---------+---------------+---------
+---------+
|San Francisco | 43      | 57      | San Francisco | 46      | 50
  |
+--------------+---------+---------+---------------+---------
+---------+
|San Francisco | 37      | 54      | San Francisco | 46      | 50
  |
+--------------+---------+---------+---------------+---------
+---------+
```

Note

The semantics of such a join are that the qualification is a truth expression defined for the Cartesian product of the classes indicated in the query. For those instances in the Cartesian product for which the qualification is true, Postgres computes and returns the values specified in the target list. Postgres SQL does not assign any meaning to duplicate values in such expressions. This means that Postgres sometimes recomputes the same target list several times; this frequently happens

when Boolean expressions are connected with an "or". To remove such duplicates, you must use the `select distinct` statement.

In this case, both W1 and W2 are surrogates for an instance of the class weather, and both range over all instances of the class. (In the terminology of most database systems, W1 and W2 are known as *range variables*.) A query can contain an arbitrary number of class names and surrogates.

15.7. Updates

You can update existing instances using the update command. Suppose you discover the temperature readings are all off by 2 degrees as of Nov 28, you may update the data as follow:

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

15.8. Deletions

Deletions are performed using the `delete` command:

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

All weather recording belongs to Hayward is removed. One should be wary of queries of the form

```
DELETE FROM classname;
```

Without a qualification, `delete` will simply remove all instances of the given class, leaving it empty. The system will not request confirmation before doing this.

15.9. Using Aggregate Functions

Like most other query languages, PostgreSQL supports aggregate functions. The current implementation of Postgres aggregate functions have some limitations. Specifically, while there are aggregates to compute such functions as the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of instances, aggregates can only appear in the target list of a query and not directly in the qualification (the *where* clause). As an example,

```
SELECT max(temp_lo) FROM weather;
```

is allowed, while

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

is not. However, as is often the case the query can be restated to accomplish the intended result; here by using a *subselect*:

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

Aggregates may also have *group by* clauses:

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

Chapter 16. Disk Storage

This section needs to be written. Some information is in the FAQ. Volunteers? - thomas 1998-01-11

Chapter 17. psql

This section needs to be converted from the original psql man page. Volunteers?

Tip

To change the paging behavior of your results, set/unset your PAGER environment variable.

17.1. Paging To Screen

Author

From Brett McCormick on the mailing list 1998-04-04.

To affect the paging behavior of your `psql` output, set or unset your PAGER environment variable. I always have to set mine before it will pause. And of course you have to do this before starting the program.

In csh/tcsh or other C shells:

```
unsetenv PAGER
```

while in sh/bash or other Bourne shells:

```
unset PAGER
```

Chapter 18. pgaccess

This section needs to be written. Volunteers?

Chapter 19. SQL Commands

This is reference information for the SQL commands supported by Postgres.

Name

ABORT — Aborts the current transaction

Synopsis

<refsynopsisdivinfo>1998-09-27</refsynopsisdivinfo>

```
ABORT
```

Inputs

None.

Outputs

ABORT

Message returned if successful.

NOTICE: UserAbortTransactionBlock and not in in-progress state ABORT

If there is not any transaction currently in progress.

Description

`ABORT` rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the SQL92 command `ROLLBACK`, and is present only for historical reasons.

Notes

Use the `COMMIT` statement to successfully terminate a transaction.

Usage

```
--To abort all changes
--
ABORT WORK;
```

Compatibility

SQL92

This command is a Postgres extension present for historical reasons. `ROLLBACK` is the SQL92 equivalent command.

Name

ALTER TABLE — Modifies table properties

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
ALTER TABLE table
    [ * ] ADD [ COLUMN ] column type
ALTER TABLE table
    [ * ] RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO newtable
```

Inputs

table

The name of an existing table to alter.

column

Name of a new or existing column.

type

Type of the new column.

newcolumn

New name for an existing column.

newtable

New name for an existing column.

Outputs

ALTER

Message returned from column or table renaming.

NEW

Message returned from column addition.

ERROR

Message returned if table or column is not available.

Description

`ALTER TABLE` changes the definition of an existing table. The new columns and their types are specified in the same style and with the the same restrictions as in `CREATE TABLE`. The RENAME clause causes the name of a table or column to change without changing any of the data contained in the affected table. Thus, the table or column will remain of the same type and size after this command is executed.

You must own the table in order to change its schema.

Notes

The keyword COLUMN is noise and can be omitted.

"[*]" following a name of a table indicates that statement should be run over that table and all tables below it in the inheritance hierarchy. The *PostgreSQL User's Guide* has further information on inheritance.

Refer to CREATE TABLE for a further description of valid arguments.

Usage

To add a column of type VARCHAR to a table:

```
ALTER TABLE distributors ADD COLUMN address VARCHAR(30);
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

Compatibility

SQL92

`ALTER TABLE/RENAME` is a Postgres language extension.

SQL92 specifies some additional capabilities for `ALTER TABLE` statement which are not yet directly supported by Postgres:

```
ALTER TABLE table ALTER [ COLUMN ] column
    SET DEFAULT default

ALTER TABLE table ALTER [ COLUMN ] column
    ADD [ CONSTRAINT constraint ] table-constraint
```

Puts the default value or constraint specified into the definition of column in the table. See `CREATE TABLE` for the syntax of the default and table-constraint clauses. If a default clause already exists, it will be replaced by the new definition. If any constraints on this column already exist, they will be retained using a boolean AND with the new constraint.

Currently, to set new default constraints on an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT * FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors (
    did      DECIMAL(3) DEFAULT 1,
    name     VARCHAR(40) NOT NULL,
    city     VARCHAR(30)
);
```

```
INSERT INTO distributors SELECT * FROM temp;
DROP TABLE temp;
```

```
ALTER TABLE table
    DROP DEFAULT default
ALTER TABLE table
    DROP CONSTRAINT constraint { RESTRICT | CASCADE }
```

Removes the default value specified by default or the rule specified by constraint from the definition of a table. If RESTRICT is specified only a constraint with no dependent constraints can be destroyed. If CASCADE is specified, Any constraints that are dependent on this constraint are also dropped.

Currently, to remove a default value or constraints on an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT * FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors AS SELECT * FROM temp;
DROP TABLE temp;
```

```
ALTER TABLE table
    DROP [ COLUMN ] column { RESTRICT | CASCADE }
```

Removes a column from a table. If RESTRICT is specified only a column with no dependent objects can be destroyed. If CASCADE is specified, all objects that are dependent on this column are also dropped.

Currently, to remove an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT did, city FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT 1,
    name     VARCHAR(40) NOT NULL,
);
INSERT INTO distributors SELECT * FROM temp;
DROP TABLE temp;
```

Name

ALTER USER — Modifies user account information

Synopsis

<refsynopsisdivinfo>1998-09-08</refsynopsisdivinfo>

```
ALTER USER username
    [ WITH PASSWORD password ]
    [ CREATEDB | NOCREATEDB ]
    [ CREATEUSER | NOCREATEUSER ]
    [ IN GROUP groupname [, ...] ]
    [ VALID UNTIL 'abstime' ]
```

Inputs

Refer to `CREATE USER` for a detailed description of each clause.

username

The Postgres account name of the user whose details are to be altered.

password

The new password to be used for this account.

groupname

The name of an access group into which this account is to be put.

abstime

The date (and, optionally, the time) at which this user's access is to be terminated.

Outputs

ALTER USER

Message returned if the alteration was successful.

ERROR: alterUser: user "username" does not exist

Error message returned if the user specified doesn't exist.

Description

`ALTER USER` is used to change the attributes of a user's Postgres account. Please note that it is not possible to alter a user's "usesysid" via the alter user statement. Also, it is only possible for the Postgres user or any user with read and modify permissions on "pg_shadow" to alter user passwords.

If any of the clauses of the alter user statement are omitted, the corresponding value in the "pg_shadow" table is left unchanged.

Notes

`ALTER USER` statement is a Postgres language extension.

Refer to `CREATE/DROP USER` to create or remove a user account.

In the current release (v6.4), the IN GROUP clause is parsed but has no affect. When it is fully implemented, it is intended to modify the pg_group relation.

Usage

Change a user password

```
ALTER USER davide WITH PASSWORD hu8jmn3;
```

Change a user's valid until date

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

Change a user's valid until date, specifying that his authorisation should expire at midday on 4th May 1998 using the time zone which is one hour ahead of UTC

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 1998 +1';
```

Give a user the ability to create other users and new databases.

```
ALTER USER miriam CREATEUSER CREATEDB;
```

Place a user in two groups

```
ALTER USER miriam IN GROUP sales, payroll;
```

Compatibility

SQL92

There is no `ALTER USER` statement in SQL92. The standard leaves the definition of users to the implementation.

Name

BEGIN WORK — Begins a transaction

Synopsis

<refsynopsisdivinfo>1998-09-08</refsynopsisdivinfo>

```
BEGIN [ WORK | TRANSACTION ]
```

Inputs

None

Outputs

BEGIN

This signifies that a new transaction has been started.

NOTICE: BeginTransactionBlock and not in default state

This indicates that a transaction was already in progress. The current transaction is not affected.

Description

`BEGIN` initiates a user transaction which Postgres will guarantee is serializable with respect to all concurrently executing transactions. Postgres uses two-phase locking to perform this task. If the transaction is committed, Postgres will ensure either that all updates are done or else that none of them are done. Transactions have the standard ACID (atomic, consistent, isolatable, and durable) property.

Notes

The keyword TRANSACTION is just a cosmetic alternative to WORK. Neither keyword need be specified.

Refer to the `LOCK` statement for further information about locking tables inside a transaction.

Use `COMMIT` or `ROLLBACK` to terminate a transaction.

Usage

To begin a user transaction:

```
BEGIN WORK;
```

Compatibility

`BEGIN` is a Postgres language extension.

SQL92

There is no explicit BEGIN WORK command in SQL92; transaction initiation is always implicit and it terminates either with a COMMIT or with a ROLLBACK statement.

Name

CLOSE — Close a cursor

Synopsis

<refsynopsisdivinfo>1998-09-08</refsynopsisdivinfo>

```
CLOSE cursor
```

Inputs

cursor

The name of an open cursor to close.

Outputs

CLOSE

Message returned if the cursor is successfully closed.

NOTICE PerformPortalClose: portal "*cursor*" not found

This warning is given if *cursor* is not declared or has already been closed.

Description

`CLOSE` frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

An implicit close is executed for every open cursor when a transaction is terminated by `COMMIT` or `ROLLBACK`.

Notes

Postgres does not have an explicit `OPEN` cursor statement; a cursor is considered open when it is declared. Use the `DECLARE` statement to declare a cursor.

Usage

Close the cursor liahona:

```
CLOSE liahona;
```

Compatibility

SQL92

`CLOSE` is fully compatible with SQL92.

Name

CLUSTER — Gives storage clustering advice to the backend

Synopsis

<refsynopsisdivinfo>1998-09-08</refsynopsisdivinfo>

```
CLUSTER indexname ON table
```

Inputs

indexname

The name of an index.

table

The name of a table.

Outputs

CLUSTER

The clustering was done successfully.

ERROR: relation <*tablerelation_number*> inherits "invoice"

ERROR: Relation x does not exist!

Description

`CLUSTER` instructs Postgres to cluster the class specified by *classname* approximately based on the index specified by *indexname*. The index must already have been defined on *classname*.

When a class is clustered, it is physically reordered based on the index information. The clustering is static. In other words, as the class is updated, the changes are not clustered. No attempt is made to keep new instances or updated tuples clustered. If one wishes, one can recluster manually by issuing the command again.

Notes

The table is actually copied to a temporary table in index order, then renamed back to the original name. For this reason, all grant permissions and other indexes are lost when clustering is performed.

In cases where you are accessing single rows randomly within a table, the actual order of the data in the heap table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using `CLUSTER`.

Another place `CLUSTER` is helpful is in cases where you use an index to pull out several rows from a table. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple

rows that match, CLUSTER will help because once the index identifies the heap page for the first row that matches, all other rows that match are probably already on the same heap page, saving disk accesses and speeding up the query.

There are two ways to cluster data. The first is with the CLUSTER command, which reorders the original table with the ordering of the index you specify. This can be slow on large tables because the rows are fetched from the heap in index order, and if the heap table is unordered, the entries are on random pages, so there is one disk page retrieved for every row moved. Postgres has a cache, but the majority of a big table will not fit in the cache.

Another way to cluster data is to use

```
SELECT ... INTO TABLE temp FROM ... ORDER BY ...
```

This uses the Postgres sorting code in ORDER BY to match the index, and is much faster for unordered data. You then drop the old table, use ALTER TABLE/RENAME to rename *temp* to the old name, and recreate any indexes. The only problem is that OIDs will not be preserved. From then on, CLUSTER should be fast because most of the heap data has already been ordered, and the existing index is used.

Usage

Cluster the employees relation on the basis of its salary attribute

```
CLUSTER emp_ind ON emp
```

Compatibility

SQL92

There is no CLUSTER statement in SQL92.

Name

COMMIT — Commits the current transaction

Synopsis

<refsynopsisdivinfo>1998-09-08</refsynopsisdivinfo>

```
COMMIT [ WORK | TRANSACTION ]
```

Inputs

None

Outputs

END

Message returned if the transaction is successfully committed.

NOTICE EndTransactionBlock and not inprogress/abort state

If there is no transaction in progress.

Description

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Notes

The keywords WORK and TRANSACTION are noise and can be omitted.

Use the `ROLLBACK` statement to abort a transaction.

Usage

To make all changes permanent:

```
COMMIT WORK;
```

Compatibility

SQL92

Full compatibility.

Name

COPY — Copies data between files and tables

Synopsis

<refsynopsisdivinfo>1998-09-08</refsynopsisdivinfo>

```
COPY [ BINARY ] table [ WITH OIDS ]
    FROM { 'filename' | stdin }
    [ USING DELIMITERS 'delimiter' ]
COPY [ BINARY ] table [ WITH OIDS ]
    TO { 'filename' | stdout }
    [ USING DELIMITERS 'delimiter' ]
```

Inputs

BINARY

Changes the behavior of field formatting, forcing all data to be stored or read as binary objects rather than as text.

table

The name of an existing table.

WITH OIDS

Copies the internal unique object id (OID) for each row.

filename

The absolute Unix pathname of the input or output file.

stdin

Specifies that input comes from a pipe or terminal.

stdout

Specifies that output goes to a pipe or terminal.

delimiter

A character that delimits the input or output fields.

Outputs

COPY

The copy completed successfully.

ERROR: *error message*

The copy failed for the reason stated in the error message.

Description

COPY moves data between Postgres tables and standard Unix files.

COPY instructs the Postgres backend to directly read from or write to a file. The file must be directly visible to the backend and the name must be specified from the viewpoint of the backend. If stdin or stdout are specified, data flows through the client frontend to the backend.

Notes

The BINARY keyword will force all data to be stored/read as binary objects rather than as text. It is somewhat faster than the normal copy command, but is not generally portable, and the files generated are somewhat larger, although this factor is highly dependent on the data itself. By default, a text copy uses a tab ("\t") character as a delimiter. The delimiter may also be changed to any other single character with the keyword phrase USING DELIMITERS. Characters in data fields which happen to match the delimiter character will be quoted.

You must have select access on any table whose values are read by COPY, and either insert or update access to a table into which values are being inserted by COPY. The backend also needs appropriate Unix permissions for any file read or written by COPY.

The keyword phrase USING DELIMITERS specifies a single character to be used for all delimiters between columns. If multiple characters are specified in the delimiter string, only the first character is used.

Tip

Do not confuse COPY with the psql instruction \copy.

File Formats

Text Format

When COPY TO is used without the BINARY option, the file generated will have each row (instance) on a single line, with each column (attribute) separated by the delimiter character. Embedded delimiter characters will be preceded by a backslash character ("\"). The attribute values themselves are strings generated by the output function associated with each attribute type. The output function for a type should not try to generate the backslash character; this will be handled by COPY itself.

The actual format for each instance is

<attr1><*separator*><attr2><*separator*>...<*separator*><attr*n*><newline>

The oid is placed on the beginning of the line if WITH OIDS is specified.

If COPY is sending its output to standard output instead of a file, it will send a backslash("\") and a period (".") followed immediately by a newline, on a separate line, when it is done. Similarly, if COPY is reading from standard input, it will expect a backslash ("\") and a period (".") followed by a newline, as the first three characters on a line to denote end-of-file. However, COPY will terminate (followed by the backend itself) if a true EOF is encountered before this special end-of-file pattern is found.

The backslash character has other special meanings. NULL attributes are output as "\N". A literal backslash character is output as two consecutive backslashes ("\\"). A literal tab character is represented as a backslash and a tab. A literal newline character is represented as a backslash and a newline. When loading text data

not generated by Postgres, you will need to convert backslash characters ("\") to double-backslashes ("\\") to ensure that they are loaded properly.

Binary Format

In the case of `COPY BINARY`, the first four bytes in the file will be the number of instances in the file. If this number is zero, the `COPY BINARY` command will read until end of file is encountered. Otherwise, it will stop reading when this number of instances has been read. Remaining data in the file will be ignored.

The format for each instance in the file is as follows. Note that this format must be followed *exactly*. Unsigned four-byte integer quantities are called uint32 in the table below.

Table 19.1. Contents of a binary copy file

At the start of the file	
uint32	number of tuples
For each tuple	
uint32	total length of tuple data
uint32	oid (if specified)
uint32	number of null attributes
[uint32,...,uint32]	attribute numbers of attributes, counting from 0
-	<tuple data>

Alignment of Binary Data

On Sun-3s, 2-byte attributes are aligned on two-byte boundaries, and all larger attributes are aligned on four-byte boundaries. Character attributes are aligned on single-byte boundaries. On most other machines, all attributes larger than 1 byte are aligned on four-byte boundaries. Note that variable length attributes are preceded by the attribute's length; arrays are simply contiguous streams of the array element type.

Usage

The following example copies a table to standard output, using a vertical bar ("|") as the field delimiter:

```
COPY country TO stdout USING DELIMITERS '|';
```

To copy data from a Unix file into a table "country":

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

Here is a sample of data suitable for copying into a table from `stdin` (so it has the termination sequence on the last line):

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
...
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

The same data, output in binary format on a Linux/i586 machine. The data is shown after filtering through the Unix utility `od -c`. The table has three fields; the first is `char(2)` and the second is `text`. All the rows have a null value in the third field. Notice how the `char(2)` field is padded with nulls to four bytes and the text field is preceded by its length:

```
355  \0  \0  \0 027  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
006  \0  \0  \0   A   F  \0  \0 017  \0  \0  \0   A   F   G   H
  A   N   I   S   T   A   N 023  \0  \0  \0 001  \0  \0  \0 002
 \0  \0  \0 006  \0  \0  \0   A   L  \0  \0  \v  \0  \0  \0   A
  L   B   A   N   I   A 023  \0  \0  \0 001  \0  \0  \0 002  \0
 \0  \0 006  \0  \0  \0   D   Z  \0  \0  \v  \0  \0  \0   A   L
  G   E   R   I   A
...              \n  \0  \0  \0   Z   A   M   B   I   A 024  \0
 \0  \0 001  \0  \0  \0 002  \0  \0  \0 006  \0  \0  \0   Z   W
 \0  \0  \f  \0  \0  \0   Z   I   M   B   A   B   W   E
```

Bugs

`COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY FROM`, but the target relation will, of course, be partially modified in a `COPY TO`. The `VACUUM` query should be used to clean up after a failed copy.

Because the Postgres backend's current working directory is not usually the same as the user's working directory, the result of copying to a file "`foo`" (without additional path information) may yield unexpected results for the naive user. In this case, `foo` will wind up in `$PGDATA/foo`. In general, the full pathname as it would appear to the backend server machine should be used when specifying files to be copied.

Files used as arguments to `COPY` must reside on or be accessible to the database server machine by being either on local disks or on a networked file system.

When a TCP/IP connection from one machine to another is used, and a target file is specified, the target file will be written on the machine where the backend is running rather than the user's machine.

Compatibility

SQL92

There is no `COPY` statement in SQL92.

Name

CREATE AGGREGATE — Defines a new aggregate function

Synopsis

<refsynopsisdivinfo>1998-09-09</refsynopsisdivinfo>

```
CREATE AGGREGATE name [ AS ]
    ( BASETYPE    = data_type
    [ , SFUNC1    = sfunc1
      , STYPE1    = sfunc1_return_type ]
    [ , SFUNC2    = sfunc2
      , STYPE2    = sfunc2_return_type ]
    [ , FINALFUNC = ffunc ]
    [ , INITCOND1 = initial_condition1 ]
    [ , INITCOND2 = initial_condition2 ]
    )
```

Inputs

name

The name of an aggregate function to create.

data_type

The fundamental data type on which this aggregate function operates.

sfunc1

The state transition function to be called for every non-NULL field from the source column. It takes a variable of type *sfunc1_return_type* as the first argument and that field as the second argument.

sfunc1_return_type

The return type of the first transition function.

sfunc2

The state transition function to be called for every non-NULL field from the source column. It takes a variable of type *sfunc2_return_type* as the only argument and returns a variable of the same type.

sfunc2_return_type

The return type of the second transition function.

ffunc

The final function called after traversing all input fields. This function must take two arguments of types *sfunc1_return_type* and *sfunc2_return_type*.

initial_condition1

The initial value for the first transition function argument.

initial_condition2

The initial value for the second transition function argument.

Outputs

CREATE

Message returned if the command completes successfully.

Description

`CREATE AGGREGATE` allows a user or programmer to extend Postgres functionality by defining new aggregate functions. Some aggregate functions for base types such as `min(int4)` and `avg(float8)` are already provided in the base distribution. If one defines new types or needs an aggregate function not already provided then `CREATE AGGREGATE` can be used to provide the desired features.

An aggregate function can require up to three functions, two state transition functions, *sfunc1* and *sfunc2*:

```
sfunc1( internal-state1, next-data_item ) ---> next-internal-state1
sfunc2( internal-state2 ) ---> next-internal-state2
```

and a final calculation function, *ffunc*:

```
ffunc(internal-state1, internal-state2) ---> aggregate-value
```

Postgres creates up to two temporary variables (referred to here as *temp1* and *temp2*) to hold intermediate results used as arguments to the transition functions.

These transition functions are required to have the following properties:

- The arguments to *sfunc1* must be *temp1* of type *sfunc1_return_type* and *column_value* of type *data_type*. The return value must be of type *sfunc1_return_type* and will be used as the first argument in the next call to *sfunc1*.
- The argument and return value of *sfunc2* must be *temp2* of type *sfunc2_return_type*.
- The arguments to the final-calculation-function must be *temp1* and *temp2* and its return value must be a Postgres base type (not necessarily *data_type* which had been specified for BASETYPE).
- FINALFUNC should be specified if and only if both state-transition functions are specified.

An aggregate function may also require one or two initial conditions, one for each transition function. These are specified and stored in the database as fields of type `text`.

Notes

Use `DROP AGGREGATE` to drop aggregate functions.

It is possible to specify aggregate functions that have varying combinations of state and final functions. For example, the `count` aggregate requires SFUNC2 (an incrementing function) but not SFUNC1 or FINALFUNC, whereas the `sum` aggregate requires SFUNC1 (an addition function) but not SFUNC2 or FINALFUNC and the `avg` aggregate requires both of the above state functions as well as a FINALFUNC (a division function) to produce its answer. In any case, at least one state function must be defined, and any SFUNC2 must have a corresponding INITCOND2.

Usage

Refer to the chapter on aggregate functions in the *PostgreSQL Programmer's Guide* on aggregate functions for complete examples of usage.

Compatibility

SQL92

`CREATE AGGREGATE` is a Postgres language extension. There is no `CREATE AGGREGATE` in SQL92.

Name

CREATE DATABASE — Creates a new database

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
CREATE DATABASE name [ WITH LOCATION = 'dbpath' ]
```

Inputs

name

The name of a database to create.

dbpath

An alternate location can be specified as either an environment variable known to the backend server (e.g. 'PGDATA2') or as an absolute path name (e.g. '/usr/local/pgsql/data'). In either case, the location must be pre-configured by initlocation.

Outputs

CREATEDB

Message returned if the command completes successfully.

WARN: createdb: database "*name*" already exists.

This occurs if *database* specified already exists.

ERROR: Unable to create database directory *directory*

There was a problem with creating the required directory; this operation will need permissions for the postgres user on the specified location.

Description

CREATE DATABASE creates a new Postgres database. The creator becomes the administrator of the new database.

Notes

CREATE DATABASE is a Postgres language extension.

Use DROP DATABASE to remove a database.

Usage

To create a new database:

```
olly=> create database lusiadas;
```

To create a new database in an alternate area ~/private_db:

```
$ mkdir private_db
$ initlocation ~/private_db
Creating Postgres database system directory /home/olly/private_db/base

$ psql olly
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: template1

olly=> create database elsewhere with location = '/home/olly/
private_db';
    CREATEDB
```

Bugs

There are security and data integrity issues involved with using alternate database locations specified with absolute path names, and by default only an environment variable known to the backend may be specified for an alternate location. See the Administrator's Guide for more information.

Compatibility

SQL92

There is no CREATE DATABASE statement in SQL92.

The equivalent command in standard SQL is CREATE SCHEMA.

Name

CREATE FUNCTION — Defines a new function

Synopsis

<refsynopsisdivinfo>1998-09-09</refsynopsisdivinfo>

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS path
    LANGUAGE 'langname'
```

Inputs

name

The name of a function to create.

ftype

The data type of function arguments.

rtype

The return data type.

path

May be either an SQL-query or an absolute path to an object file.

langname

may be 'C', 'sql', 'internal' or '*plname*', where '*plname*' is the name of a created procedural language. See CREATE LANGUAGE for details.

Outputs

CREATE

This is returned if the command completes successfully.

Description

CREATE FUNCTION allows a Postgres user to register a function with a database. Subsequently, this user is treated as the owner of the function.

Notes

Refer to the chapter on functions in the *PostgreSQL Programmer's Guide* for further information.

Use DROP FUNCTION to drop user-defined functions.

Usage

To create a simple SQL function:

```
CREATE FUNCTION one() RETURNS int4
    AS 'SELECT 1 AS RESULT'
    LANGUAGE 'sql';

SELECT one() AS answer;

answer
------
1
```

To create a C function, calling a routine from a user-created shared library. This particular routine calculates a check digit and returns TRUE if the check digit in the function parameters is correct. It is intended for use in a CHECK contraint.

```
CREATE FUNCTION ean_checkdigit(bpchar, bpchar) RETURNS bool
    AS '/usr1/proj/bray/sql/funcs.so' LANGUAGE 'c';

CREATE TABLE product
(
     id              char(8)         PRIMARY KEY,
     eanprefix       char(8)         CHECK (eanprefix ~ '[0-9]{2}-
[0-9]{5}')
                                     REFERENCES brandname(ean_prefix),
     eancode         char(6)         CHECK (eancode ~ '[0-9]{6}'),
     CONSTRAINT ean CHECK (ean_checkdigit(eanprefix, eancode))
);
```

Bugs

A C function cannot return a set of values.

Compatibility

`CREATE FUNCTION` is a Postgres language extension.

SQL/PSM

Note

PSM stands for Persistent Stored Modules. It is a procedural language and it was originally hoped that PSM would be ratified as an official standard by late 1996. As of mid-1998, this has not yet happened, but it is hoped that PSM will eventually become a standard.

SQL/PSM `CREATE FUNCTION` has the following syntax:

```
CREATE FUNCTION name
    ( [ [ IN | OUT | INOUT ] parm type [, ...] ] )
     RETURNS rtype
     LANGUAGE 'langname'
     ESPECIFIC routine
     SQL-statement
```

Name

CREATE INDEX — Constructs a secondary index

Synopsis

<refsynopsisdivinfo>1998-09-09</refsynopsisdivinfo>

```
CREATE [ UNIQUE ] INDEX index_name
    ON table [ USING acc_name ]
    ( column [ ops_name] [, ...] )
CREATE [ UNIQUE ] INDEX index_name
    ON table [ USING acc_name ]
    ( func_name( column [, ... ]) ops_name )
```

Inputs

UNIQUE

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update non-duplicate data will generate an error.

index_name

The name of the index to be created.

table

The name of the table to be indexed.

acc_name

the name of the access method which is to be used for the index. The default access method is BTREE. Postgres provides three access methods for secondary indexes:

BTREE

an implementation of the Lehman-Yao high-concurrency btrees.

RTREE

implements standard rtrees using Guttman's quadratic split algorithm.

HASH

an implementation of Litwin's linear hashing.

column

The name of a column of the table.

ops_name

An associated operator class. The following select list returns all ops_names:

```
SELECT am.amname AS acc_name,
```

```
           opc.opcname AS ops_name,
           opr.oprname AS ops_comp
        FROM pg_am am, pg_amop amop,
             pg_opclass opc, pg_operator opr
        WHERE amop.amopid = am.oid AND
              amop.amopclaid = opc.oid AND
              amop.amopopr = opr.oid
        ORDER BY acc_name, ops_name, ops_comp
```

func_name

A user-defined function, which returns a value that can be indexed.

Outputs

CREATE

The message returned if the index is successfully created.

ERROR: Cannot create index: 'index_name' already exists.

This error occurs if it is impossible to create the index.

Description

`CREATE INDEX` constructs an index *index_name*. on the specified *table*.

Tip

Indexes are primarily used to enhance database performance. But inappropriate use will result in slower performance.

In the first syntax shown above, the key fields for the index are specified as column names; a column may also have an associated operator class. An operator class is used to specify the operators to be used for a particular index. For example, a btree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. The default operator class is the appropriate operator class for that field type.

In the second syntax, an index is defined on the result of a user-defined function *func_name* applied to one or more attributes of a single class. These functional indexes can be used to obtain fast access to data based on operators that would normally require some transformation to apply them to the base data.

Notes

Currently, only the BTREE access method supports multi-column indexes. Up to 7 keys may be specified.

Use `DROP INDEX` to remove an index.

Usage

To create a btree index on the field `title` in the table `films`:

```
CREATE UNIQUE INDEX title_idx
    ON films (title);
```

Compatibility

SQL92

CREATE INDEX is a Postgres language extension.

There is no `CREATE INDEX` command in SQL92.

Name

CREATE LANGUAGE — Defines a new language for functions

Synopsis

<refsynopsisdivinfo>1998-09-09</refsynopsisdivinfo>

```
CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'langname'
    HANDLER call_handler
    LANCOMPILER 'comment'
```

Inputs

`TRUSTED`

`TRUSTED` specifies that the call handler for the language is safe; that is, it offers an unprivileged user no functionality to bypass access restrictions. If this keyword is omitted when registering the language, only users with the Postgres superuser privilege can use this language to create new functions (like the 'C' language).

`langname`

The name of the new procedural language. The language name is case insensitive. A procedural language cannot override one of the built-in languages of Postgres.

HANDLER *`call_handler`*

`call_handler` is the name of a previously registered function that will be called to execute the PL procedures.

`comment`

The `LANCOMPILER` argument is the string that will be inserted in the `LANCOMPILER` attribute of the new `pg_language` entry. At present, Postgres does not use this attribute in any way.

Outputs

CREATE

This message is returned if the language is successfully created.

ERROR: PL handler function *`funcname`*() doesn't exist

This error is returned if the function *`funcname`*() is not found.

Description

Using `CREATE LANGUAGE`, a Postgres user can register a new language with Postgres. Subsequently, functions and trigger procedures can be defined in this new language. The user must have the Postgres superuser privilege to register a new language.

Writing PL handlers

The call handler for a procedural language must be written in a compiler language such as 'C' and registered with Postgres as a function taking no arguments and returning the `opaque` type, a placeholder for

unspecified or undefined types.. This prevents the call handler from being called directly as a function from queries.

However, arguments must be supplied on the actual call when a PL function or trigger procedure in the language offered by the handler is to be executed.

- When called from the trigger manager, the only argument is the object ID from the procedure's `pg_proc` entry. All other information from the trigger manager is found in the global `CurrentTriggerData` pointer.
- When called from the function manager, the arguments are the object ID of the procedure's `pg_proc` entry, the number of arguments given to the PL function, the arguments in a `FmgrValues` structure and a pointer to a boolean where the function tells the caller if the return value is the SQL NULL value.

It's up to the call handler to fetch the `pg_proc` entry and to analyze the argument and return types of the called procedure. The AS clause from the `CREATE FUNCTION` of the procedure will be found in the `prosrc` attribute of the `pg_proc` table entry. This may be the source text in the procedural language itself (like for PL/Tcl), a pathname to a file or anything else that tells the call handler what to do in detail.

Notes

Use `CREATE FUNCTION` to create a function.

Use `DROP LANGUAGE` to drop procedural languages.

Refer to the table `pg_language` for further information:

```
Table    = pg_language
+--------------------------+--------------------------+-------+
|          Field           |          Type            | Length|
+--------------------------+--------------------------+-------+
| lanname                  | name                     |    32 |
| lancompiler              | text                     |   var |
+--------------------------+--------------------------+-------+

lanname |lancompiler
--------+--------------
internal|n/a
lisp    |/usr/ucb/liszt
C       |/bin/cc
sql     |postgres
```

Restrictions

Since the call handler for a procedural language must be registered with Postgres in the 'C' language, it inherits all the capabilities and restrictions of 'C' functions.

Bugs

At present, the definitions for a procedural language cannot be changed once they have been created.

Usage

This is a template for a PL handler written in 'C':

```
#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.h"
#include "fmgr.h"        /* for FmgrValues struct */
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

Datum
plsample_call_handler(
     Oid       prooid,
     int       pronargs,
     FmgrValues     *proargs,
     bool      *isNull)
{
     Datum          retval;
     TriggerData    *trigdata;

     if (CurrentTriggerData == NULL) {
          /*
           * Called as a function
           */

          retval = ...
     } else {
          /*
           * Called as a trigger procedure
           */
          trigdata = CurrentTriggerData;
          CurrentTriggerData = NULL;

          retval = ...
     }

     *isNull = false;
     return retval;
}
```

Only a few thousand lines of code have to be added instead of the dots to complete the PL call handler. See `CREATE FUNCTION` for information on how to compile it into a loadable module .

The following commands then register the sample procedural language:

```
CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '/usr/local/pgsql/lib/plsample.so'
    LANGUAGE 'C';

CREATE PROCEDURAL LANGUAGE 'plsample'
    HANDLER plsample_call_handler
    LANCOMPILER 'PL/Sample';
```

Compatibility

CREATE LANGUAGE is a Postgres extension.

SQL92

There is no `CREATE LANGUAGE` statement in SQL92.

Name

CREATE OPERATOR — Defines a new user operator

Synopsis

<refsynopsisdivinfo>1998-09-09</refsynopsisdivinfo>

```
CREATE OPERATOR name (
        PROCEDURE  = func_name
     [, LEFTARG    = type1 ]
     [, RIGHTARG   = type2 ]
     [, COMMUTATOR = com_op ]
     [, NEGATOR    = neg_op ]
     [, RESTRICT   = res_proc ]
     [, HASHES ]
     [, JOIN       = join_proc ]
     [, SORT       = sort_op [, ...] ]
    )
```

Inputs

name

The operator to be defined. See below for allowable characters.

func_name

The function used to implement this operator.

type1

The type for the left-hand side of the operator, if any. This option would be omitted for a right-unary operator.

type2

The type for the right-hand side of the operator, if any. This option would be omitted for a left-unary operator.

com_op

The corresponding commutative operator.

neg_op

The corresponding negation operator.

res_proc

The corresponding restriction operator.

HASHES

This operator can support a hash-join algorithm.

join_proc

Procedure supporting table joins.

sort_op

Operator to use for sorting.

Outputs

CREATE

Message returned if the operator is successfully created.

Description

`CREATE OPERATOR` defines a new operator, *name*. The user who defines an operator becomes its owner.

The operator *name* is a sequence of up to thirty two (32) characters in any combination from the following:

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

Note

No alphabetic characters are allowed in an operator name. This enables Postgres to parse SQL input into tokens without requiring spaces between each token.

The operator "!=" is mapped to "<>" on input, so they are therefore equivalent.

At least one of LEFTARG and RIGHTARG must be defined. For binary operators, both should be defined. For right unary operators, only LEFTARG should be defined, while for left unary operators only RIGHTARG should be defined.

Also, the *func_name* procedure must have been previously defined using `CREATE FUNCTION` and must be defined to accept the correct number of arguments (either one or two).

The commutator operator is present so that Postgres can reverse the order of the operands if it wishes. For example, the operator area-less-than, <<<, would have a commutator operator, area-greater-than, >>>. Hence, the query optimizer could freely convert:

```
"0,0,1,1"::box  >>> MYBOXES.description
```

to

```
MYBOXES.description <<< "0,0,1,1"::box
```

This allows the execution code to always use the latter representation and simplifies the query optimizer some what.

Suppose that an operator, area-equal, ===, exists, as well as an area not equal, !==. The negator operator allows the query optimizer to convert

```
NOT MYBOXES.description === "0,0,1,1"::box
```

to

```
MYBOXES.description !== "0,0,1,1"::box
```

If a commutator operator name is supplied, Postgres searches for it in the catalog. If it is found and it does not yet have a commutator itself, then the commutator's entry is updated to have the current (new) operator as its commutator. This applies to the negator, as well.

This is to allow the definition of two operators that are the commutators or the negators of each other. The first operator should be defined without a commutator or negator (as appropriate). When the second operator is defined, name the first as the commutator or negator. The first will be updated as a side effect.

The next two specifications are present to support the query optimizer in performing joins. Postgres can always evaluate a join (i.e., processing a clause with two tuple variables separated by an operator that returns a boolean) by iterative substitution [WONG76]. In addition, Postgres is planning on implementing a hash-join algorithm along the lines of [SHAP86]; however, it must know whether this strategy is applicable. For example, a hash-join algorithm is usable for a clause of the form:

```
MYBOXES.description === MYBOXES2.description
```

but not for a clause of the form:

```
MYBOXES.description <<< MYBOXES2.description.
```

The HASHES flag gives the needed information to the query optimizer concerning whether a hash join strategy is usable for the operator in question.

Similarly, the two sort operators indicate to the query optimizer whether merge-sort is a usable join strategy and what operators should be used to sort the two operand classes. For the === clause above, the optimizer must sort both relations using the operator, <<<. On the other hand, merge-sort is not usable with the clause:

```
MYBOXES.description <<< MYBOXES2.description
```

If other join strategies are found to be practical, Postgres will change the optimizer and run-time system to use them and will require additional specification when an operator is defined. Fortunately, the research community invents new join strategies infrequently, and the added generality of user-defined join strategies was not felt to be worth the complexity involved.

The last two pieces of the specification are present so the query optimizer can estimate result sizes. If a clause of the form:

```
MYBOXES.description <<< "0,0,1,1"::box
```

is present in the qualification, then Postgres may have to estimate the fraction of the instances in MYBOXES that satisfy the clause. The function *res_proc* must be a registered function (meaning it is already defined using define function(l)) which accepts one argument of the correct data type and returns a floating point number. The query optimizer simply calls this function, passing the parameter "0,0,1,1" and multiplies the result by the relation size to get the desired expected number of instances.

Similarly, when the operands of the operator both contain instance variables, the query optimizer must estimate the size of the resulting join. The function join_proc will return another floating point number which will be multiplied by the cardinalities of the two classes involved to compute the desired expected result size.

The difference between the function

```
my_procedure_1 (MYBOXES.description, "0,0,1,1"::box)
```

and the operator

```
MYBOXES.description === "0,0,1,1"::box
```

is that Postgres attempts to optimize operators and can decide to use an index to restrict the search space when operators are involved. However, there is no attempt to optimize functions, and they are performed by brute force. Moreover, functions can have any number of arguments while operators are restricted to one or two.

Notes

Refer to the chapter on operators in the *PostgreSQL User's Guide* for further information. Refer to `DROP OPERATOR` to delete user-defined operators from a database.

Usage

The following command defines a new operator, area-equality, for the BOX data type.

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    HASHES,
    JOIN = area-join-procedure,
    SORT = <<<, <<<)
```

Compatibility

CREATE OPERATOR is a Postgres extension.

SQL92

There is no CREATE OPERATOR statement in SQL92.

Name

CREATE RULE — Defines a new rule

Synopsis

<refsynopsisdivinfo>1998-09-11</refsynopsisdivinfo>

```
CREATE RULE name
    AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] [ action | NOTHING ]
```

Inputs

name

The name of a rule to create.

event

Event is one of `select`, `update`, `delete` or `insert`.

object

Object is either *table* or *table.column*.

condition

Any SQL WHERE clause. `new` or `current` can appear instead of an instance variable whenever an instance variable is permissible in SQL.

action

Any SQL statement. `new` or `current` can appear instead of an instance variable whenever an instance variable is permissible in SQL.

Outputs

CREATE

Message returned if the rule is successfully created.

Description

The semantics of a rule is that at the time an individual instance is accessed, updated, inserted or deleted, there is a current instance (for retrieves, updates and deletes) and a new instance (for updates and appends). If the *event* specified in the ON clause and the *condition* specified in the WHERE clause are true for the current instance, the *action* part of the rule is executed. First, however, values from fields in the current instance and/or the new instance are substituted for `current.`*attribute-name* and `new.`*attribute-name*.

The *action* part of the rule executes with the same command and transaction identifier as the user command that caused activation.

Notes

A caution about SQL rules is in order. If the same class name or instance variable appears in the *event*, the *condition* and the *action* parts of a rule, they are all considered different tuple variables. More accurately, new and current are the only tuple variables that are shared between these clauses. For example, the following two rules have the same semantics:

```
on update to EMP.salary where EMP.name = "Joe"
    do update EMP ( ... ) where ...

on update to EMP-1.salary where EMP-2.name = "Joe"
    do update EMP-3 ( ... ) where ...
```

Each rule can have the optional tag INSTEAD. Without this tag, *action* will be performed in addition to the user command when the *event* in the *condition* part of the rule occurs. Alternately, the *action* part will be done instead of the user command. In this later case, the *action* can be the keyword NOTHING.

When choosing between the rewrite and instance rule systems for a particular rule application, remember that in the rewrite system, current refers to a relation and some qualifiers whereas in the instance system it refers to an instance (tuple).

It is very important to note that the rewrite rule system will neither detect nor process circular rules. For example, though each of the following two rule definitions are accepted by Postgres, the retrieve command will cause Postgres to crash:

Example 19.1. Example of a circular rewrite rule combination.

```
create rule bad_rule_combination_1 is
    on select to EMP
    do instead select to TOYEMP

create rule bad_rule_combination_2 is
    on select to TOYEMP
    do instead select to EMP
```

This attempt to retrieve from EMP will cause Postgres to crash.

```
select * from EMP
```

You must have rule definition access to a class in order to define a rule on it. Use GRANT and REVOKE to change permissions.

Usage

Make Sam get the same salary adjustment as Joe:

```
create rule example_1 is
    on update EMP.salary where current.name = "Joe"
    do update EMP (salary = new.salary)
    where EMP.name = "Sam"
```

At the time Joe receives a salary adjustment, the event will become true and Joe's current instance and proposed new instance are available to the execution routines. Hence, his new salary is substituted into the action part of the rule which is subsequently executed. This propagates Joe's salary on to Sam.

Make Bill get Joe's salary when it is accessed:

```
create rule example_2 is
    on select to EMP.salary
    where current.name = "Bill"
    do instead
    select (EMP.salary) from EMP
        where EMP.name = "Joe"
```

Deny Joe access to the salary of employees in the shoe department (`current_user` returns the name of the current user):

```
create rule example_3 is
    on select to EMP.salary
    where current.dept = "shoe" and current_user = "Joe"
    do instead nothing
```

Create a view of the employees working in the toy department.

```
create TOYEMP(name = char16, salary = int4)

create rule example_4 is
    on select to TOYEMP
    do instead
    select (EMP.name, EMP.salary) from EMP
        where EMP.dept = "toy"
```

All new employees must make 5,000 or less

```
create rule example_5 is
    on insert to EMP where new.salary > 5000
    do update newset salary = 5000
```

Bugs

The object in a SQL rule cannot be an array reference and cannot have parameters.

Aside from the "oid" field, system attributes cannot be referenced anywhere in a rule. Among other things, this means that functions of instances (e.g., "`foo(emp)`" where "`emp`" is a class) cannot be called anywhere in a rule.

The rule system stores the rule text and query plans as text attributes. This implies that creation of rules may fail if the rule plus its various internal representations exceed some value that is on the order of one page (8KB).

Compatibility

CREATE RULE statement is a Postgres language extension.

SQL92

There is no CREATE RULE statement in SQL92.

Name

CREATE SEQUENCE — Creates a new sequence number generator

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
CREATE SEQUENCE seqname
    [ INCREMENT increment ]
    [ MINVALUE  minvalue ]
    [ MAXVALUE  maxvalue ]
    [ START     start ]
    [ CACHE     cache ]
    [ CYCLE ]
```

Inputs

seqname

The name of a sequence to be created.

increment

The INCREMENT *increment* clause is optional. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is one (1).

minvalue

The optional clause MINVALUE *minvalue* determines the minimum value a sequence can generate. The defaults are 1 and -2147483647 for ascending and descending sequences, respectively.

maxvalue

Use the optional clause MAXVALUE *maxvalue* to determine the maximum value for the sequence. The defaults are 2147483647 and -1 for ascending and descending sequences, respectively.

start

The optional START *start* clause enables the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

The CACHE *cache* option enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e. no cache) and this is also the default.

CYCLE

The optional CYCLE keyword may be used to enable the sequence to continue when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be whatever the *minvalue* or *maxvalue* is, as appropriate.

Outputs

CREATE

Message returned if the command is successful.

ERROR: amcreate: '*seqname*' relation already exists

If the sequence specified already exists.

ERROR: DefineSequence: START value (*start*) can't be > MAXVALUE (*maxvalue*)

If the specified starting value is out of range.

ERROR: DefineSequence: START value (*start*) can't be < MINVALUE (*minvalue*)

If the specified starting value is out of range.

ERROR: DefineSequence: MINVALUE (*minvalue*) can't be >= MAXVALUE (*maxvalue*)

If the minimum and maximum values are inconsistant.

Description

CREATE SEQUENCE will enter a new sequence number generator into the current data base. This involves creating and initialising a new single-row table with the name *seqname*. The generator will be "owned" by the user issuing the command.

After a sequence is created, you may use the function `nextval(`*seqname*`)` to get a new number from the sequence. The function `currval('`*seqname*`')` may be used to determine the number returned by the last call to `nextval(`*seqname*`)` for the specified sequence in the current session. The function `setval('`*seqname*`', `*newvalue*`)` may be used to set the current value of the specified sequence. The next call to `nextval(`*seqname*`)` will return the given value plus the sequence increment.

Use a query like

```
SELECT * FROM sequence_name;
```

to get the parameters of a sequence. Aside from fetching the original parameters, you can use

```
SELECT last_value FROM sequence_name;
```

to obtain the last value allocated by any backend. parameters, you can use

Low-level locking is used to enable multiple simultaneous calls to a generator.

Caution

Unexpected results may be obtained if a cache setting greater than one is used for a sequence object that will be used concurrently by multiple backends. Each backend will allocate "cache" successive sequence values during one access to the sequence object and increase the sequence object's last_value accordingly. Then, the next cache-1 uses of nextval within that backend simply return the preallocated values without touching the shared object. So, numbers allocated but not used in the current session will be lost. Furthermore, although multiple backends are

guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the backends are considered. (For example, with a cache setting of 10, backend A might reserve values 1..10 and return nextval=1, then backend B might reserve values 11..20 and return nextval=11 before backend A has generated nextval=2.) Thus, with a cache setting of one it is safe to assume that nextval values are generated sequentially; with a cache setting greater than one you should only assume that the nextval values are all distinct, not that they are generated purely sequentially. Also, last_value will reflect the latest value reserved by any backend, whether or not it has yet been returned by nextval.

Notes

Refer to the DROP SEQUENCE statement to remove a sequence.

Each backend uses its own cache to store allocated numbers. Numbers that are cached but not used in the current session will be lost, resulting in "holes" in the sequence.

Usage

Create an ascending sequence called `serial`, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence

```
SELECT NEXTVAL ('serial');

nextval
-------
    114
```

Use this sequence in an INSERT:

```
INSERT INTO distributors VALUES (NEXTVAL('serial'),'nothing');
```

Set the sequence value after a COPY FROM:

```
CREATE FUNCTION distributors_id_max() RETURNS INT4
       AS 'SELECT max(id) FROM distributors'
       LANGUAGE 'sql';
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', distributors_id_max());
END;
```

Compatibility

`CREATE SEQUENCE` is a Postgres language extension.

SQL92

There is no `CREATE SEQUENCE` statement in SQL92.

Name

CREATE TABLE — Creates a new table

Synopsis

<refsynopsisdivinfo>1998-09-11</refsynopsisdivinfo>

```
CREATE TABLE table (
    column type
    [ DEFAULT value]
    [, NOT NULL ] [ ,UNIQUE ]
    [column_constraint_clause | PRIMARY KEY } [ ... ] ]
    [, ... ]
    [, PRIMARY KEY ( column [, ...] ) ]
    [, CHECK ( condition ) ]
    [, table_constraint_clause ]
    ) [ INHERITS ( inherited_table [, ...] ) ]
```

Inputs

table

The name of a new table to be created.

column

The name of a column.

type

The type of the column. This may include array specifiers. Refer to the *PostgreSQL User's Guide* for further information about data types and arrays.

DEFAULT *value*

A default value for a column. See the DEFAULT clause for more information.

column_constraint_clause

The optional column constraint clauses specify a list of integrity constraints or tests which new or updated entries must satisfy for an insert or update operation to succeed. Each constraint must evaluate to a boolean expression. Although SQL92 requires the *column_constraint_clause* to refer to that column only, Postgres allows multiple columns to be referenced within a single column constraint. See the column constraint clause for more information.

table_constraint_clause

The optional table CONSTRAINT clause specifies a list of integrity constraints which new or updated entries must satisfy for an insert or update operation to succeed. Each constraint must evaluate to a boolean expression. Multiple columns may be referenced within a single constraint. Only one PRIMARY KEY clause may be specified for a table; PRIMARY KEY *column* (a table constraint) and PRIMARY KEY (a column constraint) are mutually exclusive.. See the table constraint clause for more information.

INHERITS *inherited_table*

The optional INHERITS clause specifies a collection of table names from which this table automatically inherits all fields. If any inherited field name appears more than once, Postgres reports an error. Postgres automatically allows the created table to inherit functions on tables above it in the inheritance hierarchy.

Aside

Inheritance of functions is done according to the conventions of the Common Lisp Object System (CLOS).

Outputs

CREATE

Message returned if table is successfully created.

ERROR

Message returned if table creation failed. This is usually accompanied by some descriptive text, such as:

```
amcreate: "table" relation already exists
```

which occurs at runtime, if the table specified already exists in the database.

ERROR: DEFAULT: type mismatched

if data type of default value doesn't match the column definition's data type.

Description

`CREATE TABLE` will enter a new table into the current data base. The table will be "owned" by the user issuing the command.

The new table is created as a heap with no initial data. A table can have no more than 1600 columns (realistically, this is limited by the fact that tuple sizes must be less than 8192 bytes), but this limit may be configured lower at some sites. A table cannot have the same name as a system catalog table.

DEFAULT Clause

`DEFAULT` *value*

Inputs

value

The possible values for the default value expression are:

- a literal value
- a user function

- a niladic function

Outputs

Description

The DEFAULT clause assigns a default data value to a column (via a column definition in the CREATE TABLE statement). The data type of a default value must match the column definition's data type.

An INSERT operation that includes a column without a specified default value will assign the NULL value to the column if no explicit data value is provided for it. Default *literal* means that the default is the specified constant value. Default *niladic-function* or *user-function* means that the default is the value of the specified function at the time of the INSERT.

There are two types of niladic functions:

niladic USER

CURRENT_USER / USER

See CURRENT_USER function

SESSION_USER

not yet supported

SYSTEM_USER

not yet supported

niladic datetime

CURRENT_DATE

See CURRENT_DATE function

CURRENT_TIME

See CURRENT_TIME function

CURRENT_TIMESTAMP

See CURRENT_TIMESTAMP function

In the current release (v6.4), Postgres evaluates all default expressions at the time the table is defined. Hence, functions which are "non-cacheable" such as `CURRENT_TIMESTAMP` may not produce the desired effect. For the particular case of date/time types, one can work around this behavior by using “DEFAULT TEXT 'now'” instead of “DEFAULT 'now'” or “DEFAULT CURRENT_TIMESTAMP”. This forces Postgres to consider the constant a string type and then to convert the value to `timestamp` at runtime.

Usage

To assign a constant value as the default for the columns `did` and `number`, and a string literal to the column `did`:

```
CREATE TABLE video_sales (
    did      VARCHAR(40) DEFAULT 'luso films',
```

```
    number    INTEGER DEFAULT 0,
    total     CASH DEFAULT '$0.0'
);
```

To assign an existing sequence as the default for the column `did`, and a literal to the column `name`:

```
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT NEXTVAL('serial'),
    name     VARCHAR(40) DEFAULT 'luso films'
);
```

Column CONSTRAINT Clause

```
[ CONSTRAINT name ] { NOT NULL | UNIQUE | PRIMARY KEY |
 CHECK constraint } [, ...]
```

Inputs

name

An arbitrary name given to the integrity constraint. If *name* is not specified, it is generated from the table and column names, which should ensure uniqueness for *name*.

NOT NULL

The column is not allowed to contain NULL values. This is equivalent to the column constraint CHECK (*column* NOT NULL).

UNIQUE

The column must have unique values. In Postgres this is enforced by an implicit creation of a unique index on the table.

PRIMARY KEY

This column is a primary key, which implies that uniqueness is enforced by the system and that other tables may rely on this column as a unique identifier for rows. See PRIMARY KEY for more information.

constraint

The definition of the constraint.

Description

A Constraint is a named rule: an SQL object which helps define valid sets of values by putting limits on the results of INSERT, UPDATE or DELETE operations performed on a Base Table.

There are two ways to define integrity constraints: table constraints, covered later, and column constraints, covered here.

A column constraint is an integrity constraint defined as part of a column definition, and logically becomes a table constraint as soon as it is created. The column constraints available are:

PRIMARY KEY
REFERENCES
UNIQUE
CHECK
NOT NULL

Note

Postgres does not yet (at release 6.4) support REFERENCES integrity constraints. The parser accepts the REFERENCES syntax but ignores the clause.

NOT NULL Constraint

```
[ CONSTRAINT name ] NOT NULL
```

The NOT NULL constraint specifies a rule that a column may contain only non-null values.

The NOT NULL constraint is a column constraint only, and not allowed as a table constraint.

Outputs

status

ERROR: ExecAppend: Fail to add null value in not null attribute "*column*".

This error occurs at runtime if one tries to insert a null value into a column which has a NOT NULL constraint.

Description

Usage

Define two NOT NULL column constraints on the table `distributors`, one of which being a named constraint:

```
CREATE TABLE distributors (
    did      DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name     VARCHAR(40) NOT NULL
);
```

UNIQUE Constraint

```
[ CONSTRAINT name ] UNIQUE
```

Inputs

CONSTRAINT *name*

An arbitrary label given to a constraint.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index.

This error occurs at runtime if one tries to insert a duplicate value into a column.

Description

The UNIQUE constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique values.

The column definitions of the specified columns do not have to include a NOT NULL constraint to be included in a UNIQUE constraint. Having more than one null value in a column without a NOT NULL constraint, does not violate a UNIQUE constraint. (This deviates from the SQL92 definition, but is a more sensible convention. See the section on compatibility for more details.).

Each UNIQUE column constraint must name a column that is different from the set of columns named by any other UNIQUE or PRIMARY KEY constraint defined for the table.

Note

Postgres automatically creates a unique index for each UNIQUE constraint, to assure data integrity. See CREATE INDEX for more information.

Usage

Defines a UNIQUE column constraint for the table distributors. UNIQUE column constraints can only be defined on one column of the table:

```
CREATE TABLE distributors (
     did      DECIMAL(3),
     name     VARCHAR(40) UNIQUE
     );
```

which is equivalent to the following specified as a table constraint:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

The CHECK Constraint

```
[ CONSTRAINT name ] CHECK ( condition [, ...] )
```

Inputs

name

An arbitrary name given to a constraint.

condition

Any valid conditional expression evaluating to a boolean result.

Outputs

status

ERROR: ExecAppend: rejected due to CHECK constraint "*table_column*".

This error occurs at runtime if one tries to insert an illegal value into a column subject to a CHECK constraint.

Description

The CHECK constraint specifies a restriction on allowed values within a column. The CHECK constraint is also allowed as a table constraint.

The SQL92 CHECK column constraints can only be defined on, and refer to, one column of the table. Postgres does not have this restriction.

PRIMARY KEY Constraint

```
[ CONSTRAINT name ] PRIMARY KEY
```

Inputs

CONSTRAINT *name*

An arbitrary name for the constraint.

Outputs

ERROR: Cannot insert a duplicate key into a unique index.

This occurs at run-time if one tries to insert a duplicate value into a column subject to a PRIMARY KEY constraint.

Description

The PRIMARY KEY column constraint specifies that a column of a table may contain only unique (non-duplicate), non-NULL values. The definition of the specified column does not have to include an explicit NOT NULL constraint to be included in a PRIMARY KEY constraint.

Only one PRIMARY KEY can be specified for a table.

Notes

Postgres automatically creates a unique index to assure data integrity. (See CREATE INDEX statement)

The PRIMARY KEY constraint should name a set of columns that is different from other sets of columns named by any UNIQUE constraint defined for the same table, since it will result in duplication of equivalent indexes and unproductive additional runtime overhead. However, Postgres does not specifically disallow this.

Table CONSTRAINT Clause

```
[ CONSTRAINT name ] { PRIMARY KEY |  UNIQUE } ( column [, ...] )
[ CONSTRAINT name ] CHECK ( constraint )
```

Inputs

CONSTRAINT *name*

An arbitrary name given to an integrity constraint.

column [, ...]

The column name(s) for which to define a unique index and, for PRIMARY KEY, a NOT NULL constraint.

CHECK (*constraint*)

A boolean expression to be evaluated as the constraint.

Outputs

The possible outputs for the table constraint clause are the same as for the corresponding portions of the column constraint clause.

Description

A table constraint is an integrity constraint defined on one or more columns of a base table. The four variations of "Table Constraint" are:

UNIQUE
CHECK
PRIMARY KEY
FOREIGN KEY

Note

Postgres does not yet (as of version 6.4) support FOREIGN KEY integrity constraints. The parser understands the FOREIGN KEY syntax, but only prints a notice and otherwise ignores the clause. Foreign keys may be partially emulated by triggers (See the CREATE TRIGGER statement).

UNIQUE Constraint

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
```

Inputs

CONSTRAINT *name*

An arbitrary name given to a constraint.

column

A name of a column in a table.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index.

This error occurs at runtime if one tries to insert a duplicate value into a column.

Description

The UNIQUE constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique values. The behavior of the UNIQUE table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

See the section on the UNIQUE column constraint for more details.

Usage

Define a UNIQUE table constraint for the table distributors:

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

PRIMARY KEY Constraint

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
```

Inputs

CONSTRAINT *name*

An arbitrary name for the constraint.

column [, ...]

The names of one or more columns in the table.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index.

This occurs at run-time if one tries to insert a duplicate value into a column subject to a PRIMARY KEY constraint.

Description

The PRIMARY KEY constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique, (non duplicate), non-null values. The column definitions of the specified columns do not have to include a NOT NULL constraint to be included in a PRIMARY KEY constraint. The

PRIMARY KEY table constraint is similar to that for column constraints, with the additional capability of encompassing multiple columns.

Refer to the section on the PRIMARY KEY column constraint for more information.

Usage

Create table films and table distributors

```
CREATE TABLE films (
       code      CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
       title     CHARACTER VARYING(40) NOT NULL,
       did       DECIMAL(3) NOT NULL,
       date_prod DATE,
       kind      CHAR(10),
       len       INTERVAL HOUR TO MINUTE
       );


CREATE TABLE distributors (
       did      DECIMAL(03) PRIMARY KEY DEFAULT NEXTVAL('serial'),
       name     VARCHAR(40) NOT NULL CHECK (name <> '')
       );
```

Create a table with a 2-dimensional array

```
CREATE TABLE array (
       vector INT[][]
       );
```

Define a UNIQUE table constraint for the table films. UNIQUE table constraints can be defined on one or more columns of the table

```
CREATE TABLE films (
    code      CHAR(5),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
    );
```

Define a CHECK column constraint.

```
CREATE TABLE distributors (
     did      DECIMAL(3) CHECK (did > 100),
     name     VARCHAR(40)
     );
```

Define a CHECK table constraint

```
CREATE TABLE distributors (
     did      DECIMAL(3),
     name     VARCHAR(40)
     CONSTRAINT con1 CHECK (did > 100 AND name > '')
     );
```

Define a PRIMARY KEY table constraint for the table films. PRIMARY KEY table constraints can be defined on one or more columns of the table

```
CREATE TABLE films (
    code      CHAR(05),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
    );
```

Defines a PRIMARY KEY column constraint for table distributors. PRIMARY KEY column constraints can only be defined on one column of the table (the following two examples are equivalent)

```
CREATE TABLE distributors (
     did      DECIMAL(03),
     name     CHAR VARYING(40),
     PRIMARY KEY(did)
     );

CREATE TABLE distributors (
     did      DECIMAL(03) PRIMARY KEY,
     name     VARCHAR(40)
     );
```

Notes

CREATE TABLE/INHERITS is a Postgres language extension.

Compatibility

SQL92

In addition to the normal CREATE TABLE, SQL92 also defines a CREATE TEMPORARY TABLE statement:

```
CREATE [ {GLOBAL | LOCAL} ] TEMPORARY TABLE table (
     column type [DEFAULT value] [CONSTRAINT column_constraint]
[, ...] )
       [CONSTRAINT table_constraint ]
       [ ON COMMIT {DELETE | PRESERVE} ROWS ]
```

For temporary tables, the CREATE TEMPORARY TABLE statement names a new table and defines the table's columns and constraints.

The optional ON COMMIT clause of CREATE TEMPORARY TABLE specifies whether or not the temporary table should be emptied of rows whenever COMMIT is executed. If the ON COMMIT clause is omitted, the default option, ON COMMIT DELETE ROWS, is assumed.

To create a temporary table:

```
CREATE TEMPORARY TABLE actors (
       id      DECIMAL(03),
       name    VARCHAR(40),
       CONSTRAINT actor_id CHECK (id < 150)
       ) ON COMMIT DELETE ROWS
```

Temporary tables are not currently available in Postgres.

Tip

In the current release of Postgres (v6.4), to create a temporary table you must create and drop the table by explicit commands.

UNIQUE clause

SQL92 specifies some additional capabilities for UNIQUE:

Table Constraint definition

```
[ CONSTRAINT name ]
UNIQUE ( column [, ...] )
[ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
[ [ NOT ] DEFERRABLE ]
```

Column Constraint definition

```
[ CONSTRAINT name ]
UNIQUE
[ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
[ [ NOT ] DEFERRABLE ]
```

NOT NULL clause

SQL92 specifies some additional capabilities for NOT NULL:

```
[ CONSTRAINT name ] NOT NULL
[ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
[ [ NOT ] DEFERRABLE ]
```

CONSTRAINT clause

SQL92 specifies some additional capabilities for constraints, and also defines assertions and domain constraints.

Note

Postgres does not yet support either domains or assertions.

An assertion is a special type of integrity constraint and share the same namespace as other constraints. However, an assertion is not necessarily dependent on one particular base table as constraints are, so SQL-92 provides the CREATE ASSERTION statement as an alternate method for defining a constraint:

```
CREATE ASSERTION name CHECK ( condition )
```

Domain constraints are defined by CREATE DOMAIN or ALTER DOMAIN statements:

Domain constraint:

```
[ CONSTRAINT name ]
    CHECK constraint
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Table constraint definition:

```
[ CONSTRAINT name ]
  { PRIMARY KEY constraint |
    FOREIGN KEY constraint |
    UNIQUE constraint |
    CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Column constraint definition:

```
[ CONSTRAINT name ]
   { NOT NULL constraint |
     PRIMARY KEY constraint |
     FOREIGN KEY constraint |
     UNIQUE constraint |
     CHECK constraint }
     [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
     [ [ NOT ] DEFERRABLE ]
```

A CONSTRAINT definition may contain one deferment attribute clause and/or one initial constraint mode clause, in any order.

NOT DEFERRABLE

means that the Constraint must be checked for violation of its rule after the execution of every SQL statement.

DEFERRABLE

means that checking of the Constraint may be deferred until some later time, but no later than the end of the current transaction.

The constraint mode for every Constraint always has an initial default value which is set for that Constraint at the beginning of a transaction.

INITIALLY IMMEDIATE

means that, as of the start of the transaction, the Constraint must be checked for violation of its rule after the execution of every SQL statement.

INITIALLY DEFERRED

means that, as of the start of the transaction, checking of the Constraint may be deferred until some later time, but no later than the end of the current transaction.

CHECK clause

SQL92 specifies some additional capabilities for CHECK in either table or column constraints.

table constraint definition:

```
[ CONSTRAINT name ]
 CHECK ( VALUE condition )
 [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
 [ [ NOT ] DEFERRABLE ]
```

column constraint definition:

```
 [ CONSTRAINT name ]
 CHECK ( VALUE condition )
 [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
 [ [ NOT ] DEFERRABLE ]
```

PRIMARY KEY clause

SQL92 specifies some additional capabilities for PRIMARY KEY:

Table Constraint definition:

```
 [ CONSTRAINT name ]
 PRIMARY KEY ( column [, ...] )
 [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
 [ [ NOT ] DEFERRABLE ]
```

Column Constraint definition:

```
 [ CONSTRAINT name ]
 PRIMARY KEY
 [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
 [ [ NOT ] DEFERRABLE ]
```

Name

CREATE TABLE AS — Creates a new table

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
CREATE TABLE table [ ( column [, ...] ) ] AS select_clause
```

Inputs

table

The name of a new table to be created.

column

The name of a column. Multiple column names can be specified using a comma-delimited list of column names.

select_clause

A valid query statement. Refer to SELECT for a description of the allowed syntax.

Outputs

Refer to CREATE TABLE and SELECT for a summary of possible output messages.

Description

CREATE TABLE AS enables a table to be created from the contents of an existing table. It has functionality equivalent to SELECT TABLE INTO, but with perhaps a more obvious syntax.

Name

CREATE TRIGGER — Creates a new trigger

Synopsis

<refsynopsisdivinfo>1998-09-21</refsynopsisdivinfo>

```
CREATE TRIGGER name { BEFORE | AFTER }
    { event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE funcname ( arguments )
```

Inputs

name

The name of an existing trigger.

table

The name of a table.

event

One of INSERT, DELETE or UPDATE.

funcname

A user-supplied function.

Outputs

CREATE

This message is returned if the trigger is successfully created.

Description

`CREATE TRIGGER` will enter a new trigger into the current data base. The trigger will be associated with the relation *relname* and will execute the specified function *funcname*.

The trigger can be specified to fire either before the operation is attempted on a tuple (before constraints are checked and the INSERT, UPDATE or DELETE is attempted) or after the operation has been attempted (e.g. after constraints are checked and the INSERT, UPDATE or DELETE has completed). If the trigger fires before the event, the trigger may skip the operation for the current tuple, or change the tuple being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the last insertion, update, or deletion, are "visible" to the trigger.

Refer to the chapters on SPI and Triggers in the *PostgreSQL Programmer's Guide* for more information.

Notes

`CREATE TRIGGER` is a Postgres language extension.

Only the relation owner may create a trigger on this relation.

As of the current release (v6.4), STATEMENT triggers are not implemented.

Refer to `DROP TRIGGER` for information on how to remove triggers.

Usage

Check if the specified distributor code exists in the distributors table before appending or updating a row in the table films:

```
CREATE TRIGGER if_dist_exists
    BEFORE INSERT OR UPDATE ON films FOR EACH ROW
    EXECUTE PROCEDURE check_primary_key ('did', 'distributors',
 'did');
```

Before cancelling a distributor or updating its code, remove every reference to the table films:

```
CREATE TRIGGER if_film_exists
    BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
    EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did', 'films',
 'did');
```

Compatibility

SQL92

There is no `CREATE TRIGGER` in SQL92.

The second example above may also be done by using a FOREIGN KEY constraint as in:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    CONSTRAINT if_film_exists
        FOREIGN KEY(did) REFERENCES films
            ON UPDATE CASCADE ON DELETE CASCADE
);
```

However, foreign keys are not yet implemented (as of version 6.4) in Postgres.

Name

CREATE TYPE — Defines a new base data type

Synopsis

<refsynopsisdivinfo>1998-09-21</refsynopsisdivinfo>

```
CREATE TYPE typename (
        INPUT          = input_function
      , OUTPUT         = output_function
      , INTERNALLENGTH = (internallength | VARIABLE)
    [ , EXTERNALLENGTH = (externallength | VARIABLE) ]
    [ , ELEMENT        = element ]
    [ , DELIMITER      = delimiter ]
    [ , DEFAULT        = "default" ]
    [ , SEND           = send_function ]
    [ , RECEIVE        = receive_function ]
    [ , PASSEDBYVALUE ]
)
```

Inputs

typename

The name of a type to be created.

INTERNALLENGTH *internallength*

A literal value, which specifies the internal length of the new type.

EXTERNALLENGTH *externallength*

A literal value, which specifies the external length of the new type.

INPUT *input_function*

The name of a function, created by CREATE FUNCTION, which converts data from its external form to the type's internal form.

OUTPUT *output_function*

The name of a function, created by CREATE FUNCTION, which converts data from its internal form to a form suitable for display.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character for the array.

default

The default text to be displayed to indicate "data not present"

send_function

The name of a function, created by CREATE FUNCTION, which converts data of this type into a form suitable for transmission to another machine.

receive_function

The name of a function, created by CREATE FUNCTION, which converts data of this type from a form suitable for transmission from another machine to internal form.

Outputs

CREATE

Message returned if the type is successfully created.

Description

CREATE TYPE allows the user to register a new user data type with Postgres for use in the current data base. The user who defines a type becomes its owner. *Typename* is the name of the new type and must be unique within the types defined for this database.

CREATE TYPE requires the registration of two functions (using create function) before defining the type. The representation of a new base type is determined by *input_function*, which converts the type's external representation to an internal representation usable by the operators and functions defined for the type. Naturally, *output_function* performs the reverse transformation. Both the input and output functions must be declared to take one or two arguments of type "opaque".

New base data types can be fixed length, in which case *internallength* is a positive integer, or variable length, in which case Postgres assumes that the new type has the same format as the Postgres-supplied data type, "text". To indicate that a type is variable-length, set *internallength* to VARIABLE. The external representation is similarly specified using the *externallength* keyword.

To indicate that a type is an array and to indicate that a type has array elements, indicate the type of the array element using the element keyword. For example, to define an array of 4 byte integers ("int4"), specify

```
ELEMENT = int4
```

To indicate the delimiter to be used on arrays of this type, *delimiter* can be set to a specific character. The default delimiter is the comma (",").

A default value is optionally available in case a user wants some specific bit pattern to mean "data not present." Specify the default with the DEFAULT keyword.

The optional functions *send_function* and *receive_function* are used when the application program requesting Postgres services resides on a different machine. In this case, the machine on which Postgres runs may use a format for the data type different from that used on the remote machine. In this case it is appropriate to convert data items to a standard form when sending from the server to the client and converting from the standard format to the machine specific format when the server receives the data from the client. If these functions are not specified, then it is assumed that the internal format of the type is acceptable on all relevant machine architectures. For example, single characters do not have to be converted if passed from a Sun-4 to a DECstation, but many other types do.

The optional flag, PASSEDBYVALUE, indicates that operators and functions which use this data type should be passed an argument by value rather than by reference. Note that you may not pass by value types whose internal representation is more than four bytes.

For new base types, a user can define operators, functions and aggregates using the appropriate facilities described in this section.

Array Types

Two generalized built-in functions, array_in and array_out, exist for quick creation of variable-length array types. These functions operate on arrays of any existing Postgres type.

Large Object Types

A "regular" Postgres type can only be 8192 bytes in length. If you need a larger type you must create a Large Object type. The interface for these types is discussed at length in Section 7, the large object interface. The length of all large object types is always VARIABLE.

Examples

This command creates the box data type and then uses the type in a class definition:

```
CREATE TYPE box (INTERNALLENGTH = 8,
     INPUT = my_procedure_1, OUTPUT = my_procedure_2)

CREATE TABLE myboxes (id INT4, description box)
```

This command creates a variable length array type with integer elements.

```
CREATE TYPE int4array
   (INPUT = array_in, OUTPUT = array_out,
    INTERNALLENGTH = VARIABLE, ELEMENT = int4)

CREATE TABLE myarrays (id int4, numbers int4array)
```

This command creates a large object type and uses it in a class definition.

```
CREATE TYPE bigobj
   (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE)

CREATE TABLE big_objs (id int4, obj bigobj)
```

Restrictions

Type names cannot begin with the underscore character ("_") and can only be 15 characters long. This is because Postgres silently creates an array type for each base type with a name consisting of the base type's name prepended with an underscore.

Notes

Refer to `DROP TYPE` to remove an existing type.

See also `CREATE FUNCTION`, `CREATE OPERATOR` and the chapter on Large Objects in the *PostgreSQL Programmer's Guide*.

Compatibility

SQL3

`CREATE TYPE` is an SQL3 statement.

Name

CREATE USER — Creates account information for a new user

Synopsis

<refsynopsisdivinfo>1998-09-21</refsynopsisdivinfo>

```
CREATE USER username
    [ WITH PASSWORD password ]
    [ CREATEDB    | NOCREATEDB ]
    [ CREATEUSER | NOCREATEUSER ]
    [ IN GROUP     groupname [, ...] ]
    [ VALID UNTIL  'abstime' ]
```

Inputs

username

The name of the user.

password

The WITH PASSWORD clause sets the user's password within the "pg_shadow" table. For this reason, "pg_shadow" is no longer accessible to the instance of Postgres that the Postgres user's password is initially set to NULL. When a user's password in the "pg_shadow" table is NULL, user authentication proceeds as it historically has (HBA, PG_PASSWORD, etc). However, if a password is set for a user, a new authentication system supplants any other configured for the Postgres instance, and the password stored in the "pg_shadow" table is used for authentication. For more details on how this authentication system functions see pg_crypt(3). If the WITH PASSWORD clause is omitted, the user's password is set to the empty string with equates to a NULL value in the authentication system mentioned above.

CREATEDB/NOCREATEDB

These clauses define a user's ability to create databases. If CREATEDB is specified, the user being defined will be allowed to create his own databases. Using NOCREATEDB will deny a user the ability to create databases. If this clause is omitted, NOCREATEDB is used by default.

CREATEUSER/NOCREATEUSER

These clauses determine whether a user will be permitted to create new users in an instance of Postgres. Omitting this clause will set the user's value of this attribute to be NOCREATEUSER.

groupname

A name of a group into which to insert the user as a new member.

abstime

The VALID UNTIL clause sets an absolute time after which the user's Postgres login is no longer valid. Please note that if a user does not have a password defined in the "pg_shadow" table, the valid until date will not be checked during user authentication. If this clause is omitted, a NULL value is stored in "pg_shadow" for this attribute, and the login will be valid for all time.

Outputs

CREATE USER

Message returned if the command completes successfully.

ERROR: removeUser: user "*username*" does not exist

if "*username*" not found.

Description

CREATE USER will add a new user to an instance of Postgres.

The new user will be given a `usesysid` of: '`SELECT MAX(usesysid) + 1 FROM pg_shadow`'. This means that Postgres users' `usesysid`s will not correspond to their operating system(OS) user ids. The exception to this rule is the '`postgres`' user, whose OS user id is used as the `usesysid` during the initdb process. If you still want the OS user id and the `usesysid` to match for any given user, use the "createuser" script provided with the Postgres distribution.

Notes

CREATE USER statement is a Postgres language extension.

Use DROP USER or ALTER USER statements to remove or modify a user account.

Refer to the `pg_shadow` table for further information.

```
Table    = pg_shadow
+--------------------------+--------------------------+-------+
|          Field           |          Type            | Length|
+--------------------------+--------------------------+-------+
| usename                  | name                     |    32 |
| usesysid                 | int4                     |     4 |
| usecreatedb              | bool                     |     1 |
| usetrace                 | bool                     |     1 |
| usesuper                 | bool                     |     1 |
| usecatupd                | bool                     |     1 |
| passwd                   | text                     |   var |
| valuntil                 | abstime                  |     4 |
+--------------------------+--------------------------+-------+
```

Usage

Create a user with no password:

```
CREATE USER jonathan
```

Create a user with a password:

```
CREATE USER davide WITH PASSWORD jw8s0F4
```

Create a user with a password, whose account is valid until the end of 2001. Note that after one second has ticked in 2002, the account is not valid:

```
CREATE USER miriam WITH PASSWORD jw8s0F4 VALID UNTIL 'Jan 1 2002'
```

Create an account where the user can create databases:

```
CREATE USER manuel WITH PASSWORD jw8s0F4 CREATEDB
```

Compatibility

SQL92

There is no CREATE USER statement in SQL92.

Name

CREATE VIEW — Constructs a virtual table

Synopsis

<refsynopsisdivinfo>1998-09-21</refsynopsisdivinfo>

```
CREATE VIEW view
    AS SELECT query
```

Inputs

view

The name of a view to be created.

query

An SQL query which will provide the columns and rows of the view.

Refer to the SELECT statement for more information about valid arguments.

Outputs

CREATE

The message returned if the view is successfully created.

WARN amcreate: "*view*" relation already exists

This error occurs if the view specified already exists in the database.

NOTICE create: attribute named "*column*" has an unknown type

The view will be created having a column with an unknown type if you do not specify it. For example, the following command gives an error:

```
CREATE VIEW vista AS SELECT 'Hello World'
```

whereas this command does not:

```
CREATE VIEW vista AS SELECT 'Hello World'::text
```

Description

CREATE VIEW will define a view of a table. This view is not physically materialized. Specifically, a query rewrite retrieve rule is automatically generated to support retrieve operations on views.

Notes

Use the DROP VIEW statement to drop views.

Bugs

Currently, views are read only.

Usage

Create a view consisting of all Comedy films:

```
CREATE VIEW kinds AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';

SELECT * FROM kinds;

code |title                     |did| date_prod|kind      |len
-----+--------------------------+---+----------+----------+------
UA502|Bananas                   |105|1971-07-13|Comedy    | 01:22
C_701|There's a Girl in my Soup|107|1970-06-11|Comedy    | 01:36
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for the CREATE VIEW statement:

```
CREATE VIEW view [ column [, ...] ]
    AS SELECT expression [AS colname] [, ...]
    FROM table
    [ WHERE condition ]
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

CHECK OPTION

This option is to do with updatable views. All INSERTs and UPDATEs on the view will be checked to ensure data satisfy the view-defining condition. If they do not, the update will be rejected.

LOCAL

Check for integrity on this view.

CASCADE

Check for integrity on this view and on any dependent view. CASCADE is assumed if neither CASCADE nor LOCAL is specified.

Name

DECLARE — Defines a cursor for table access

Synopsis

<refsynopsisdivinfo>1998-09-04</refsynopsisdivinfo>

```
DECLARE cursor [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] ]
```

Inputs

cursor

The name of the cursor to be used in subsequent FETCH operations..

BINARY

Causes the cursor to fetch data in binary rather than in text format.

INSENSITIVE

SQL92 keyword indicating that data retrieved from the cursor should be unaffected by updates from other processes or cursors. Since cursor operations occur within transactions in Postgres this is always the case. This keyword has no effect.

SCROLL

SQL92 keyword indicating that data may be retrieved in multiple rows per FETCH operation. Since this is allowed at all times by Postgres this keyword has no effect.

query

An SQL query which will provide the rows to be governed by the cursor. Refer to the SELECT statement for further information about valid arguments.

READ ONLY

SQL92 keyword indicating that the cursor will be used in a readonly mode. Since this is the only cursor access mode available in Postgres this keyword has no effect.

UPDATE

SQL92 keyword indicating that the cursor will be used to update tables. Since cursor updates are not currently supported in Postgres this keyword provokes an informational error message.

column

Column(s) to be updated. Since cursor updates are not currently supported in Postgres the UPDATE clause provokes an informational error message.

Outputs

SELECT

The message returned if the SELECT is run successfully.

NOTICE BlankPortalAssignName: portal "*cursor*" already exists

This error occurs if cursor "*cursor*" is already declared.

ERROR: Named portals may only be used in begin/end transaction blocks

This error occurs if the cursor is not declared within a transaction block.

Description

DECLARE allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. Cursors can return data either in text or in binary foramt.

Normal cursors return data in text format, either ASCII or another encoding scheme depending on how the Postgres backend was built. Since data is stored natively in binary format, the system must do a conversion to produce the text format. In addition, text formats are often larger in size than the corresponding binary format. Once the information comes back in text form, the client application may have to convert it to a binary format to manipulate it anyway.

BINARY cursors give you back the data in the native binary representation. So binary cursors will tend to be a little faster since they suffer less conversion overhead.

As an example, if a query returns a value of one from an integer column, you would get a string of '1' with a default cursor whereas with a binary cursor you would get a 4-byte value equal to control-A ('^A').

Caution

BINARY cursors should be used carefully. User applications such as psql are not aware of binary cursors and expect data to come back in a text format.

However, string representation is architecture-neutral whereas binary representation can differ between different machine architectures. Therefore, if your client machine and server machine use different representations (e.g. "big-endian" versus "little-endian"), you will probably not want your data returned in binary format.

Tip

If you intend to display the data in ASCII, getting it back in ASCII will save you some effort on the client side.

Notes

Cursors are only available in transactions.

Postgres does not have an explicit `OPEN cursor` statement; a cursor is considered to be open when it is declared.

Note

In SQL92 cursors are only available in embedded applications. ecpg, the embedded SQL preprocessor for Postgres, supports the SQL92 conventions, including those involving DECLARE and OPEN statements.

Usage

To declare a cursor:

```
DECLARE liahona CURSOR
    FOR SELECT * FROM films;
```

Compatibility

SQL92

SQL92 allows cursors only in embedded SQL and in modules. Postgres permits cursors to be used interactively. SQL92 allows embedded or modular cursors to update database information. All Postgres cursors are readonly. The BINARY keyword is a Postgres extension.

Name

DELETE — Deletes rows from a table

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
DELETE FROM table [ WHERE condition ]
```

Inputs

table

The name of an existing table.

condition

This is an SQL selection query which returns the rows which are to be deleted.

Refer to the SELECT statement for further description of the WHERE clause.

Outputs

DELETE *count*

Message returned if items are successfully deleted. The *count* is the number of rows deleted.

If *count* is 0, no rows were deleted.

Description

DELETE removes rows which satisfy the WHERE *condition*, from the specified table.

If the *condition* is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

You must have write access to the table in order to modify it, as well as read access to any table whose values are read in the *condition*.

Usage

Remove all films but musicals:

```
DELETE FROM films WHERE kind <> 'Musical';

SELECT * FROM films;

code |title                    |did| date_prod|kind      |len
-----+-------------------------+---+----------+----------+------
UA501|West Side Story          |105|1961-01-03|Musical   | 02:32
TC901|The King and I           |109|1956-08-11|Musical   | 02:13
WD101|Bed Knobs and Broomsticks|111|          |Musical   | 01:57
(3 rows)
```

Clear the table `films`:

```
DELETE FROM films;

SELECT * FROM films;
code|title|did|date_prod|kind|len
----+-----+---+---------+----+---
(0 rows)
```

Compatibility

SQL92

SQL92 allows a positioned DELETE statement:

```
DELETE FROM table WHERE CURRENT OF cursor
```

where *`cursor`* identifies an open cursor. Interactive cursors in Postgres are read-only.

Name

DROP AGGREGATE — Removes the definition of an aggregate function

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
DROP AGGREGATE name type
```

Inputs

name

The name of an existing aggregate function.

type

The type of an existing aggregate function. (Refer to the *PostgreSQL User's Guide* for further information about data types).

Outputs

DROP

Message returned if the command is successful.

WARN RemoveAggregate: aggregate '*name*' for '*type*' does not exist

This message occurs if the aggregate function specified does not exist in the database.

Description

`DROP AGGREGATE` will remove all references to an existing aggregate definition. To execute this command the current user must be the owner of the aggregate.

Notes

The `DROP AGGREGATE` statement is a Postgres language extension.

Refer to the `CREATE AGGREGATE` statement to create aggregate functions.

Usage

To remove the `myavg` aggregate for type `int4`:

```
DROP AGGREGATE myavg int4;
```

Compatibility

SQL92

There is no DROP AGGREGATE statement in SQL92.

Name

DROP DATABASE — Destroys an existing database

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
DROP DATABASE name
```

Inputs

name

The name of an existing database to remove.

Outputs

DESTROYDB

This message is returned if the command is successful.

WARN: destroydb: database "*name*" does not exist.

This message occurs if the specified database does not exist.

ERROR: destroydb cannot be executed on an open database

This message occurs if the specified database does not exist.

Description

`DROP DATABASE` removes the catalog entries for an existing database and deletes the directory containing the data. It can only be executed by the database administrator (See the `CREATE DATABASE` command for details).

Notes

DROP DATABASE statement is a Postgres language extension.

Tip

This query cannot be executed while connected to the target database. It is usually preferable to use the `destroydb` script instead.

Refer to the `CREATE DATABASE` statement for information on how to create a database.

Compatibility

SQL92

There is no `DROP DATABASE` in SQL92.

Name

DROP FUNCTION — Removes a user-defined C function

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Inputs

name

The name of an existing function.

type

The type of function parameters.

Outputs

DROP

Message returned if the command completes successfully.

WARN RemoveFunction: Function "*name*" ("*types*") does not exist

This message is given if the function specified does not exist in the current database.

Description

DROP FUNCTION will remove references to an existing C function. To execute this command the user must be the owner of the function. The input argument types to the function must be specified, as only the function with the given name and argument types will be removed.

Notes

Refer to CREATE FUNCTION to create aggregate functions.

Usage

This command removes the square root function:

```
DROP FUNCTION sqrt(int4);
```

Bugs

No checks are made to ensure that types, operators or access methods that rely on the function have been removed first.

Compatibility

DROP FUNCTION is a Postgres language extension.

SQL/PSM

SQL/PSM is a proposed standard to enable function extensibility. The SQL/PSM DROP FUNCTION statement has the following syntax:

```
DROP [ SPECIFIC ] FUNCTION name { RESTRICT | CASCADE }
```

Name

DROP INDEX — Removes an index from a database

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
DROP INDEX index_name
```

Inputs

index_name

The name of the index to remove.

Outputs

DROP

The message returned if the index is successfully dropped.

ERROR: index "*index_name*" nonexistent

This message occurs if *index_name* is not an index in the database.

Description

`DROP INDEX` drops an existing index from the database system. To execute this command you must be the owner of the index.

Notes

`DROP INDEX` is a Postgres language extension.

Refer to the `CREATE INDEX` statement for information on how to create indexes.

Usage

This command will remove the `title_idx` index:

```
DROP INDEX title_idx;
```

Compatibility

SQL92

SQL92 defines commands by which to access a generic relational database. Indexes are an implementation-dependent feature and hence there are no index-specific commands or definitions in the SQL92 language.

Name

DROP LANGUAGE — Removes a user-defined procedural language

Synopsis

<refsynopsisdivinfo>1998-04-15</refsynopsisdivinfo>

```
DROP PROCEDURAL LANGUAGE 'langname'
```

Inputs

langname

The name of an existing language.

Outputs

DROP

This message is returned if the language is successfully dropped.

ERROR: Language "*langname*" doesn't exist

This message occurs if the language "*langname*" is not found.

Description

`DROP PROCEDURAL LANGUAGE` will remove the definition of the previously registered procedural language having the name '*langname*'.

Notes

The `DROP PROCEDURAL LANGUAGE` statement is a Postgres language extension.

Refer to `CREATE PROCEDURAL LANGUAGE` for information on how to create procedural languages.

Bugs

No checks are made if functions or trigger procedures registered in this language still exist. To re-enable them without having to drop and recreate all the functions, the pg_proc's prolang attribute of the functions must be adjusted to the new object ID of the recreated pg_language entry for the PL.

Usage

This command removes the PL/Sample language:

```
DROP PROCEDURAL LANGUAGE 'plsample'
```

Compatibility

SQL92

There is no `DROP PROCEDURAL LANGUAGE` in SQL92.

Name

DROP OPERATOR — Removes an operator from the database

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP OPERATOR id ( type | NONE [,...] )
```

Inputs

id

The identifier of an existing operator.

type

The type of function parameters.

Outputs

DROP

The message returned if the command is successful.

ERROR: RemoveOperator: binary operator '*id*' taking '*type1*' and '*type2*' does not exist

This message occurs if the specified binary operator does not exist.

ERROR: RemoveOperator: left unary operator '*id*' taking '*type*' does not exist

This message occurs if the specified left unary operator specified does not exist.

ERROR: RemoveOperator: right unary operator '*id*' taking '*type*' does not exist

This message occurs if the specified right unary operator specified does not exist.

Description

The DROP OPERATOR statement drops an existing operator from the database. To execute this command you must be the owner of the operator.

The left or right type of a left or right unary operator, respectively, may be specified as NONE.

Notes

The `DROP OPERATOR` statement is a Postgres language extension.

Refer to `CREATE OPERATOR` for information on how to create operators.

It is the user's responsibility to remove any access methods and operator classes that rely on the deleted operator.

Usage

Remove power operator `a^n` for `int4`:

```
DROP OPERATOR ^ (int4, int4);
```

Remove left unary operator `!a` for booleans:

```
DROP OPERATOR ! (none, bool);
```

Remove right unary factorial operator `a!` for `int4`:

```
DROP OPERATOR ! (int4, none);
```

Compatibility

SQL92

There is no `DROP OPERATOR` in SQL92.

Name

DROP RULE — Removes an existing rule from the database

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP RULE name
```

Inputs

name

The name of an existing rule to drop.

Outputs

DROP

Message returned if successfully.

ERROR: RewriteGetRuleEventRel: rule "*name*" not found

This message occurs if the specified rule does not exist.

Description

`DROP RULE` drops a rule from the specified Postgres rule system. Postgres will immediately cease enforcing it and will purge its definition from the system catalogs.

Notes

The `DROP RULE` statement is a Postgres language extension.

Refer to `CREATE RULE` for information on how to create rules.

Bugs

Once a rule is dropped, access to historical information the rule has written may disappear.

Usage

To drop the rewrite rule `newrule`:

```
DROP RULE newrule
```

Compatibility

SQL92

There is no `DROP RULE` in SQL92.

Name

DROP SEQUENCE — Removes an existing sequence

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP SEQUENCE seqname [, ...]
```

Inputs

seqname

The name of a sequence.

Outputs

DROP

The message returned if the sequence is successfully dropped.

WARN: Relation "*seqname*" does not exist.

This message occurs if the specified sequence does not exist.

Description

`DROP SEQUENCE` removes sequence number generators from the data base. With the current implementation of sequences as special tables it works just like the `DROP TABLE` statement.

Notes

The `DROP SEQUENCE` statement is a Postgres language extension.

Refer to the `CREATE SEQUENCE` statement for information on how to create a sequence.

Usage

To remove sequence `serial` from database:

```
DROP SEQUENCE serial
```

Compatibility

SQL92

There is no `DROP SEQUENCE` in SQL92.

Name

DROP TABLE — Removes existing tables from a database

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP TABLE table [, ...]
```

Inputs

table

The name of an existing table or view to drop.

Outputs

DROP

The message returned if the command completes successfully.

ERROR Relation "*table*" Does Not Exist!

If the specified table or view does not exist in the database.

Description

`DROP TABLE` removes tables and views from the database. Only its owner may destroy a table or view. A table may be emptied of rows, but not destroyed, by using `DELETE`.

If a table being destroyed has secondary indexes on it, they will be removed first. The removal of just a secondary index will not affect the contents of the underlying table.

Notes

Refer to `CREATE TABLE` and `ALTER TABLE` for information on how to create or modify tables.

Usage

To destroy the `films` and `distributors` tables:

```
DROP TABLE films, distributors
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for DROP TABLE:

```
DROP TABLE table { RESTRICT | CASCADE }
```

RESTRICT

Ensures that only a table with no dependent views or integrity constraints can be destroyed.

CASCADE

Any referencing views or integrity constraints will also be dropped.

Tip

At present, to remove a referenced view you must drop it explicitly.

Name

DROP TRIGGER — Removes the definition of a trigger

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP TRIGGER name ON table
```

Inputs

name

The name of an existing trigger.

table

The name of a table.

Outputs

DROP

The message returned if the trigger is successfully dropped.

ERROR: DropTrigger: there is no trigger *name* on relation "*table*"

This message occurs if the trigger specified does not exist.

Description

`DROP TRIGGER` will remove all references to an existing trigger definition. To execute this command the current user must be the owner of the trigger.

Notes

`DROP TRIGGER` is a Postgres language extension.

Refer to `CREATE TRIGGER` for information on how to create triggers.

Usage

Destroy the `if_dist_exists` trigger on table `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

Compatibility

SQL92

There is no `DROP TRIGGER` statement in SQL92.

Name

DROP TYPE — Removes a user-defined type from the system catalogs

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP TYPE typename
```

Inputs

typename

The name of an existing type.

Outputs

DROP

The message returned if the command is successful.

ERROR: RemoveType: type '*typename*' does not exist

This message occurs if the specified type is not found.

Description

`DROP TYPE` will remove a user type from the system catalogs.

Only the owner of a type can remove it.

Notes

DROP TYPE statement is a Postgres language extension.

Refer to `CREATE TYPE` for inforamation on how to create types.

It is the user's responsibility to remove any operators, functions, aggregates, access methods, subtypes, and classes that use a deleted type.

Bugs

If a built-in type is removed, the behavior of the backend is unpredictable.

Usage

To remove the `box` type:

```
DROP TYPE box
```

Compatibility

SQL3

DROP TYPE is a SQL3 statement.

Name

DROP USER — Removes an user account information

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP USER username
```

Inputs

username

The name of an existing user.

Outputs

DROP

The message returned if the user is successfully deleted.

ERROR: removeUser: user "*username*" does not exist.

This message occurs if the username is not found.

Description

`DROP USER` removes the specified user from the database, along with any databases owned by the user. It does not remove tables, views, or triggers owned by the named user in databases not owned by the user. This statement can be used in place of the destroyuser script, regardless of how the user was created.

Notes

`DROP USER` is a Postgres language extension.

Refer to `CREATE USER` and `ALTER USER` for information on how to create or modify user accounts.

Usage

To drop a user account:

```
DROP USER Jonathan;
```

Compatibility

SQL92

There is no `DROP USER` in SQL92.

Name

DROP VIEW — Removes an existing view from a database

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP VIEW view
```

Inputs

view

The name of an existing view.

Outputs

DROP

The message returned if the command is successful.

ERROR: RewriteGetRuleEventRel: rule "_RET*view*" not found

This message occurs if the specified view does not exist in the database.

Description

`DROP VIEW` drops an existing view from the database. To execute this command you must be the owner of the view.

Notes

The Postgres `DROP TABLE` statement also drops views.

Refer to `CREATE VIEW` for information on how to create views.

Usage

This command will remove the view called `kinds`:

```
DROP VIEW kinds;
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for `DROP VIEW`:

```
DROP VIEW view { RESTRICT | CASCADE }
```

Inputs

RESTRICT

Ensures that only a view with no dependent views or integrity constraints can be destroyed.

CASCADE

Any referencing views and integrity constraints will be dropped as well.

Notes

Tip

At present, to remove a referenced view from a Postgres database, you must drop it explicitly.

Name

EXPLAIN — Shows statement execution details

Synopsis

<refsynopsisdivinfo>1998-09-01</refsynopsisdivinfo>

```
EXPLAIN [ VERBOSE ] query
```

Inputs

VERBOSE

Flag to show detailed query plan.

query

Any *query*.

Outputs

NOTICE: QUERY PLAN: *plan*

Explicit query plan from the Postgres backend.

EXPLAIN

Flag sent after query plan is shown.

Description

This command outputs details about the supplied query. The default output is the computed query cost. VERBOSE displays the full query plan and cost to your screen, and pretty-prints the plan to the postmaster log file.

Notes

There is only sparse documentation on the optimizer's use of cost information in Postgres. General information on cost estimation for query optimization can be found in database textbooks. Refer to the *Programmer's Guide* in the chapters on indexes and the genetic query optimizer for more information.

Usage

To show a query plan for a simple query:

```
postgres=> explain select * from foo;
NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00 size=0 width=4)

EXPLAIN
```

Compatibility

SQL92

There is no EXPLAIN statement defined in SQL92.

Name

FETCH — Gets rows using a cursor

Synopsis

<refsynopsisdivinfo>1998-09-01</refsynopsisdivinfo>

```
FETCH [ selector ] [ count ]
    { IN | FROM } cursor
FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ]
    FROM ] cursor
```

Inputs

selector

selector defines the fetch direction. It can be one the following:

FORWARD

fetch next row(s). This is the default if *selector* is omitted.

BACKWARD

fetch previous row(s).

RELATIVE

Noise word for SQL92 compatibility.

count

count determines how many rows to fetch. It can be one of the following:

#

A signed integer that specify how many rows to fetch. Note that a negative integer is equivalent to changing the sense of FORWARD and BACKWARD.

ALL

Retrieve all remaining rows.

NEXT

Equivalent to specifying a count of `1`.

PRIOR

Equivalent to specifying a count of `-1`.

cursor

An open cursor's name.

Outputs

FETCH returns the results of the query defined by the specified cursor. The following messages will be returned if the query fails:

NOTICE: PerformPortalFetch: portal "*cursor*" not found

If *cursor* is not previously declared. The cursor must be declared within a transaction block.

NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE

Postgres does not support absolute positioning of cursors.

ERROR: FETCH/RELATIVE at current position is not supported

SQL92 allows one to repetatively retrieve the cursor at its "current position" using the syntax

```
FETCH RELATIVE 0 FROM cursor
```

Postgres does not currently support this notion; in fact the value zero is reserved to indicate that all rows should be retrieved and is equivalent to specifying the ALL keyword. If the RELATIVE keyword has been used, the Postgres assumes that the user intended SQL92 behavior and returns this error message.

Description

FETCH allows a user to retrieve rows using a cursor. The number of rows retrieved is specified by #. If the number of rows remaining in the cursor is less than #, then only those available are fetched. Substituting the keyword ALL in place of a number will cause all remaining rows in the cursor to be retrieved. Instances may be fetched in both FORWARD and BACKWARD directions. The default direction is FORWARD.

Tip

Negative numbers are now allowed to be specified for the row count. A negative number is equivalent to reversing the sense of the FORWARD and BACKWARD keywords. For example, `FORWARD -1` is the same as `BACKWARD 1`.

Note that the FORWARD and BACKWARD keywords are Postgres extensions. The SQL92 syntax is also supported, specified in the second form of the command. See below for details on compatibility issues.

Once all rows are fetched, every other fetch access returns no rows.

Updating data in a cursor is not supported by Postgres, because mapping cursor updates back to base tables is not generally possible, as is also the case with VIEW updates. Consequently, users must issue explicit UPDATE commands to replace data.

Cursors may only be used inside of transactions because the data that they store spans multiple user queries.

Notes

Refer to MOVE statements to change cursor position. Refer to DECLARE statements to declare a cursor. Refer to BEGIN WORK, COMMIT WORK, ROLLBACK WORK statements for further information about transactions.

Usage

```
--set up and use a cursor:
--
BEGIN WORK;
  DECLARE liahona CURSOR
```

```
     FOR SELECT * FROM films;

--Fetch first 5 rows in the cursor liahona:
--
  FETCH FORWARD 5 IN liahona;

  code |title                   |did| date_prod|kind      |len
  -----+------------------------+---+----------+----------+------
  BL101|The Third Man           |101|1949-12-23|Drama     | 01:44
  BL102|The African Queen       |101|1951-08-11|Romantic  | 01:43
  JL201|Une Femme est une Femme|102|1961-03-12|Romantic  | 01:25
  P_301|Vertigo                 |103|1958-11-14|Action    | 02:08
  P_302|Becket                  |103|1964-02-03|Drama     | 02:28


--Fetch previous row:
--
  FETCH BACKWARD 1 IN liahona;

  code |title                   |did| date_prod|kind      |len
  -----+------------------------+---+----------+----------+------
  P_301|Vertigo                 |103|1958-11-14|Action    | 02:08

-- close the cursor and commit work:
--
  CLOSE liahona;
COMMIT WORK;
```

Compatibility

The non-embedded use of cursors is a Postgres extension. The syntax and usage of cursors is being compared against the embedded form of cursors defined in SQL92.

SQL92

SQL92 allows absolute positioning of the cursor for FETCH, and allows placing the results into explicit variables.

```
FETCH ABSOLUTE #
    FROM cursor
    INTO :variable [, ...]
```

ABSOLUTE

The cursor should be positioned to the specified absolute row number. All row numbers in Postgres are relative numbers so this capability is not supported.

:*variable*

Target host variable(s).

Name

GRANT — Grants access privilege to a user, a group or all users

Synopsis

<refsynopsisdivinfo>1998-09-23</refsynopsisdivinfo>

```
GRANT privilege [, ...]
    ON object [, ...]
    TO { PUBLIC | GROUP group | username }
```

Inputs

privilege

The possible privileges are:

SELECT

Access all of the columns of a specific table/view.

INSERT

Insert data into all columns of a specific table.

UPDATE

Update all columns of a specific table.

DELETE

Delete rows from a specific table.

RULE

Define rules on the table/view (See CREATE RULE statement).

ALL

Grant all privileges.

object

The name of an object to which to grant access. The possible objects are:

- table
- view
- sequence
- index

PUBLIC

A short form representing all users.

GROUP *group*

A *group* to whom to grant privileges. In the current release, the group must be created explicitly as described below.

username

The name of a user to whom grant privileges. PUBLIC is a short form representing all users.

Outputs

CHANGE

Message returned if successful.

ERROR: ChangeAcl: class "*object*" not found

Message returned if the specified object is not available or if it is impossible to give privileges to the specified group or users.

Description

GRANT allows the creator of an object to give specific permissions to all users (PUBLIC) or to a certain user or group. Users other than the creator don't have any access permission unless the creator GRANTs permissions, after the object is created.

Once a user has a privilege on an object, he is enabled to exercise that privilege. There is no need to GRANT privileges to the creator of an object, the creator automatically holds ALL privileges, and can also drop the object.

Notes

Use the `psql \z` command for further information about permissions on existing objects:

```
Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------------+---------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a GROUP
           =arwR -- privileges granted to PUBLIC

               r -- SELECT
               w -- UPDATE/DELETE
               a -- INSERT
               R -- RULE
            arwR -- ALL
```

Tip

Currently, to create a GROUP you have to insert data manually into table pg_group as:

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

Refer to REVOKE statements to revoke access privileges.

Usage

```
-- grant insert privilege to all users on table films:
--
GRANT INSERT ON films TO PUBLIC;

-- grant all privileges to user manuel on view kinds:
--
GRANT ALL ON kinds TO manuel;
```

Compatibility

SQL92

The SQL92 syntax for GRANT allows setting privileges for individual columns within a table, and allows setting a privilege to grant the same privileges to others.

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] }
    [ WITH GRANT OPTION ]
```

Fields are compatible with the those in the Postgres implementation, with the following additions:

privilege SELECT

SQL92 permits additional privileges to be specified:

REFERENCES

Allowed to reference some or all of the columns of a specific table/view in integrity constraints.

USAGE

Allowed to use a domain, character set, collation or translation. If an object specifies anything other than a table/view, *privilege* must specify only USAGE.

Tip

Currently, to grant privileges in Postgres to only few columns, you must create a view having desired columns and then grant privileges to that view.

object

object

SQL92 allows an additional non-functional keyword:

[TABLE] table

CHARACTER SET

Allowed to use the specified character set.

COLLATION

Allowed to use the specified collation sequence.

TRANSLATION

Allowed to use the specified character set translation.

DOMAIN

Allowed to use the specified domain.

WITH GRANT OPTION

Allowed to grant the same privilege to others.

Name

INSERT — Inserts new rows into a table

Synopsis

<refsynopsisdivinfo>1998-09-23</refsynopsisdivinfo>

```
INSERT INTO table [ ( column [, ...] ) ]
    { VALUES ( expression [, ...] ) | SELECT query }
```

Inputs

table

The name of an existing table.

column

The name of a column in *table*.

expression

A valid expression or value to assign to *column*.

query

A valid query. Refer to the SELECT statement for a further description of valid arguments.

Outputs

INSERT *oid* 1

Message returned if only one row was inserted. *oid* is the row identifier.

INSERT 0 *#*

Message returned if more than one rows were inserted. *#* is the number of rows inserted.

Description

INSERT allows one to insert new rows into a table. One can insert a single row at time or several rows as a result of a query. The columns in the target list may be listed in any order. In every column not present in the target list will be inserted the default value, if column has not a declared default value it will be assumed as NULL. If the expression for each column is not of the correct data type, automatic type coercion will be attempted.

You must have insert privilege to a table in order to append to it, as well as select privilege on any table specified in a WHERE clause.

Usage

```
--Insert a single row into table films;
--(in the second example the column date_prod is omitted
--therefore will be stored in it a default value of NULL):
```

```
--
INSERT INTO films VALUES
    ('UA502','Bananas',105,'1971-07-13','Comedy',INTERVAL '82
 minute');

INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');

--Insert a single row into table distributors, note that
--only column "name" is specified, to the non specified
--column "did" will be assigned its default value:
--
INSERT INTO distributors (name) VALUES ('British Lion');

--Insert several rows into table films from table tmp:
--
INSERT INTO films
    SELECT * FROM tmp;

--Insert into arrays:
--Create an empty 3x3 gameboard for noughts-and-crosses
--(all of these queries create the same board attribute)
--(Refer to the PostgreSQL User's Guide for further
--information about arrays).

INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2,'{}');
INSERT INTO tictactoe (game, board)
    VALUES (3,'{{,,},{,,},{,,}}');
```

Compatibility

SQL92

The INSERT statement is fully compatible with SQL92. Possible limitations in features of the *query* clause are documented for the SELECT statement.

Name

LISTEN — Listen for notification on a notify condition

Synopsis

<refsynopsisdivinfo>1998-10-07</refsynopsisdivinfo>

```
LISTEN notifyname
```

Inputs

notifyname

Name of notify condition.

Outputs

LISTEN

Message returned upon successful completion of registration.

NOTICE Async_Listen: We are already listening on notifyname

If this backend is already registered for that notify condition.

Description

LISTEN registers the current Postgres backend as a listener on the notify condition *notifyname*.

Whenever the command NOTIFY *notifyname* is invoked, either by this backend or another one connected to the same database, all the backends currently listening on that notify condition are notified, and each will in turn notify its connected frontend application. See the discussion of NOTIFY for more information.

A backend can be deregistered for a given notify condition with the UNLISTEN command. Also, a backend's listen registrations are automatically cleared when the backend process exits.

The method a frontend application must use to detect notify events depends on which Postgres application programming interface it uses. With the basic libpq library, the application issues LISTEN as an ordinary SQL command, and then must periodically call the routine PQnotifies to find out whether any notify events have been received. Other interfaces such as libpgtcl provide higher-level methods for handling notify events; indeed, with libpgtcl the application programmer should not even issue LISTEN or UNLISTEN directly. See the documentation for the library you are using for more details.

The reference page for NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Notes

notifyname can be any string valid as a name; it need not correspond to the name of any actual table. If *notifyname* is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 31 characters long.

In some previous releases of Postgres, *notifyname* had to be enclosed in double-quotes when it did not correspond to any existing table name, even if syntactically valid as a name. That is no longer required.

Usage

```
-- Configure and execute a listen/notify sequence from psql
postgres=> listen virtual;
LISTEN
postgres=> notify virtual;
NOTIFY
ASYNC NOTIFY of 'virtual' from backend pid '11239' received
```

Compatibility

SQL92

There is no `LISTEN` in SQL92.

Name

LOAD — Dynamically loads an object file

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
LOAD 'filename'
```

Inputs

filename

Object file for dynamic loading.

Outputs

LOAD

Message returned on successful completion.

ERROR: LOAD: could not open file '*filename*'

Message returned if the specified file is not found. The file must be visible *to the Postgres backend*, with the appropriate full path name specified, to avoid this message.

Description

Loads an object (or ".o") file into the Postgres backend address space. Once a file is loaded, all functions in that file can be accessed. This function is used in support of user-defined types and functions.

If a file is not loaded using `LOAD`, the file will be loaded automatically the first time the function is called by Postgres. `LOAD` can also be used to reload an object file if it has been edited and recompiled. Only objects created from C language files are supported at this time.

Notes

Functions in loaded object files should not call functions in other object files loaded through the `LOAD` command. For example, all functions in file `A` should call each other, functions in the standard or math libraries, or in Postgres itself. They should not call functions defined in a different loaded file `B`. This is because if `B` is reloaded, the Postgres loader is not able to relocate the calls from the functions in `A` into the new address space of `B`. If `B` is not reloaded, however, there will not be a problem.

Object files must be compiled to contain position independent code. For example, on DECstations you must use /bin/cc with the `-G 0` option when compiling object files to be loaded.

Note that if you are porting Postgres to a new platform, `LOAD` will have to work in order to support ADTs.

Usage

```
--Load the file /usr/postgres/demo/circle.o
--
```

```
LOAD '/usr/postgres/demo/circle.o'
```

Compatibility

SQL92

There is no `LOAD` in SQL92.

Name

LOCK — Explicit lock of a table inside a transaction

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
LOCK [ TABLE ] table
```

Inputs

table

The name of an existing table to lock.

Outputs

DELETE 0

Message returned on a successful lock. LOCK is implemented as a DELETE FROM *table* which is guaranteed to not delete any rows.

ERROR *table*: Table does not exist.

Message returned if *table* does not exist.

Description

LOCK locks in exclusive mode a table inside a transaction. The classic use for this is the case where you want to select some data, then update it inside a transaction. If you don't explicit lock a table using LOCK statement, it will be implicit locked only at the first UPDATE, INSERT, or DELETE operation. If you don't exclusive lock the table before the select, some other user may also read the selected data, and try and do their own update, causing a deadlock while you both wait for the other to release the select-induced shared lock so you can get an exclusive lock to do the update.

Another example of deadlock is where one user locks one table, and another user locks a second table. While both keep their existing locks, the first user tries to lock the second user's table, and the second user tries to lock the first user's table. Both users deadlock waiting for the tables to become available. The only solution to this is for both users to lock tables in the same order, so user's lock acquisitions and requests to not form a deadlock.

Note

Postgres does detect deadlocks and will rollback transactions to resolve the deadlock. Usually, at least one of the deadlocked transactions will complete successfully.

Notes

LOCK is a Postgres language extension.

LOCK works only inside transactions.

Bug

If the locked table is dropped then it will be automatically unlocked even if a transaction is still in progress.

Usage

```
--Explicit locking to prevent deadlock:
--
BEGIN WORK;
    LOCK films;
    SELECT * FROM films;
    UPDATE films SET len = INTERVAL '100 minute'
        WHERE len = INTERVAL '117 minute';
COMMIT WORK;
```

Compatibility

SQL92

There is no `LOCK TABLE` in SQL92, which instead uses `SET TRANSACTION` to specify concurrency level on transactions.

Name

MOVE — Moves cursor position

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
MOVE [ selector ] [ count ]
    { IN | FROM } cursor
FETCH [ RELATIVE ] [ { [ # | ALL | NEXT | PRIOR ] } ] FROM ] cursor
```

Description

`MOVE` allows a user to move cursor position a specified number of rows. `MOVE` works like the `FETCH` command, but only positions the cursor and does not return rows.

Refer to the `FETCH` command for details on syntax and usage.

Notes

`MOVE` is a Postgres language extension.

Refer to `FETCH` for a description of valid arguments. Refer to `DECLARE` to declare a cursor. Refer to `BEGIN WORK`, `COMMIT WORK`, `ROLLBACK WORK` statements for further information about transactions.

Usage

```
--set up and use a cursor:
--
BEGIN WORK;
    DECLARE liahona CURSOR  FOR SELECT * FROM films;

    --Skip first 5 rows:
    --
    MOVE FORWARD 5 IN liahona;
MOVE
    --Fetch 6th row in the cursor liahona:
    --
    FETCH 1 IN liahona;
FETCH
code |title |did| date_prod|kind      |len
-----+------+---+----------+----------+------
P_303|48 Hrs|103|1982-10-22|Action    | 01:37
(1 row)
    -- close the cursor liahona and commit work:
    --
    CLOSE liahona;
COMMIT WORK;
```

Compatibility

SQL92

There is no SQL92 `MOVE` statement. Instead, SQL92 allows one to `FETCH` rows from an absolute cursor position, implicitly moving the cursor to the correct place.

Name

NOTIFY — Signals all frontends and backends listening on a notify condition

Synopsis

<refsynopsisdivinfo>1998-10-07</refsynopsisdivinfo>

```
NOTIFY notifyname
```

Inputs

notifyname

Notify condition to be signaled.

Outputs

NOTIFY

Acknowledgement that notify command has executed.

Notify events

Events are delivered to listening frontends; whether and how each frontend application reacts depends on its programming.

Description

The NOTIFY command sends a notify event to each frontend application that has previously executed LISTEN *notifyname* for the specified notify condition in the current database.

The information passed to the frontend for a notify event includes the notify condition name and the notifying backend process's PID. It is up to the database designer to define the condition names that will be used in a given database and what each one means.

Commonly, the notify condition name is the same as the name of some table in the database, and the notify event essentially means "I changed this table, take a look at it to see what's new". But no such association is enforced by the NOTIFY and LISTEN commands. For example, a database designer could use several different condition names to signal different sorts of changes to a single table.

NOTIFY provides a simple form of signal or IPC (interprocess communication) mechanism for a collection of processes accessing the same Postgres database. Higher-level mechanisms can be built by using tables in the database to pass additional data (beyond a mere condition name) from notifier to listener(s).

When NOTIFY is used to signal the occurrence of changes to a particular table, a useful programming technique is to put the NOTIFY in a rule that is triggered by table updates. In this way, notification happens automatically when the table is changed, and the application programmer can't accidentally forget to do it.

NOTIFY interacts with SQL transactions in some important ways. Firstly, if a NOTIFY is executed inside a transaction, the notify events are not delivered until and unless the transaction is committed. This is appropriate, since if the transaction is aborted we would like all the commands within it to have had no effect --- including NOTIFY. But it can be disconcerting if one is expecting the notify events to be delivered immediately. Secondly, if a listening backend receives a notify signal while it is within a transaction, the notify event will not be delivered to its connected frontend until just after the transaction is completed

(either committed or aborted). Again, the reasoning is that if a notify were delivered within a transaction that was later aborted, one would want the notification to be undone somehow --- but the backend cannot "take back" a notify once it has sent it to the frontend. So notify events are only delivered between transactions. The upshot of this is that applications using NOTIFY for real-time signaling should try to keep their transactions short.

NOTIFY behaves like Unix signals in one important respect: if the same condition name is signaled multiple times in quick succession, recipients may get only one notify event for several executions of NOTIFY. So it is a bad idea to depend on the number of notifies received. Instead, use NOTIFY to wake up applications that need to pay attention to something, and use a database object (such as a sequence) to keep track of what happened or how many times it happened.

It is common for a frontend that sends NOTIFY to be listening on the same notify name itself. In that case it will get back a notify event, just like all the other listening frontends. Depending on the application logic, this could result in useless work --- for example, re-reading a database table to find the same updates that that frontend just wrote out. In Postgres 6.4 and later, it is possible to avoid such extra work by noticing whether the notifying backend process's PID (supplied in the notify event message) is the same as one's own backend's PID (available from libpq). When they are the same, the notify event is one's own work bouncing back, and can be ignored. (Despite what was said in the preceding paragraph, this is a safe technique. Postgres keeps self-notifies separate from notifies arriving from other backends, so you cannot miss an outside notify by ignoring your own notifies.)

Notes

notifyname can be any string valid as a name; it need not correspond to the name of any actual table. If *notifyname* is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 31 characters long.

In some previous releases of Postgres, *notifyname* had to be enclosed in double-quotes when it did not correspond to any existing table name, even if syntactically valid as a name. That is no longer required.

In Postgres releases prior to 6.4, the backend PID delivered in a notify message was always the PID of the frontend's own backend. So it was not possible to distinguish one's own notifies from other clients' notifies in those earlier releases.

Usage

```
-- Configure and execute a listen/notify sequence from psql
postgres=> listen virtual;
LISTEN
postgres=> notify virtual;
NOTIFY
ASYNC NOTIFY of 'virtual' from backend pid '11239' received
```

Compatibility

SQL92

There is no NOTIFY statement in SQL92.

Name

RESET — Restores run-time parameters for session to default values

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
RESET variable
```

Inputs

variable

Refer to the SET statement for more information on available variables.

Outputs

RESET VARIABLE

Message returned if *variable* is successfully reset to its default value..

Description

`RESET` restores variables to the default values. Refer to the `SET` command for details on allowed values and defaults. `RESET` is an alternate form for

```
SET variable = DEFAULT
```

Notes

The RESET statement is a Postgres language extension.

Refer to SET/SHOW statements to set/show variable values.

Usage

```
-- reset DateStyle to its default;
RESET DateStyle;


-- reset Geqo to its default;
RESET GEQO;
```

Compatibility

SQL92

There is no `RESET` in SQL92.

Name

REVOKE — Revokes access privilege from a user, a group or all users.

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
REVOKE privilege [, ...]
    ON object [, ...]
    FROM { PUBLIC | GROUP group | username }
```

Inputs

privilege

The possible privileges are:

SELECT

Privilege to access all of the columns of a specific table/view.

INSERT

Privilege to insert data into all columns of a specific table.

UPDATE

Privilege to update all columns of a specific table.

DELETE

Privilege to delete rows from a specific table.

RULE

Privilege to define rules on table/view. (See `CREATE RULE`).

ALL

Rescind all privileges.

object

The name of an object from which to revoke access. The possible objects are:
- table
- view
- sequence
- index

group

The name of a group from whom to revoke privileges.

username

The name of a user from whom revoke privileges. Use the PUBLIC keyword to specify all users.

PUBLIC

Rescind the specified privilege(s) for all users.

Outputs

CHANGE

Message returned if successfully.

ERROR

Message returned if object is not available or impossible to revoke privileges from a group or users.

Description

REVOKE allows creator of an object to revoke permissions granted before, from all users (via PUBLIC) or a certain user or group.

Notes

Refer to psql \z command for further information about permissions on existing objects:

```
Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------------+---------------------------------------------+
Legend:
      uname=arwR -- privileges granted to a user
group gname=arwR -- privileges granted to a GROUP
           =arwR -- privileges granted to PUBLIC

               r -- SELECT
               w -- UPDATE/DELETE
               a -- INSERT
               R -- RULE
            arwR -- ALL
```

Tip

Currently, to create a GROUP you have to insert data manually into table pg_group as:

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

Usage

```
-- revoke insert privilege from all users on table films:
--
REVOKE INSERT ON films FROM PUBLIC;
```

```
-- revoke all privileges from user manuel on view kinds:
--
REVOKE ALL ON kinds FROM manuel;
```

Compatibility

SQL92

The SQL92 syntax for `REVOKE` has additional capabilities for rescinding privileges, including those on individual columns in tables:

```
REVOKE { SELECT | DELETE | USAGE | ALL PRIVILEGES } [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
REVOKE { INSERT | UPDATE | REFERENCES } [, ...] [ ( column
 [, ...] ) ]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

Refer to the `GRANT` command for details on individual fields.

```
REVOKE GRANT OPTION FOR privilege [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT | CASCADE }
```

Rescinds authority for a user to grant the specified privilege to others. Refer to the `GRANT` command for details on individual fields.

The possible objects are:

[TABLE] table/view
CHARACTER SET character-set
COLLATION collation
TRANSLATION translation
DOMAIN domain

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3 then user1 can revoke this privilege in cascade using the CASCADE keyword.

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3 then if user1 try revoke this privilege it fails if he/she specify the RESTRICT keyword.

Name

ROLLBACK — Aborts the current transaction

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
ROLLBACK [ WORK ]
```

Inputs

None.

Outputs

ABORT

Message returned if successful.

NOTICE: UserAbortTransactionBlock and not in in-progress state ABORT

If there is not any transaction currently in progress.

Description

`ROLLBACK` rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Notes

The keyword WORK is noise and can be omitted.

Use the `COMMIT` statement to successfully terminate a transaction.

Usage

```
--To abort all changes:
--
ROLLBACK WORK;
```

Compatibility

SQL92

Full compatibility.

Name

SELECT — Retrieve rows from a table or view.

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
SELECT [ALL|DISTINCT [ON column] ]
    expression [ AS name ] [, ...]
    [ INTO [TABLE] new_table ]
    [ FROM table [alias ] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ UNION [ALL] select ]
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

Inputs

expression

The name of a table's column or an expression.

name

Specifies another name for a column or an expression using the AS clause. *name* cannot be used in the WHERE condition. It can, however, be referenced in associated ORDER BY or GROUP BY clauses.

new_table

If the INTO TABLE clause is specified, the result of the query will be stored in another table with the indicated name. If *new_table* does not exist, it will be created automatically. Refer to SELECT INTO for more information.

Note

The CREATE TABLE AS statement will also create a new table from a select query.

table

The name of an existing table referenced by the FROM clause.

alias

An alternate name for the preceding *table*. It is used for brevity or to eliminate ambiguity for joins within a single table.

condition

A boolean expression giving a result of true or false. See the WHERE clause.

column

The name of a table's column.

select

A select statement with all features except the ORDER BY clause.

Outputs

Rows

The complete set of rows resulting from the query specification.

count

The count of rows returned by the query.

Description

SELECT will get all rows which satisfy the WHERE condition or all rows of a table if WHERE is omitted.

The GROUP BY clause allows a user to divide a table conceptually into groups. (See GROUP BY clause).

The HAVING clause specifies a grouped table derived by the elimination of groups from the result of the previously specified clause. (See HAVING clause).

The ORDER BY clause allows a user to specify that he/she wishes the rows sorted according to the ASCending or DESCending mode operator. (See ORDER BY clause)

The UNION clause specifies a table derived from a Cartesian product union join. (See UNION clause).

You must have SELECT privilege to a table to read its values (See GRANT/REVOKE statements).

WHERE Clause

The optional WHERE condition has the general form:

```
WHERE expr cond_op expr [ log_op ... ]
```

where *cond_op* can be one of: =, <, <=, >, >=, <> or a conditional operator like ALL, ANY, IN, LIKE, et cetera and *log_op* can be one of: AND, OR, NOT. The comparison returns either TRUE or FALSE and all instances will be discarded if the expression evaluates to FALSE.

GROUP BY Clause

GROUP BY specifies a grouped table derived by the application of the this clause:

```
GROUP BY column [, ...]
```

HAVING Clause

The optional HAVING condition has the general form:

```
HAVING cond_expr
```

where *cond_expr* is the same as specified for the WHERE clause.

HAVING specifies a grouped table derived by the elimination of groups from the result of the previously specified clause that do not meet the *cond_expr*.

Each column referenced in *cond_expr* shall unambiguously reference a grouping column.

ORDER BY Clause

```
ORDER BY column [ ASC | DESC ] [, ...]
```

column can be either a column name or an ordinal number.

The ordinal numbers refers to the ordinal (left-to-right) position of the column. This feature makes it possible to define an ordering on the basis of a column that does not have a proper name. This is never absolutely necessary because it is always possible assign a name to a calculated column using the AS clause, e.g.:

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

The columns in the ORDER BY must appear in the SELECT clause. Thus the following statement is illegal:

```
SELECT name FROM distributors ORDER BY code;
```

Optionally one may add the keyword DESC (descending) or ASC (ascending) after each column name in the ORDER BY clause. If not specified, ASC is assumed by default.

UNION Clause

```
table_query UNION [ ALL ] table_query
     [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

where *table_query* specifies any select expression without an ORDER BY clause.

The UNION operator specifies a table derived from a Cartesian product. The two tables that represent the direct operands of the UNION must have the same number of columns, and corresponding columns must be of compatible data types.

By default, the result of UNION does not contain any duplicate rows unless the ALL clause is specified.

Multiple UNION operators in the same SELECT statement are evaluated left to right. Note that the ALL keyword is not global in nature, being applied only for the current pair of table results.

Usage

To join the table `films` with the table `distributors`:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

    title                    |did|name            | date_prod|kind
    -------------------------+---+----------------+----------
+----------
    The Third Man            |101|British Lion    |1949-12-23|Drama
```

```
    The African Queen         |101|British Lion    |1951-08-11|Romantic
    Une Femme est une Femme   |102|Jean Luc Godard |1961-03-12|Romantic
    Vertigo                   |103|Paramount       |1958-11-14|Action
    Becket                    |103|Paramount       |1964-02-03|Drama
    48 Hrs                    |103|Paramount       |1982-10-22|Action
    War and Peace             |104|Mosfilm         |1967-02-12|Drama
    West Side Story           |105|United Artists  |1961-01-03|Musical
    Bananas                   |105|United Artists  |1971-07-13|Comedy
    Yojimbo                   |106|Toho            |1961-06-16|Drama
    There's a Girl in my Soup|107|Columbia        |1970-06-11|Comedy
    Taxi Driver               |107|Columbia        |1975-05-15|Action
    Absence of Malice         |107|Columbia        |1981-11-15|Action
    Storia di una donna       |108|Westward        |1970-08-15|Romantic
    The King and I            |109|20th Century Fox|1956-08-11|Musical
    Das Boot                  |110|Bavaria Atelier |1981-11-11|Drama
    Bed Knobs and Broomsticks|111|Walt Disney     |          |Musical
```

To sum the column `len` of all films and group the reults by `kind`:

```
SELECT kind, SUM(len) AS total FROM films GROUP BY kind;
```

```
    kind      |total
    ----------+------
    Action    | 07:34
    Comedy    | 02:58
    Drama     | 14:28
    Musical   | 06:42
    Romantic  | 04:38
```

To sum the column `len` of all films, group the reults by `kind` and show those group totals that are less than 5 hours:

```
SELECT kind, SUM(len) AS total
    FROM films
    GROUP BY kind
    HAVING SUM(len) < INTERVAL '5 hour';
```

```
    kind      |total
    ----------+------
    Comedy    | 02:58
    Romantic  | 04:38
```

The following two examples are identical ways of sorting the individual results according to the contents of the second column (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

```
    did|name
    ---+---------------
    109|20th Century Fox
    110|Bavaria Atelier
```

```
    101|British Lion
    107|Columbia
    102|Jean Luc Godard
    113|Luso films
    104|Mosfilm
    103|Paramount
    106|Toho
    105|United Artists
    111|Walt Disney
    112|Warner Bros.
    108|Westward
```

This example shows how to obtain the union of the tables `distributors` and `actors`, restricting the results to those that begin with letter W in each table. Only distinct rows are to be used, so the ALL keyword is omitted:

```
    --        distributors:               actors:
    --        did|name                    id|name
    --        ---+------------            --+--------------
    --        108|Westward                 1|Woody Allen
    --        111|Walt Disney              2|Warren Beatty
    --        112|Warner Bros.             3|Walter Matthau
    --        ...                         ...

SELECT distributors.name
    FROM   distributors
    WHERE  distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM   actors
    WHERE  actors.name LIKE 'W%'

    name
    --------------
    Walt Disney
    Walter Matthau
    Warner Bros.
    Warren Beatty
    Westward
    Woody Allen
```

Compatibility

SQL92

SELECT Clause

In the SQL92 standard, the optional keyword "AS" is just noise and can be omitted without affecting the meaning. The Postgres parser requires this keyword when renaming columns because the type extensibility features lead to parsing ambiguities in this context.

In the SQL92 standard, the new column name specified in an "AS" clause may be referenced in GROUP BY and HAVING clauses. This is not currently allowed in Postgres.

UNION Clause

The SQL92 syntax for UNION allows an additional CORRESPONDING BY clause:

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
    table_query
```

The CORRESPONDING BY clause is not supported by Postgres.

Name

SELECT INTO — Create a new table from an existing table or view

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
SELECT [ ALL | DISTINCT ] expression [ AS name ] [, ...]
    INTO [ TABLE ] new_table ]
    [ FROM table [alias] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ UNION [ ALL ] select]
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

Inputs

All input fields are described in detail for SELECT.

Outputs

All output fields are described in detail for SELECT.

Description

SELECT INTO creates a new table from the results of a query. Typically, this query draws data from an existing table, but any SQL query is allowed.

Note

CREATE TABLE AS is functionally equivalent to the SELECT INTO command.

Name

SET — Set run-time parameters for session

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
SET variable { TO | = } { 'value' | DEFAULT }
SET TIME ZONE { 'timezone' | LOCAL };
```

Inputs

variable

Settable global parameter.

value

New value of parameter.

The possible variables and allowed values are:

DateStyle

ISO

use ISO 8601-style dates and times

SQL

use Oracle/Ingres-style dates and times

Postgres

use traditional Postgres format

European

use dd/mm/yyyy for numeric date representations.

NonEuropean

use mm/dd/yyyy for numeric date representations.

German

use dd.mm.yyyy for numeric date representations.

US

same as 'NonEuropean'

default

restores the default values ('US,Postgres')

Date format initialization my be done by:

Setting PGDATESTYLE environment variable.
Running postmaster using -oe parameter to set dates to the 'European' convention. Note that this affects only the some combinations of date styles; for example the ISO style is not affected by this parameter.
Changing variables in `src/backend/utils/init/globals.c`.

The variables in `globals.c` which can be changed are:

```
bool EuroDates = false
                 true
int  DateStyle = USE_ISO_DATES
                 USE_POSTGRES_DATES
                 USE_ISO_DATES
                 USE_SQL_DATES
                 USE_GERMAN_DATES
```

TIMEZONE

The possible values for timezone depends on your operating system. For example on Linux /usr/lib/zoneinfo contains the database of timezones.

Here are some valid values for timezone:

'PST8PDT'

set the timezone for California

'Portugal'

set time zone for Portugal.

'Europe/Rome'

set time zone for Italy.

DEFAULT

set time zone to your local timezone (value of the TZ environment variable).

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

A frontend which uses libpq may be initialized by setting the PGTZ environment variable.

The second syntax shown above, allows one to set the timezone with a syntax similar to SQL92 `SET TIME ZONE`. The LOCAL keyword is just an alternate form of DEFAULT for SQL92 compatibility.

There are also several internal or optimization parameters which can be specified by the `SET` command:

COST_HEAP

Sets the default cost of a heap scan for use by the optimizer.

`float4`

Set the cost of a heap scan to the specified floating point value.

DEFAULT

Sets the cost of a heap scan to the default value.

The frontend may be initialized by setting the PGCOSTHEAP environment variable.

COST_INDEX

Sets the default cost of an index scan for use by the optimizer.

float4

Set the cost of an index scan to the specified floating point value.

DEFAULT

Sets the cost of an index scan to the default value.

The frontend may be initialized by setting the PGCOSTINDEX environment variable.

GEQO

Sets the threshold for using the genetic optimizer algorithm.

On

enables the genetic optimizer algorithm for statements with 8 or more tables.

On=#

Takes an integer argument to enable the genetic optimizer algorithm for statements with # or more tables in the query.

Off

disables the genetic optimizer algorithm.

DEFAULT

Equivalent to specifying `SET GEQO='on'`

This algorithm is on by default, which used GEQO for statements of eight or more tables. (See the chapter on GEQO in the Programmer's Guide for more information).

The frontend may be initialized by setting PGGEQO environment variable.

R_PLANS

Determines whether right-hand plan evaluation is allowed:

On

enables right-hand evaluation of plans.

Off

disables right-hand evaluation of plans.

DEFAULT

Equivalent to specifying `SET R_PLANS='off'`.

It may be useful when joining big relations with small ones. This algorithm is off by default. It's not used by GEQO anyway.

The frontend may be initialized by setting the PGRPLANS environment variable.

KSQO

Key Set Query Optimizer forces the query optimizer to optimize repetative OR clauses such as generated by MicroSoft Access:

On

enables this optimization.

Off

disables this optimization.

DEFAULT

Equivalent to specifying `SET KSQO='off'`.

It may be useful when joining big relations with small ones. This algorithm is off by default. It's not used by GEQO anyway.

The frontend may be initialized by setting the PGRPLANS environment variable.

QUERY_LIMIT

Sets the number of rows returned by a query.

Value

Maximum number of rows to return for a query. The default is to allow an unlimited number of rows.

#

Sets the maximum number of rows returned by a query to *#*.

DEFAULT

Sets the maximum number of rows returned by a query to be unlimited.

By default, there is no limit to the number of rows returned by a query.

Outputs

SET VARIABLE

Message returned if successfully.

WARN: Bad value for *`variable`* (*`value`*)

If the command fails to set variable.

Description

`SET` will modify configuration parameters for variable during a session.

Current values can be obtained using `SHOW`, and values can be restored to the defaults using `RESET`. Parameters and values are case-insensitive. Note that the value field is always specified as a string, so is enclosed in single-quotes.

`SET TIME ZONE` changes the session's default time zone offset. A SQL-session always begins with an initial default time zone offset. The `SET TIME ZONE` statement is used to change the default time zone offset for the current SQL session.

Notes

The `SET` *`variable`* statement is a Postgres language extension.

Refer to `SHOW` and `RESET` to display or reset the current values.

Usage

```
--Set the style of date to ISO:
--
SET DATESTYLE TO 'ISO';


--Set GEQO to default:
--
SET GEQO = DEFAULT;


--Turn on right-hand evaluation of plans:
--
SET R_PLANS TO 'on';


--set the timezone for Berkeley, California:
SET TIME ZONE 'PST8PDT';

SELECT CURRENT_TIMESTAMP AS today;

   today
   ----------------------
   1998-03-31 07:41:21-08


--set the timezone for Italy:
SET TIME ZONE 'Europe/Rome';

SELECT CURRENT_TIMESTAMP AS today;

   today
   ----------------------
   1998-03-31 17:41:31+02
```

Compatibility

SQL92

There is no `SET variable` in SQL92. The SQL92 syntax for `SET TIME ZONE` is slightly different, allowing only a single integer value for time zone specification:

```
SET TIME ZONE { interval_value_expression | LOCAL }
```

Name

SHOW — Shows run-time parameters for session

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
SHOW variable
```

Inputs

variable

Refer to SET for more information on available variables.

Outputs

NOTICE: *variable* is *value* SHOW VARIABLE

Message returned if successfully.

NOTICE: Unrecognized variable *value*

Message returned if value does not exist.

NOTICE: Time zone is unknown SHOW VARIABLE

If the TZ environment variable is not set.

Description

SHOW will display the current configuration parameters for variable during a session.

The session can be configured using SET statement, and values can be restored to the defaults using RESET statement. Parameters and values are case-insensitive.

Notes

The SHOW is a Postgres language extension.

Refer to SET/RESET to set/reset variable values. See also SET TIME ZONE.

Usage

```
-- show DateStyle;
SHOW DateStyle;
NOTICE:DateStyle is Postgres with US (NonEuropean) conventions

-- show Geqo;
SHOW GEQO;
NOTICE:GEQO is ON
```

Compatibility

SQL92

There is no `SET` defined in SQL92.

Name

UNLISTEN — Stop listening for notification

Synopsis

<refsynopsisdivinfo>1998-10-19</refsynopsisdivinfo>

```
UNLISTEN { notifyname | * }
```

Inputs

notifyname

Name of previously registered notify condition.

*

All current listen registrations for this backend are cleared.

Outputs

UNLISTEN

Acknowledgement that statement has executed.

Description

UNLISTEN is used to remove an existing NOTIFY registration. UNLISTEN cancels any existing registration of the current Postgres session as a listener on the notify condition *notifyname*. The special condition wildcard "*" cancels all listener registrations for the current session.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Notes

classname needs not to be a valid class name but can be any string valid as a name up to 32 characters long.

The backend does not complain if you UNLISTEN something you were not listening for. Each backend will automatically execute UNLISTEN * when exiting.

A restriction in some previous releases of Postgres that a *classname* which does not correspond to an actual table must be enclosed in double-quotes is no longer present.

Usage

```
postgres=> LISTEN virtual;
LISTEN
postgres=> NOTIFY virtual;
NOTIFY
ASYNC NOTIFY of 'virtual' from backend pid '12317' received

postgres=> UNLISTEN virtual;
```

```
UNLISTEN
postgres=> NOTIFY virtual;
NOTIFY
-- notice no NOTIFY event is received
postgres=>
```

Compatibility

SQL92

There is no `UNLISTEN` in SQL92.

Name

UPDATE — Replaces values of columns in a table

Synopsis

<refsynopsisdivinfo>1998-09-24</refsynopsisdivinfo>

```
UPDATE table SET column = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Inputs

table

The name of an existing table.

column

The name of a column in *table*.

expression

A valid expression or value to assign to column.

fromlist

A Postgres non-standard extension to allow columns from other tables to appear in the WHERE condition.

condition

Refer to the SELECT statement for a further description of the WHERE clause.

Outputs

UPDATE #

Message returned if successful. The # means the number of rows updated. If # is equal 0 no rows are updated.

Description

UPDATE changes the values of the columns specified for all rows which satisfy condition. Only the columns to be modified need appear as column.

Array references use the same syntax found in SELECT. That is, either single array elements, a range of array elements or the entire array may be replaced with a single query.

You must have write access to the table in order to modify it, as well as read access to any table whose values are mentioned in the WHERE condition.

Usage

```
--Change word "Drama" with "Dramatic" on column kind:
--
UPDATE films
   SET kind = 'Dramatic'
   WHERE kind = 'Drama';

SELECT * FROM films WHERE kind = 'Dramatic' OR kind = 'Drama';

code |title        |did| date_prod|kind      |len
-----+-------------+---+----------+----------+------
BL101|The Third Man|101|1949-12-23|Dramatic  | 01:44
P_302|Becket       |103|1964-02-03|Dramatic  | 02:28
M_401|War and Peace|104|1967-02-12|Dramatic  | 05:57
T_601|Yojimbo      |106|1961-06-16|Dramatic  | 01:50
DA101|Das Boot     |110|1981-11-11|Dramatic  | 02:29
```

Compatibility

SQL92

SQL92 defines a different syntax for positioned UPDATE statement:

```
UPDATE table SET column = expression [, ...]
       WHERE CURRENT OF cursor
```

where *cursor* identifies an open cursor.

Name

VACUUM — Clean and analyze a Postgres database

Synopsis

<refsynopsisdivinfo>1998-10-04</refsynopsisdivinfo>

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
VACUUM [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

Inputs

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates column statistics used by the optimizer to determine the most efficient way to execute a query. The statistics represent the disbursion of the data in each column. This information is valuable when several execution paths are possible.

table

The name of a specific table to vacuum. Defaults to all tables.

column

The name of a specific column to analyze. Defaults to all columns.

Outputs

VACUUM

The command has been accepted and the database is being cleaned.

NOTICE: --Relation *table*--

The report header for *table*.

NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0 sec.

The analysis for *table* itself.

NOTICE: Index *index*: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.

The analysis for an index on the target table.

Description

VACUUM serves two purposes in Postgres as both a means to reclaim storage and also a means to collect information for the optimizer.

VACUUM opens every class in the database, cleans out records from rolled back transactions, and updates statistics in the system catalogs. The statistics maintained include the number of tuples and number of pages stored in all classes. Running VACUUM periodically will increase the speed of the database in processing user queries.

Notes

The open database is target for VACUUM.

We recommend that active production databases be cleaned nightly, in order to keep statistics relatively current. The VACUUM query may be executed at any time, however. In particular, after copying a large class into Postgres or after deleting a large number of records, it may be a good idea to issue a VACUUM query. This will update the system catalogs with the results of all recent changes, and allow the Postgres query optimizer to make better choices in planning user queries.

If the server crashes during a VACUUM command, chances are it will leave a lock file hanging around. Attempts to re-run the VACUUM command result in an error message about the creation of a lock file. If you are sure VACUUM is not running, remove the pg_vlock file in your database directory (i.e. PGDATA/base/dbname/pg_vlock).

Usage

The following is an example from running VACUUM on a table in the regression database:

```
regression=> vacuum verbose analyze onek;
NOTICE:  --Relation onek--
NOTICE:  Pages 98: Changed 25, Reapped 74, Empty 0, New 0;
         Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen
 188;
         Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail.
 Pages 0/74.
         Elapsed 0/0 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Rel onek: Pages: 98 --> 25; Tuple(s) moved: 1000. Elapsed 0/1
 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
VACUUM
```

Compatibility

SQL92

There is no `VACUUM` statement in SQL92.

Chapter 20. Utility Applications

This is reference information for the Postgres support utilities.

Name

createdb — Create a new Postgres database

Synopsis

<refsynopsisdivinfo>1998-10-02</refsynopsisdivinfo>

```
createdb [ dbname ]
createdb [ -h host ] [ -p port ]
    [ -D datadir ]
    [ -u ] [ dbname ]
```

Inputs

-h *host*

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for *username* and *password*.

-D *datadir*

Specifies the alternate database location for this database installation. This is the location of the installation system tables, not the location of this specific database, which may be different.

dbname

Specifies the name of the database to be created. The name must be unique among all Postgres databases in this installation. *dbname* defaults to the value of the USER environment variable.

Outputs

createdb will create files in the PGDATA/*dbname*/ data area for the new database.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'? createdb: database creation failed on *dbname*.

createdb could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database 'template1' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow' createdb: database creation failed on *dbname*.

You do not have a valid entry in the relation pg_shadow and and will not be allowed to access Postgres. Contact your Postgres administrator.

ERROR: user '*username*' is not allowed to create/destroy databases createdb: database creation failed on *dbname*.

You do not have permission to create new databases. Contact your Postgres site administrator.

ERROR: createdb: database '*dbname*' already exists. createdb: database creation failed on *dbname*.

The database already exists.

createdb: database creation failed on *dbname*.

An internal error occurred in psql or in the backend server. Ensure that your site administrator has properly installed Postgresand initialized the site with initdb.

Note

createdb internally runs CREATE DATABASE from psql while connected to the `template1` database.

Description

createdb creates a new Postgres database. The person who executes this command becomes the database administrator, or DBA, for this database and is the only person, other than the Postgres super-user, who can destroy it.

createdb is a shell script that invokes psql. Hence, a postmaster process must be running on the database server host before createdb is executed. The `PGOPTION` and `PGREALM` environment variables will be passed on to psql and processed as described in psql.

Usage

To create the database `demo` using the postmaster on the local host, port 5432:

```
createdb demo
```

To create the database `demo` using the postmaster on host eden, port 5000:

```
createdb -p 5000 -h eden demo
```

Name

createuser — Create a new Postgres user

Synopsis

<refsynopsisdivinfo>1998-10-02</refsynopsisdivinfo>

```
createuser [ username ]
createuser [ -h host ] [ -p port ]
    [ -i userid ]
    [ -d | -D ] [ -u | -U ] [ username ]
```

Inputs

-h *host*

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-d

Allows the user to create databases.

-D

Forbids the user to create databases.

-i *userid*

Specifies the numeric identifier to be associated with this user. This identifier must be unique among all Postgres users, and is not required to match the operating system UID. You will be prompted for an identifier if none is specified on the command line, and it will suggest an identifier matching the UID.

-u

Allows the user to create other users.

-U

Forbids the user to create other users.

username

Specifies the name of the Postgres user to be created. This name must be unique among all Postgres users. You will be prompted for a name if none is specified on the command line.

Outputs

createuser will add an entry in the pg_user or pg_shadow system table.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'? createuser: database access failed.

createuser could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database 'template1' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow' createuser: database access failed.

You do not have a valid entry in the relation `pg_shadow` and and will not be allowed to access Postgres. Contact your Postgres administrator.

createuser: *username* cannot create users.

You do not have permission to create new users; contact your Postgres site administrator.

createuser: user "*username*" already exists

The user to be added already has an entry in the `pg_shadow` class.

database access failed

An internal error occurred in psql or in the backend server. Ensure that your site administrator has properly installed Postgresand initialized the site with initdb.

Note

createuser internally runs CREATE USER from psql while connected to the `template1` database.

Description

createuser creates a new Postgres user. Only users with `usesuper` set in the `pg_shadow` class can create new Postgres users. As shipped, the user `postgres` can create users.

createuser is a shell script that invokes psql. Hence, a postmaster process must be running on the database server host before createuser is executed. The `PGOPTION` and `PGREALM` environment variables will be passed on to psql and processed as described in psql. Once invoked, createuser will ask a series of questions to obtain parameters not specified on the command line. The new user's database login name and a numeric user identifier must be specified.

Note

The Postgres user identifier does not need to be the same as the user's Unix UID. However, typically they are assigned to be the same.

You must also describe the privileges of the new user for security purposes. Specifically, you will be asked whether the new user should be able to act as Postgres super-user, whether the new user may create new databases and whether the new user is allowed to create other new users.

Name

destroydb — Create a new Postgres database

Synopsis

<refsynopsisdivinfo>1998-10-02</refsynopsisdivinfo>

```
destroydb [ dbname ]
destroydb [ -h host ] [ -p port ]
    [ -i ] [ dbname ]
```

Inputs

-h *host*

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-i

Run in interactive mode. Prompts for confirmation before destroying a database.

dbname

Specifies the name of the database to be destroyed. The database must be one of the existing Postgres databases in this installation. *dbname* defaults to the value of the USER environment variable.

Outputs

destroydb will remove files from the PGDATA/*dbname*/ data area for the existing database.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'? destroydb: database destroy failed on *dbname*.

destroydb could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database 'template1' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow' destroydb: database destroy failed on *dbname*.

You do not have a valid entry in the relation pg_shadow and and will not be allowed to access Postgres. Contact your Postgres administrator.

ERROR: user '*username*' is not allowed to create/destroy databases destroydb: database destroy failed on *dbname*.

You do not have permission to destroy (or create) databases. Contact your Postgres site administrator.

ERROR: destroydb: database '*dbname*' does not exist. destroydb: database destroy failed on *dbname*.

The database to be removed does not have an entry in the `pg_database` class.

ERROR: destroydb: database '*dbname*' is not owned by you. destroydb: database destroy failed on *dbname*.

You are not the Database Administrator (DBA) for the specified database.

destroydb: database destroy failed on *dbname*.

An internal error occurred in psql or in the backend server. Ensure that your site administrator has properly installed Postgresand initialized the site with initdb.

Note

destroydb internally runs `DESTROY DATABASE` from psql while connected to the `template1` database.

Description

destroydb destroys an existing Postgres database. The person who executes this command must be the database administrator, or DBA, or must be the Postgres super-user. The program runs silently; no confirmation message will be displayed. After the database is destroyed, a Unix shell prompt will reappear.

All references to the database are removed, including the directory containing this database and its associated files.

destroydb is a shell script that invokes psql. Hence, a postmaster process must be running on the database server host before destroydb is executed. The `PGOPTION` and `PGREALM` environment variables will be passed on to psql and processed as described in psql.

Usage

To destroy the database `demo` using the postmaster on the local host, port 5432:

```
destroydb demo
```

To destroy the database `demo` using the postmaster on host eden, port 5000:

```
destroydb -p 5000 -h eden demo
```

Name

destroyuser — Destroy a Postgres user and associated databases

Synopsis

<refsynopsisdivinfo>1998-10-02</refsynopsisdivinfo>

```
destroyuser [ username ]
destroyuser [ -h host ] [ -p port ]
    [ username ]
```

Inputs

-h *host*

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

username

Specifies the name of the Postgres user to be removed. This name must exist in the Postgres installation. You will be prompted for a name if none is specified on the command line.

Outputs

destroyuser will remove an entry in the pg_user or pg_shadow system table, and will remove all databases for which that user is the administrator (DBA).

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'? destroyuser: database access failed.

destroyuser could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database 'template1' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow' destroyuser: database access failed.

You do not have a valid entry in the relation pg_shadow and and will not be allowed to access Postgres. Contact your Postgres administrator.

destroyuser: *username* cannot delete users.

You do not have permission to delete users; contact your Postgres site administrator.

destroyuser: user "*username*" already exists

The user to be added already has an entry in the pg_shadow class.

database access failed

An internal error occurred in psql or in the backend server. Ensure that your site administrator has properly installed Postgresand initialized the site with initdb.

destroydb on *dbname* failed - exiting

An internal error occurred in psql or in the backend server. There was possibly a Unix permissions problem with the specified database.

delete of user *username* was UNSUCCESSFUL

An internal error occurred in psql or in the backend server.

Note

destroyuser internally runs DROP USER from psql while connected to the `template1` database.

Description

destroyuser removes an existing Postgres user and the databases for which that user is database administrator. Only users with `usesuper` set in the `pg_shadow` class can destroy Postgres users. As shipped, the user `postgres` can remove users.

destroyuser is a shell script that invokes psql. Hence, a postmaster process must be running on the database server host before destroyuser is executed. The `PGOPTION` and `PGREALM` environment variables will be passed on to psql and processed as described in psql.

Once invoked, destroyuser will warn you about the databases that will be destroyed in the process and permit you to abort the removal of the user if desired.

Name

initdb — Create a new Postgres database installation

Synopsis

<refsynopsisdivinfo>1998-10-02</refsynopsisdivinfo>

```
initdb [ --pgdata=dbdir | -r dbdir ]
     [ --pglib=libdir | -l libdir ]
     [ --template=template | -t template ]
     [ --username=name | -u name ]
     [ --noclean | -n ] [ --debug | -d ]
```

Inputs

--pglib=*libdir*
-l *libdir*
PGLIB

Where are the files that make up Postgres? Apart from files that have to go in particular directories because of their function, the files that make up the Postgres software were installed in a directory called the *libdir* directory. An example of a file that will be found there that initdb needs is `global1.bki.source`, which contains all the information that goes into the shared catalog tables.

--pgdata=*dbdir*
-r *dbdir*
PGDATA

Where in your Unix filesystem do you want the database data to go? The top level directory is called the `PGDATA` directory.

--username=*name*
-u *name*
PGUSER

Who will be the Postgres superuser for this database system? The Postgres superuser is a Unix user who owns all files that store the database system and also owns the postmaster and backend processes that access them. Or just let it default to you (the Unix user who runs initdb).

Note

Only the Unix superuser (`root`) can create a database system with an owner different from the Postgres superuser.

Other, less commonly used, parameters are also available:

--template=*template*
-t *template*

Replace the `template1` database in an existing database system, and don't touch anything else. This is useful when you need to upgrade your `template1` database using initdb from a newer release of Postgres, or when your `template1` database has become corrupted by some system problem. Normally the contents of `template1` remain constant throughout the life of the database system. You can't destroy anything by running initdb with the `--template` option.

--noclean
-n

By default, when initdb determines that error prevent it from completely creating the database system, it removes any files it may have created before determining that it can't finish the job. That includes any core files left by the programs it invokes. This option inhibits any tidying-up and is thus useful for debugging.

--debug
-d

Print debugging output from the bootstrap backend. The bootstrap backend is the program initdb uses to create the catalog tables. This option generates a tremendous amount of output. It also turns off the final vacuuming step.

Files are also input to initdb:

postconfig

If appearing somewhere in the Unix command search path (defined by the PATH environment variable). This is a program that specifies defaults for some of the command options. See below.

`PGLIB/global1.bki.source`

Contents for the shared catalog tables in the new database system. This file is part of the Postgres software.

`PGLIB/local1_template1.bki.source`

Contents for the template1 tables in the new database system. This file is part of the Postgres software.

Outputs

initdb will create files in the `PGDATA` data area which are the system tables and framework for a complete installation.

Description

initdb creates a new Postgres database system. A database system is a collection of databases that are all administered by the same Unix user and managed by a single postmaster.

Creating a database system consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that don't belong to any particular database), and creating the `template1` database. What is the `template1` database? When you create a database, Postgres does it by copying everything from the `template1` database. It contains catalog tables filled in for things like the builtin types.

After initdb creates the database, it completes the initialization by running vacuum, which resets some optimization parameters.

There are three ways to give parameters to initdb. First, you can use initdb command options. Second, you can set environment variables before invoking initdb. Third, you can have a program called postconfig in your Unix command search path. initdb invokes that program and that program then writes initdb parameters to its standard output stream. This third option is not a common thing to do, however.

Command options always override parameters specified any other way. The values returned by postconfig override any environment variables, but your postconfig program may base its output on the environment variables if you want their values to be used.

The value that postconfig outputs must have the format

```
var1=value1 var2=value2 ...
```

It can output nothing if it doesn't want to supply any parameters. The *var* values are equal to the corresponding environment variable names. For example,

```
PGDATA=/tmp/postgres_test
```

has the same effect as invoking initdb with an environment variable called `PGDATA` whose value is `/tmp/postgres_test`.

Name

initlocation — Create a secondary Postgres database storage area

Synopsis

<refsynopsisdivinfo>1998-10-02</refsynopsisdivinfo>

```
initlocation [ --location=altdir | -D altdir ]
     [ --username=name | -u name ]
     [ altdir ]
```

Inputs

--location=*altdir*
-D *altdir*
altdir

Where in your Unix filesystem do you want alternate databases to go? The top level directory is called the PGDATA directory, so you might want to point your first alternate location at PGDATA2.

--username=*name*
-u *name*
PGUSER

Who will be the Unix filesystem owner of this database storage area? The Postgres superuser is a Unix user who owns all files that store the database system and also owns the postmaster and backend processes that access them. Usually, this is the user who should run initlocation and who will thus have ownership of the directories and files.

Note

Only the Unix superuser can create a database system with a different user as the Postgres superuser. Specifying a user other than the Postgres superuser may lead to database security and data integrity problems. Refer to the *PostgreSQL Administrator's Guide* for more information.

Outputs

initlocation will create directories in the specified place.

We are initializing the database area with username postgres (uid=500). This user will own all the files and must also own the server process. Creating Postgres database system directory *altdir* Creating Postgres database system directory *altdir*

Successful completion.

We are initializing the database area with username postgres (uid=500). This user will own all the files and must also own the server process. Creating Postgres database system directory /usr/local/src/testlocation mkdir: cannot make directory `*altdir*': Permission denied

You do not have filesystem permission to write to the specified directory area.

Valid username not given. You must specify the username for the Postgres superuser for the database system you are initializing, either with the --username option or by default to the USER environment variable.

The username which you have specified is not the Postgres superuser.

Can't tell what username to use. You don't have the USER environment variable set to your username and didn't specify the --username option

Specify the `--username` command line option.

Description

initlocation creates a new Postgres secondary database storage area. A secondary storage area contains a required tree of directories with the correct file permissions on those directories.

Creating a database storage area consists of creating the directories in which database data might live.

There are two kinds of arguments for initlocation. First, you can specify an environment variable (e.g. `PGDATA2`). This environment variable should be known to the backend for later use in `CREATE DATABASE/WITH LOCATION` or `createdb -D` *`altdir`*. However, *the backend daemon must have this variable in it's environment* for this to succeed. Second, you may be able to specify an explicit absolute path to the top directory of the storage area. However,this second option is possible only if explicitly enabled during the Postgres installation. It is usually disabled to alleviate security and data integrity concerns.

Note

Postgres will add `/base/` to the specified path to create the storage area.

The backend requires that any argument to `WITH LOCATION` which is in all uppercase and which has no path delimiters is an environment variable.

Usage

To create a database in an alternate location, using an environment variable:

```
% setenv PGDATA2 /opt/postgres/data

% initlocation PGDATA2
% createdb -D PGDATA2
```

Name

pg_dump — Extract a Postgres database into a script file

Synopsis

<refsynopsisdivinfo>1998-10-04</refsynopsisdivinfo>

```
pg_dump [ dbname ]
pg_dump [ -h host ] [ -p port ]
     [ -t table ]
     [ -f outputfile ]
     [ -a ] [ -d ] [ -D ] [ -o ] [ -s ] [ -u ] [ -v ] [ -z ]
     [ dbname ]
```

Inputs

pg_dump accepts the following command line arguments:

dbname

Specifies the name of the database to be extracted. *dbname* defaults to the value of the USER environment variable.

-a

Dump out only the data, no schema (definitions).

-d

Dump data as proper insert strings.

-D

Dump data as inserts with attribute names

-f *filename*

Specifies the output file. Defaults to stdout.

-n

Suppress double quotes around identifiers unless absolutely necessary. This may cause trouble loading this dumped data if there are reserved words used for identifiers.

-o

Dump object identifiers (OIDs) for every table.

-s

Dump out only the schema (definitions), no data.

-t *table*

Dump data for *table* only.

-u

Use password authentication. Prompts for username and password.

-v

Specifies verbose mode

-z

Include ACLs (grant/revoke commands) and table ownership information.

pg_dump also accepts the following command line arguments for connection parameters:

-h *host*

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for *username* and *password*.

Outputs

pg_dump will create a file or write to stdout.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'?

pg_dump could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database '*dbname*' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow'

You do not have a valid entry in the relation pg_shadow and and will not be allowed to access Postgres. Contact your Postgres administrator.

dumpSequence(*table*): SELECT failed

You do not have permission to read the database. Contact your Postgres site administrator.

Note

pg_dump internally executes SELECT statements. If you have problems running pg_dump, make sure you are able to select information from the database using, for example, psql.

Description

pg_dump is a utility for dumping out a Postgres database into a script file containing query commands. The script files are in text format and can be used to reconstruct the database, even on other machines and other architectures. pg_dump will produce the queries necessary to re-generate all user-defined types,

functions, tables, indices, aggregates, and operators. In addition, all the data is copied out in text format so that it can be readily copied in again, as well as imported into tools for editing.

pg_dump is useful for dumping out the contents of a database to move from one Postgres installation to another. After running pg_dump, one should examine the output script file for any warnings, especially in light of the limitations listed below.

Notes

pg_dump has a few limitations. The limitations mostly stem from difficulty in extracting certain meta-information from the system catalogs.

partial indices

pg_dump does not understand partial indices. The reason is the same as above; partial index predicates are stored as plans.

large objects

pg_dump does not handle large objects. Large objects are ignored and must be dealt with manually.

Usage

To dump a database of the same name as the user:

```
% pg_dump > db.out
```

To reload this database:

```
psql -e database < db.out
```

Name

pg_dumpall — Extract all Postgres databases into a script file

Synopsis

<refsynopsisdivinfo>1998-10-04</refsynopsisdivinfo>

```
pg_dumpall
pg_dumpall [ -h host ] [ -p port ]
    [ -a ] [ -d ] [ -D ] [ -o ] [ -s ] [ -u ] [ -v ] [ -z ]
```

Inputs

pg_dumpall accepts the following command line arguments:

-a

Dump out only the data, no schema (definitions).

-d

Dump data as proper insert strings.

-D

Dump data as inserts with attribute names

-n

Suppress double quotes around identifiers unless absolutely necessary. This may cause trouble loading this dumped data if there are reserved words used for identifiers.

-o

Dump object identifiers (OIDs) for every table.

-s

Dump out only the schema (definitions), no data.

-u

Use password authentication. Prompts for username and password.

-v

Specifies verbose mode

-z

Include ACLs (grant/revoke commands) and table ownership information.

pg_dumpall also accepts the following command line arguments for connection parameters:

-h *host*

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for *username* and *password*.

Outputs

pg_dumpall will create a file or write to stdout.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'?

pg_dumpall could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database '*dbname*' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow'

You do not have a valid entry in the relation pg_shadow and and will not be allowed to access Postgres. Contact your Postgres administrator.

dumpSequence(*table*): SELECT failed

You do not have permission to read the database. Contact your Postgres site administrator.

Note

pg_dumpall internally executes SELECT statements. If you have problems running pg_dumpall, make sure you are able to select information from the database using, for example, psql.

Description

pg_dumpall is a utility for dumping out all Postgres databases into one file. It also dumps the pg_shadow table, which is global to all databases. pg_dumpall includes in this file the proper commands to automatically create each dumped database before loading.

pg_dumpall takes all pg_dump options, but -f, -t and *dbname* should be omitted.

Refer to pg_dump for more information on this capability.

Usage

To dump all databases:

```
% pg_dumpall -o > db.out
```

Tip

You can use most pg_dump options for pg_dumpall.

To reload this database:

```
psql -e template1 < db.out
```

Tip

You can use most psql options when reloading.

Name

psql — Postgres interactive client

Synopsis

<refsynopsisdivinfo>1998-09-26</refsynopsisdivinfo>

```
psql [ dbname ]
psql -A [ -c query ] [ -d dbname ]
     -e [ -f filename ] [ -F separator ] [ -h hostname ] -Hln
     [ -o filename ] [ -p port ] -qsSt ] [ -T table_options ] -ux
  [ dbname ]
```

Inputs

psql accepts many command-line arguments, a rich set of meta-commands, and the full SQL language supported by Postgres. The most common command-line arguments are:

dbname

The name of an existing database to access. *dbname* defaults to the value of the USER environment variable or, if that's not set, to the Unix account name of the current user.

-c *query*

A single query to run. psql will exit on completion.

The full set of command-line arguments and meta-commands are described in a subsequent section.

There are some environment variables which can be used in liu of command line arguments. Additionally, the Postgres frontend library used by the psql application looks for other optional environment variables to configure, for example, the style of date/time representation and the local time zone. Refer to the chapter on libpq in the *Programmer's Guide* for more details.

You may set any of the following environment variables to avoid specifying command-line options:

PGHOST

The DNS host name of the database server. Setting PGHOST to a non-zero-length string causes TCP/IP communication to be used, rather than the default local Unix domain sockets.

PGPORT

The port number on which a Postgres server is listening. Defaults to 5432.

PGTTY

The target for display of messages from the client support library. Not required.

PGOPTION

If PGOPTION is specified, then the options it contains are parsed *before* any command-line options.

PGREALM

PGREALM only applies if Kerberos authentication is in use. If this environment variable is set, Postgres will attempt authentication with servers for this realm and will use separate ticket files to avoid

conflicts with local ticket files. See the *PostgreSQL Administrator's Guide* for additional information on Kerberos.

Outputs

psql returns 0 to the shell on successful completion of all queries, 1 for errors, 2 for abrupt disconnection from the backend. The default TAB delimiter is used. psql will also return 1 if the connection to a database could not be made for any reason.

Description

psql is a character-based front-end to Postgres. It enables you to type in queries interactively, issue them to Postgres, and see the query results.

psql is a Postgres client application. Hence, a postmaster process must be running on the database server host before psql is executed. In addition, the correct parameters to identify the database server, such as the postmaster host name, may need to be specified as described below.

When psql starts, it reads SQL commands from `/etc/psqlrc` and then from `$(HOME)/.psqlrc` This allows SQL commands like `SET` which can be used to set the date style to be run at the start of every session.

Connecting To A Database

psql attempts to make a connection to the database at the hostname and port number specified on the command line. If the connection could not be made for any reason (e.g. insufficient privileges, postmaster is not running on the server, etc) .IR psql will return an error that says

```
Connection to database failed.
```

The reason for the connection failure is not provided.

Entering Queries

In normal operation, psql provides a prompt with the name of the database that psql is current connected to followed by the string "=>". For example,

```
$ psql testdb
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \e? for help on slash commands
   type \eq to quit
   type \eg or terminate with semicolon to execute query
 You are currently connected to the database: testdb

testdb=>
```

At the prompt, the user may type in SQL queries. Unless the -S option is set, input lines are sent to the backend when a query-terminating semicolon is reached.

Whenever a query is executed, psql also polls for asynchronous notification events generated by `LISTEN` and `NOTIFY`.

psql can be used in a pipe sequence, and automatically detects when it is not listening or talking to a real tty.

Command-line Options

psql understands the following command-line options:

-A

Turn off fill justification when printing out table elements.

-c *query*

Specifies that psql is to execute one query string, *query*, and then exit. This is useful for shell scripts, typically in conjunction with the `-q` option in shell scripts.

-d *dbname*

Specifies the name of the database to connect to. This is equivalent to specifying *dbname* as the last field in the command line.

-e

Echo the query sent to the backend

-f *filename*

Use the file *filename* as the source of queries instead of reading queries interactively. This file must be specified for and visible to the client frontend.

-F *separator*

Use *separator* as the field separator. The default is an ASCII vertical bar ("|").

-h *hostname*

Specifies the host name of the machine on which the postmaster is running. Without this option, communication is performed using local Unix domain sockets.

-H

Turns on HTML 3.0 tabular output.

-l

Lists all available databases, then exit. Other non-connection options are ignored.

-n

Do not use the readline library for input line editing and command history.

-o *filename*

Put all output into file *filename*. The path must be writable by the client.

-p *port*

Specifies the TCP/IP port or, by omission, the local Unix domain socket file extension on which the postmaster is listening for connections. Defaults to the value of the `PGPORT` environment variable, if set, or to 5432.

-q

Specifies that psql should do its work quietly. By default, it prints welcome and exit messages and prompts for each query, and prints out the number of rows returned from a query. If this option is used, none of this happens. This is useful with the `-c` option.

-s

Run in single-step mode where the user is prompted for each query before it is sent to the backend.

-S

Runs in single-line mode where each query is terminated by a newline, instead of a semicolon.

-t

Turn off printing of column names. This is useful with the `-c` option in shell scripts.

-T *`table_options`*

Allows you to specify options to be placed within the `table ...` tag for HTML 3.0 tabular output.For example, `border` will give you tables with borders. This must be used in conjunction with the `-H` option.

-u

Asks the user for the user name and password before connecting to the database. If the database does not require password authentication then these are ignored. If the option is not used (and the PGPASSWORD environment variable is not set) and the database requires password authentication, then the connection will fail. The user name is ignored anyway.

-x

Turns on extended row format mode. When enabled each row will have its column names printed on the left with the column values printed on the right. This is useful for rows which are otherwise too long to fit into one screen line. HTML row output supports this mode also.

You may set environment variables to avoid typing some of the above options. See the section on environment variables below.

psql Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command. Anything else is SQL and simply goes into the current query buffer (and once you have at least one complete query, it gets automatically submitted to the backend). psql meta-commands are also called slash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of white space characters.

With single character command verbs, you don't actually need to separate the command verb from the argument with white space, for historical reasons. You should anyway.

The following meta-commands are defined:

`\a`

Toggle field alignment when printing out table elements.

`\C` *caption*

Set the HTML3.0 table caption to "*caption*".

`\connect` *dbname* [*username*]

Establish a connection to a new database, using the default *username* if none is specified. The previous connection is closed.

`\copy` *dbname* { FROM | TO } *filename*

Perform a frontend (client) copy. This is an operation that runs a SQL COPY command, but instead of the backend reading or writing the specified file, and consequently requiring backend access and special user privilege, psql reads or writes the file and routes the data to or from the backend. The default `tab` delimiter is used.

Tip

This operation is not as efficient as the SQL `COPY` command because all data must pass through the client/server IP or socket connection. For large amounts of data this other technique may be preferable.

`\d` [*table*]

List tables in the database, or if *table* is specified, list the columns in that table. If table name is specified as an asterisk ("*"), list all tables and column information for each tables.

`\da`

List all available aggregates.

`\dd` *object*

List the description from `pg_description` of the specified object, which can be a table, table.column, type, operator, or aggregate.

Tip

Not all objects have a description in `pg_description`. This meta-command can be useful to get a quick description of a native Postgres feature.

`\df`

List functions.

`\di`

List only indexes.

`\do`

List only operators.

`\ds`

List only sequences.

`\dS`

List system tables and indexes.

`\dt`

List only non-system tables.

`\dT`

List types.

`\e [ filename ]`

Edit the current query buffer or the contents of the file *filename*.

`\E [ filename ]`

Edit the current query buffer or the contents of the file *filename* and execute it upon editor exit.

`\f [ separator ]`

Set the field separator. Default is a single blank space.

`\g [ { filename | |command } ]`

Send the current query input buffer to the backend and optionally save the output in *filename* or pipe the output into a separate Unix shell to execute *command*.

`\h [ command ]`

Give syntax help on the specified SQL command. If *command* is not a defined SQL command (or is not documented in psql), or if *command* is not specified, then psql will list all the commands for which syntax help is available. If *command* is an asterisk ("*"), then give syntax help on all SQL commands.

`\H`

Toggle HTML3 output. This is equivalent to the `-H` command-line option.

`\i filename`

Read queries from the file *filename* into the query input buffer.

`\l`

List all the databases in the server.

`\m`

Toggle the old monitor-like table display, which includes border characters surrounding the table. This is standard SQL output. By default, psql includes only field separators between columns.

`\o [ { filename | |command } ]`

Save future query results to the file *filename* or pipe future results into a separate Unix shell to execute *command*. If no arguments are specified, send query results to `stdout`.

`\p`

Print the current query buffer.

`\q`

Quit the psql program.

`\r`

Reset(clear) the query buffer.

`\s [ filename ]`

Print or save the command line history to *filename*. If *filename* is omitted, do not save subsequent commands to a history file. This option is only available if psql is configured to use readline.

`\t`

Toggle display of output column name headings and row count footer (defaults to on).

`\T table_options`

Allows you to specify options to be placed within the `table ...` tag for HTML 3.0 tabular output.For example, `border` will give you tables with borders. This must be used in conjunction with the `\H` meta-command.

`\x`

Toggles extended row format mode. When enabled each row will have its column names printed on the left with the column values printed on the right. This is useful for rows which are otherwise too long to fit into one screen line. HTML row output mode supports this flag too.

`\w filename`

Outputs the current query buffer to the file *filename*.

`\z`

Produces a list of all tables in the database with their appropriate ACLs (grant/revoke permissions) listed.

`\! [ command ]`

Escape to a separate Unix shell or execute the Unix command *command*.

`\?`

Get help information about the slash ("\") commands.

Part III. Administrator's Guide

Installation and maintenance information.

Table of Contents

Chapter 21. Introduction

This document is the Administrator's Manual for the PostgreSQL[1] database management system, originally developed at the University of California at Berkeley. PostgreSQL is based on Postgres release 4.2[2]. 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.

21.1. Resources

This manual set is organized into several parts:

Tutorial

An introduction for new users. Does not cover advanced features.

User's Guide

General information for users, including available commands and data types.

Programmer's Guide

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

Administrator's Guide

Installation and management information. List of supported machines.

Developer's Guide

Information for Postgres developers. This is intended for those who are contributing to the Postgres project; application development information should appear in the *Programmer's Guide*. Currently included in the *Programmer's Guide*.

Reference Manual

Detailed reference information on command syntax. Currently included in the *User's Guide*.

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

man pages

The man pages have general information on command syntax.

FAQs

The Frequently Asked Questions (FAQ) documents address both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

[1] http://postgresql.org/
[2] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

Web Site

The Postgres[3] web site has some information not appearing in the distribution. There is a mhonarc catalog of mailing list traffic which is a rich resource for many topics.

Mailing Lists

The Postgres Questions[4] mailing list is a good place to have user questions answered. Other mailing lists are available; consult the web page for details.

Yourself!

Postgres is an open source product. As such, it depends on the user community for ongoing support. As you begin to use Postgres, 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 it. 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 Postgres Documentation[5] mailing list is the place to get going.

21.2. Terminology

In the following documentation, *site* may be interpreted as the host machine on which Postgres is installed. Since it is possible to install more than one set of Postgres databases on a single host, this term more precisely denotes any particular set of installed Postgres binaries and databases.

The Postgres *superuser* is the user named `postgres` who owns the Postgres binaries and database files. As the database superuser, all protection mechanisms may be bypassed and any data accessed arbitrarily. In addition, the Postgres superuser is allowed to execute some support programs which are generally not available to all users. Note that the Postgres superuser is *not* the same as the Unix superuser (which will be referred to as *root*). The superuser should have a non-zero user identifier (*UID*) for security reasons.

The *database administrator* or DBA, is the person who is responsible for installing Postgres with mechanisms to enforce a security policy for a site. The DBA can add new users by the method described below and maintain a set of template databases for use by createdb.

The postmaster is the process that acts as a clearing-house for requests to the Postgres system. Frontend applications connect to the postmaster, which keeps tracks of any system errors and communication between the backend processes. The postmaster can take several command-line arguments to tune its behavior. However, supplying arguments is necessary only if you intend to run multiple sites or a non-default site.

The Postgres backend (the actual executable program postgres) may be executed directly from the user shell by the Postgres super-user (with the database name as an argument). However, doing this bypasses the shared buffer pool and lock table associated with a postmaster/site, therefore this is not recommended in a multiuser site.

21.3. Notation

"..." or `/usr/local/pgsql/` at the front of a file name is used to represent the path to the Postgres superuser's home directory.

[3] postgresql.org
[4] mailto:questions@postgresql.org
[5] mailto:docs@postgresql.org

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.

In examples, parentheses (“(” and “)”) are used to group boolean expressions. “|” is the boolean operator OR.

Examples will show commands executed from various accounts and programs. Commands executed from the root account will be preceeded with “>”. Commands executed from the Postgres superuser account will be preceeded with “%”, while commands executed from an unprivileged user's account will be preceeded with “$”. SQL commands will be preceeded with “=>” or will have no leading prompt, depending on the context.

Note

At the time of writing (Postgres v6.4) the notation for flagging commands is not universally consistant throughout the documentation set. Please report problems to the Documentation Mailing List[6].

21.4. Y2K Statement

Author

Written by Thomas Lockhart[7] on 1998-10-22.

The PostgreSQL Global Development Team provides the Postgres software code tree as a public service, without warranty and without liability for it's behavior or performance. However, at the time of writing:

- The author of this statement, a volunteer on the Postgres support team since November, 1996, is not aware of any problems in the Postgres code base related to time transitions around Jan 1, 2000 (Y2K).
- The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of Postgres. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.
- To the best of the author's knowledge, the assumptions Postgres makes about dates specified with a two-digit year are documented in the current User's Guide[8] in the chapter on data types. For two-digit years, the significant transition year is 1970, not 2000; e.g. “70-01-01” is interpreted as “1970-01-01”, whereas “69-01-01” is interpreted as “2069-01-01”.
- Any Y2K problems in the underlying OS related to obtaining "the current time" may propagate into apparent Y2K problems in Postgres.

Refer to The Gnu Project[9] and The Perl Institute[10] for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

21.5. Copyrights and Trademarks

[6] mailto:docs@postgresql.org
[7] mailto:lockhart@alumni.caltech.edu
[8] http://www.postgresql.org/docs/user/datatype.htm
[9] http://www.gnu.org/software/year2000.html
[10] http://language.perl.com/news/y2k.html

Chapter 22. Ports

This manual describes version 6.4 of Postgres. The Postgres developer community has compiled and tested Postgres on a number of platforms. Check the web site[1] for the latest information.

22.1. Currently Supported Platforms

At the time of publication, the following platforms have been tested:

Table 22.1. Supported Platforms

OS	Processor	Version	Reported	Remarks
AIX 4.2.1	RS6000	v6.4	1998-10-27	(Andreas Zeugswetter[2])
BSDI	x86	v6.4	1998-10-25	(Bruce Momjian[3]
FreeBSD 2.2.x-3.x	x86	v6.4	1998-10-26	(Tatsuo Ishii[4], Marc Fournier[5])
DGUX 5.4R4.11	m88k	v6.3	1998-03-01	v6.4 probably OK. Needs new maintainer. (Brian E Gallew[6])
Digital Unix 4.0	Alpha	v6.4	1998-10-29	Minor patchable problems (Pedro J. Lobo[7])
HPUX	PA-RISC	v6.4	1998-10-25	Both 9.0x and 10.20 (Tom Lane[8], Stan Brown[9])
IRIX 6.x	MIPS	v6.3	1998-03-01	5.x is different (Andrew Martin[10])
linux 2.0.x	Alpha	v6.3.2	1998-04-16	Mostly successful. Needs work for v6.4. (Ryan Kirkpatrick[11])
linux 2.0.x	x86	v6.4	1998-10-27	(Thomas Lockhart[12])

[1] http://www.postgresql.org/docs/admin/ports.htm
[2] mailto:Andreas.Zeugswetter@telecom.at
[3] mailto:maillist@candle.pha.pa.us
[4] mailto:t-ishii@sra.co.jp
[5] mailto:scrappy@hub.org
[6] mailto:geek+@cmu.edu
[7] mailto:pjlobo@euitt.upm.es
[8] mailto:tgl@sss.pgh.pa.us
[9] mailto:stanb@awod.com
[10] mailto:martin@biochemistry.ucl.ac.uk
[11] mailto:rkirkpat@nag.cs.colorado.edu
[12] mailto:lockhart@alumni.caltech.edu

OS	Processor	Version	Reported	Remarks
linux 2.0.x/glibc2	x86	v6.4	1998-10-25	(Oliver Elphick[13], Taral[14])
linux 2.0.x	Sparc	v6.4	1998-10-25	(Tom Szybist[15])
linuxPPC 2.1.24	PPC603e	v6.4	1998-10-26	Powerbook 2400c(Tatsuo Ishii[16])
mklinux DR3	PPC750	v6.4	1998-09-16	PowerMac 7600 (Tatsuo Ishii[17])
NetBSD/i386 1.3.2	x86	v6.4	1998-10-25	(Brook Milligan[18])
NetBSD-current	NS32532	v6.4	1998-10-27	(small problems in date/time math (Jon Buller[19])
NetBSD/sparc 1.3H	Sparc	v6.4	1998-10-27	(Tom I Helbekkmo[20])
NetBSD 1.3	VAX	v6.3	1998-03-01	(Tom I Helbekkmo[21])
SCO UnixWare 2.x	x86	v6.3	1998-03-01	aka UNIVEL (Billy G. Allie[22])
SCO UnixWare 7	x86	v6.4	1998-10-04	(Billy G. Allie[23])
Solaris	x86	v6.4	1998-10-28	(Marc Fournier[24])
Solaris 2.6-2.7	Sparc	v6.4	1998-10-28	(Tom Szybist[25], Frank Ridderbusch[26])
SunOS 4.1.4	Sparc	v6.3	1998-03-01	patches submitted (Tatsuo Ishii[27])
SVR4	MIPS	v6.4	1998-10-28	no 64-bit int support (Frank Ridderbusch[28])
SVR4 4.4	m88k	v6.2.1	1998-03-01	confirmed with patching (Doug Winterburn[29])

[13] mailto:olly@lfix.co.uk
[14] mailto:taral@mail.utexas.edu
[15] mailto:szybist@boxhill.com
[16] mailto:t-ishii@sra.co.jp
[17] mailto:t-ishii@sra.co.jp
[18] mailto:brook@trillium.NMSU.Edu
[19] mailto:jonb@metronet.com
[20] mailto:tih@hamartun.priv.no
[21] mailto:tih@hamartun.priv.no
[22] mailto:Bill.Allie@mug.org
[23] mailto:Bill.Allie@mug.org
[24] mailto:scrappy@hub.org
[25] mailto:szybist@boxhill.com
[26] mailto:ridderbusch.pad@sni.de
[27] mailto:t-ishii@sra.co.jp
[28] mailto:ridderbusch.pad@sni.de
[29] mailto:dlw@seavme.xroads.com

OS	Processor	Version	Reported	Remarks
Windows NT	x86	v6.4	1998-10-08	Mostly working with the Cygwin library. No DLLs yet. (Horak Daniel[30])

Platforms listed for v6.3.x should also work with v6.4, but we did not receive confirmation of such at the time this list was compiled.

Note

For Windows NT, the server-side port of Postgres has recently been accomplished. Check the Askesis Postgres Home Page[31] for up to date information. You may also want to look for possible patches on the Postgres web site[32].

22.2. Unsupported Platforms

There are a few platforms which have been attempted and which have been reported to not work with the standard distribution. Others listed here do not provide sufficient library support for an attempt.

Table 22.2. Possibly Incompatible Platforms

OS	Processor	Version	Reported	Remarks
MacOS	all	v6.3	1998-03-01	not library compatible; use ODBC/JDBC
NetBSD	arm32	v6.3	1998-03-01	not yet working (Dave Millen[33])
NetBSD	m68k	v6.3	1998-03-01	Amiga, HP300, Mac; not yet working (Henry Hotz[34])
NextStep	x86	v6.x	1998-03-01	client-only support; v1.0.9 worked with patches (David Wetzel[35])
Ultrix	MIPS,VAX?	v6.x	1998-03-01	no recent reports; obsolete?
Windows	x86	v6.3	1998-03-01	not library compatible; client side maybe; use ODBC/JDBC

[30] mailto:Dan.Horak@email.cz
[31] http://www.askesis.nl/AskesisPostgresIndex.html
[32] http://postgresql.org
[33] mailto:dmill@globalnet.co.uk
[34] mailto:hotz@jpl.nasa.gov
[35] mailto:dave@turbocat.de

Note that Windows ports of the frontend are apparently possible using third-party Posix porting tools and libraries.

Chapter 23. Configuration Options

23.1. Parameters for Configuration (configure)

The full set of parameters available in configure can be obtained by typing

```
$ ./configure --help
```

The following parameters may be of interest to installers:

```
Directory and file names:
  --prefix=PREFIX         install architecture-independent files in
 PREFIX
                          [/usr/local/pgsql]
  --bindir=DIR            user executables in DIR [EPREFIX/bin]
  --libdir=DIR            object code libraries in DIR [EPREFIX/lib]
  --includedir=DIR        C header files in DIR [PREFIX/include]
  --mandir=DIR            man documentation in DIR [PREFIX/man]
Features and packages:
  --disable-FEATURE       do not include FEATURE (same as --enable-
FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-
PACKAGE=no)
--enable and --with options recognized:
  --with-template=template
                          use operating system template file
                              see template directory
  --with-includes=incdir  site header files for tk/tcl, etc in DIR
  --with-libs=incdir      also search for libraries in DIR
  --with-libraries=libdir also search for libraries in DIR
  --enable-locale         enable locale support
  --enable-recode         enable cyrillic recode support
  --with-mb=encoding    enable multi-byte support
  --with-pgport=portnum change default startup port
  --with-tcl              use tcl
  --with-tclconfig=tcldir tclConfig.sh and tkConfig.sh are in DIR
  --with-perl             use perl
  --with-odbc             build ODBC driver package
  --with-odbcinst=odbcdir change default directory for odbcinst.ini
  --enable-cassert        enable assertion checks (debugging)
  --with-CC=compiler      use specific C compiler
  --with-CXX=compiler     use specific C++ compiler
```

Some systems may have trouble building a specific feature of Postgres. For example, systems with a damaged C++ compiler may need to specify `--without-CXX` to encourage the build procedure to ignore the `libpq++` construction.

23.2. Parameters for Building (make)

Many installation-related parameters can be set in the building stage of Postgres installation.

In most cases, these parameters should be place in a file, `Makefile.custom`, intended just for that purpose. The default distribution does not contain this optional file, so you will create it using a text editor of your choice. When upgrading installations, you can simply copy your old Makefile.custom to the new installation before doing the build.

`make [ variable=value [,...] ]`

A few of the many variables which can be specified are:

`POSTGRESDIR`

Top of the installation tree.

`BINDIR`

Location of applications and utilities.

`LIBDIR`

Location of object libraries, including shared libraries.

`HEADERDIR`

Location of include files.

`ODBCINST`

Location of installation-wide psqlODBC (ODBC) configuration file.

There are other optional parameters which are not as commonly used. Many of those listed below are appropriate when doing Postgres server code development.

`CFLAGS`

Set flags for the C compiler. Should be assigned with "+=" to retain relevant default parameters.

YFLAGS

Set flags for the yacc/bison parser. `-v` might be used to help diagnose problems building a new parser. Should be assigned with "+=" to retain relevant default parameters.

`USE_TCL`

Enable Tcl interface building.

`HSTYLE`

DocBook HTML style sheets for building the documentation from scratch. Not used unless you are developing new documentation from the DocBook-compatible SGML source documents in `doc/src/sgml/`.

`PSTYLE`

DocBook style sheets for building printed documentation from scratch. Not used unless you are developing new documentation from the DocBook-compatible SGML source documents in `doc/src/sgml/`.

Here is an example `Makefile.custom` for a PentiumPro Linux system:

```
# Makefile.custom
```

```
# Thomas Lockhart 1998-03-01

POSTGRESDIR= /opt/postgres/current
CFLAGS+= -m486 # -g -O0
USE_TCL= true
TCL_LIB= -ltcl
X_LIBS= -L/usr/X11/lib
TK_LIB= -ltk

# documentation

HSTYLE= /home/tgl/SGML/db118.d/docbook/html
PSTYLE= /home/tgl/SGML/db118.d/docbook/print
```

23.3. Locale Support

Note

Written by Oleg Bartunov. See Oleg's web page[1] for additional information on locale and Russian language support.

While doing a project for a company in Moscow, Russia, I encountered the problem that postgresql had no support of national alphabets. After looking for possible workarounds I decided to develop support of locale myself. I'm not a C-programer but already had some experience with locale programming when I work with perl (debugging) and glimpse. After several days of digging through the Postgres source tree I made very minor corections to src/backend/utils/adt/varlena.c and src/backend/main/main.c and got what I needed! I did support only for `LC_CTYPE` and `LC_COLLATE`, but later `LC_MONETARY` was added by others. I got many messages from people about this patch so I decided to send it to developers and (to my surprise) it was incorporated into the Postgres distribution.

People often complain that locale doesn't work for them. There are several common mistakes:

- Didn't properly configure postgresql before compilation. You must run configure with --enable-locale option to enable locale support. Didn't setup environment correctly when starting postmaster. You must define environment variables `LC_CTYPE` and `LC_COLLATE` before running postmaster because backend gets information about locale from environment. I use following shell script (runpostgres):

  ```
  #!/bin/sh

  export LC_CTYPE=koi8-r
  export LC_COLLATE=koi8-r
  postmaster -B 1024 -S -D/usr/local/pgsql/data/ -o '-Fe'
  ```

  and run it from rc.local as

  ```
  /bin/su - postgres -c "/home/postgres/runpostgres"
  ```

- Broken locale support in OS (for example, locale support in libc under Linux several times has changed and this caused a lot of problems). Latest perl has also support of locale and if locale is broken `perl -v` will complain something like:

  ```
  8:17[mira]:~/WWW/postgres>setenv LC_CTYPE not_exist
  8:18[mira]:~/WWW/postgres>perl -v
  ```

[1] http://www.sai.msu.su/~megera/postgres/

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
        LC_ALL = (unset),
        LC_CTYPE = "not_exist",
        LANG = (unset)
    are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

- Wrong location of locale files! Possible locations include: `/usr/lib/locale` (Linux, Solaris), `/usr/share/locale` (Linux), `/usr/lib/nls/loc` (DUX 4.0). Check `man locale` to find the correct location. Under Linux I did a symbolic link between `/usr/lib/locale` and `/usr/share/locale` to be sure that the next libc will not break my locale.

23.3.1. What are the Benefits?

You can use ~* and order by operators for strings contain characters from national alphabets. Non-english users definitely need that. If you won't use locale stuff just undefine the USE_LOCALE variable.

23.3.2. What are the Drawbacks?

There is one evident drawback of using locale - it's speed! So, use locale only if you really need it.

23.4. Kerberos Authentication

Kerberos is an industry-standard secure authentication system suitable for distributed computing over a public network.

23.4.1. Availability

The Kerberos authentication system is not distributed with Postgres. Versions of Kerberos are typically available as optional software from operating system vendors. In addition, a source code distribution may be obtained through MIT Project Athena[2].

Note

You may wish to obtain the MIT version even if your vendor provides a version, since some vendor ports have been deliberately crippled or rendered non-interoperable with the MIT version.

Users located outside the United States of America and Canada are warned that distribution of the actual encryption code in Kerberos is restricted by U. S. Government export regulations.

Inquiries regarding your Kerberos should be directed to your vendor or MIT Project Athena[3]. Note that FAQLs (Frequently-Asked Questions Lists) are periodically posted to the Kerberos mailing list[4] (send mail to subscribe[5]), and USENET news group[6].

23.4.2. Installation

Installation of Kerberos itself is covered in detail in the *Kerberos Installation Notes* . Make sure that the server key file (the `srvtab` or `keytab`) is somehow readable by the Postgres account.

[2] ftp://athena-dist.mit.edu
[3] info-kerberos@athena.mit.edu
[4] mailto:kerberos@ATHENA.MIT.EDU
[5] mailto:kerberos-request@ATHENA.MIT.EDU
[6] news:comp.protocols.kerberos

Postgres and its clients can be compiled to use either Version 4 or Version 5 of the MIT Kerberos protocols by setting the `KRBVERS` variable in the file `src/Makefile.global` to the appropriate value. You can also change the location where Postgres expects to find the associated libraries, header files and its own server key file.

After compilation is complete, Postgres must be registered as a Kerberos service. See the *Kerberos Operations Notes* and related manual pages for more details on registering services.

23.4.3. Operation

After initial installation, Postgres should operate in all ways as a normal Kerberos service. For details on the use of authentication, see the *PostgreSQL User's Guide* reference sections for postmaster and psql.

In the Kerberos Version 5 hooks, the following assumptions are made about user and service naming:

- User principal names (anames) are assumed to contain the actual Unix/Postgres user name in the first component.
- The Postgres service is assumed to be have two components, the service name and a hostname, canonicalized as in Version 4 (i.e., with all domain suffixes removed).

Table 23.1. Kerberos Parameter Examples

Parameter	Example
user	frew@S2K.ORG
user	aoki/HOST=miyu.S2K.Berkeley.EDU@S2K.ORG
host	postgres_dbms/ucbvax@S2K.ORG

Support for Version 4 will disappear sometime after the production release of Version 5 by MIT.

Chapter 24. Installation

Complete installation instructions for Postgres v6.4.

Before installing Postgres, you may wish to visit www.postgresql.org[1] for up to date information, patches, etc.

These installation instructions assume:

- Commands are Unix-compatible. See note below.
- Defaults are used except where noted.
- User `postgres` is the Postgres superuser.
- The source path is `/usr/src/pgsql` (other paths are possible).
- The runtime path is `/usr/local/pgsql` (other paths are possible).

Commands were tested on RedHat Linux version 4.2 using the tcsh shell. Except where noted, they will probably work on most systems. Commands like `ps` and `tar` may vary wildly between platforms on what options you should use. *Use common sense* before typing in these commands.

Our Makefiles require GNU make (called "gmake" in this document). They will *not* work with non-GNU make programs. If you have GNU make installed under the name "make" instead of "gmake", then you will use the command `make` instead. That's OK, but you need to have the GNU form of make to succeed with an installation.

24.1. Requirements to Run Postgres

Up to date information on supported platforms is at http://www.postgresql.org/docs/admin/install.htm. In general, most Unix-compatible platforms with modern libraries should be able to run Postgres.

Although the minimum required memory for running Postgres is as little as 8MB, there are noticable improvements in runtimes for the regression tests when expanding memory up to 96MB on a relatively fast dual-processor system running X-Windows. The rule is you can never have too much memory.

Check that you have sufficient disk space. You will need about 30 Mbytes for `/usr/src/pgsql`, about 5 Mbytes for `/usr/local/pgsql` (excluding your database) and 1 Mbyte for an empty database. The database will temporarily grow to about 20 Mbytes during the regression tests. You will also need about 3 Mbytes for the distribution tar file.

We therefore recommend that during installation and testing you have well over 20 Mbytes free under `/usr/local` and another 25 Mbytes free on the disk partition containing your database. Once you delete the source files, tar file and regression database, you will need 2 Mbytes for `/usr/local/pgsql`, 1 Mbyte for the empty database, plus about five times the space you would require to store your database data in a flat file.

To check for disk space, use

```
$ df -k
```

24.2. Installation Procedure

[1] http://www.postgresql.org

Postgres Installation

For a fresh install or upgrading from previous releases of Postgres:

1. Read any last minute information and platform specific porting notes. There are some platform specific notes at the end of this file for Ultrix4.x, Linux, BSD/OS and NeXT. There are other files in directory `/usr/src/pgsql/doc`, including files FAQ-Irix and FAQ-Linux. Also look in directory ftp://ftp.postgresql.org/pub. If there is a file called INSTALL in this directory then this file will contain the latest installation information.

   Please note that a "tested" platform in the list given earlier simply means that someone went to the effort at some point of making sure that a Postgres distribution would compile and run on this platform without modifying the code. Since the current developers will not have access to all of these platforms, some of them may not compile cleanly and pass the regression tests in the current release due to minor problems. Any such known problems and their solutions will be posted in ftp://ftp.postgresql.org/pub/INSTALL.

2. (Optional) Create the Postgres superuser account (`postgres` is commonly used) if it does not already exist.

   The owner of the Postgres files can be any unprivileged user account. It *must not* be `root`, `bin`, or any other account with special access rights, as that would create a security risk.

3. Log in to the Postgres superuser account. Most of the remaining steps in the installation will happen in this account.

4. Ftp file `ftp://ftp.postgresql.org/pub/postgresql-v6.4.tar.gz`[2] from the Internet. Store it in your home directory.

5. Some platforms use flex. If your system uses flex then make sure you have a good version. To check, type

   ```
   $ flex --version
   ```

   If the flex command is not found then you probably do not need it. If the version is 2.5.2 or 2.5.4 or greater then you are okay. If it is 2.5.3 or before 2.5.2 then you will have to upgrade flex. You may get it at ftp://prep.ai.mit.edu/pub/gnu/flex-2.5.4.tar.gz.

   If you need flex and don't have it or have the wrong version, then you will be told so when you attempt to compile the program. Feel free to skip this step if you aren't sure you need it. If you do need it then you will be told to install/upgrade flex when you try to compile Postgres.

   You may want to do the entire flex installation from the root account, though that is not absolutely necessary. Assuming that you want the installation to place files in the usual default areas, type the following:

   ```
   $ su -
   $ cd /usr/local/src
   ftp prep.ai.mit.edu
   ftp> cd /pub/gnu/
   ftp> binary
   ftp> get flex-2.5.4.tar.gz
   ftp> quit
   $ gunzip -c flex-2.5.4.tar.gz | tar xvf -
   $ cd flex-2.5.4
   ```

[2] ftp://ftp.postgresql.org/pub/postgresql-v6.4.tar.gz

```
$ configure --prefix=/usr
$ gmake
$ gmake check
# You must be root when typing the next line:
$ gmake install
$ cd /usr/local/src
$ rm -rf flex-2.5.4
```

This will update files `/usr/man/man1/flex.1`, `/usr/bin/flex`, `/usr/lib/libfl.a`, `/usr/include/FlexLexer.h` and will add a link `/usr/bin/flex++` which points to flex.

6. If you are not upgrading an existing system then skip to Step 9. If you are upgrading an existing system then back up your database. For alpha- and beta-level releases, the database format is liable to change, often every few weeks, with no notice besides a quick comment in the HACKERS mailing list. Full releases always require a dump/reload from previous releases. It is therefore a bad idea to skip this step.

   ## Tip

   Do not use the pg_dumpall script from v6.0 or everything will be owned by the Postgres super user.

   To dump your fairly recent post-v6.0 database installation, type

   ```
   $ pg_dumpall -z > db.out
   ```

   To use the latest pg_dumpall script on your existing older database before upgrading Postgres, pull the most recent version of pg_dumpall from the new distribution:

   ```
   $ cd
   $ gunzip -c postgresql-v6.4.tar.gz \
       | tar xvf - src/bin/pg_dump/pg_dumpall
   $ chmod a+x src/bin/pg_dump/pg_dumpall
   $ src/bin/pg_dump/pg_dumpall -z > db.out
   $ rm -rf src
   ```

   If you wish to preserve object id's (oids), then use the -o option when running pg_dumpall. However, unless you have a special reason for doing this (such as using OIDs as keys in tables), don't do it.

   If the pg_dumpall command seems to take a long time and you think it might have died, then, from another terminal, type

   ```
   $ ls -l db.out
   ```

   several times to see if the size of the file is growing.

   Please note that if you are upgrading from a version prior to Postgres95 v1.09 then you must back up your database, install Postgres95 v1.09, restore your database, then back it up again. You should also read the release notes which should cover any release-specific issues.

   ## Caution

   You must make sure that your database is not updated in the middle of your backup. If necessary, bring down postmaster, edit the permissions in file `/usr/local/pgsql/data/pg_hba.conf` to allow only you on, then bring postmaster back up.

7. If you are upgrading an existing system then kill the postmaster. Type

   ```
   $ ps -ax | grep postmaster
   ```

   This should list the process numbers for a number of processes. Type the following line, with *pid* replaced by the process id for process `postmaster`. (Do not use the id for process "grep postmaster".) Type

   ```
   $ kill pid
   ```

   to actually stop the process.

   ## Tip

   On systems which have Postgres started at boot time, there is probably a startup file which will accomplish the same thing. For example, on my Linux system I can type

   ```
   $ /etc/rc.d/init.d/postgres.init stop
   ```

   to halt Postgres.

8. If you are upgrading an existing system then move the old directories out of the way. If you are short of disk space then you may have to back up and delete the directories instead. If you do this, save the old database in the `/usr/local/pgsql/data` directory tree. At a minimum, save file `/usr/local/pgsql/data/pg_hba.conf`.

   Type the following:

   ```
   $ su -
   $ cd /usr/src
   $ mv pgsql pgsql_6_0
   $ cd /usr/local
   $ mv pgsql pgsql_6_0
   $ exit
   ```

   If you are not using `/usr/local/pgsql/data` as your data directory (check to see if environment variable PGDATA is set to something else) then you will also want to move this directory in the same manner.

9. Make new source and install directories. The actual paths can be different for your installation but you must be consistant throughout this procedure.

   ## Note

   There are two places in this installation procedure where you will have an opportunity to specify installation locations for programs, libraries, documentation, and other files. Usually it is sufficient to specify these at the `make install` stage of installation.

   Type

   ```
   $ su
   $ cd /usr/src
   $ mkdir pgsql
   $ chown postgres:postgres pgsql
   $ cd /usr/local
   $ mkdir pgsql
   ```

```
$ chown postgres:postgres pgsql
$ exit
```

10. Unzip and untar the new source file. Type

```
$ cd /usr/src/pgsql
$ gunzip -c ~/postgresql-v6.4.tar.gz | tar xvf -
```

11. Configure the source code for your system. It is this step at which you can specify your actual installation path for the build process (see the --prefix option below). Type

```
$ cd /usr/src/pgsql/src
$ ./configure [ options ]
```

    a. (Optional) Among other chores, the configure script selects a system-specific "template" file from the files provided in the template subdirectory. If it cannot guess which one to use for your system, it will say so and exit. In that case you'll need to figure out which one to use and run configure again, this time giving the `--with-template=TEMPLATE` option to make the right file be chosen.

Please Report Problems

If your system is not automatically recognized by configure and you have to do this, please send email to scrappy@hub.org[3] with the output of the program ./config.guess. Indicate what the template file should be.

    b. (Optional) Choose configuration options. Check Configuration Options for details. However, for a plain-vanilla first installation with no extra options like multi-byte character support or locale collation support it may be adequate to have chosen the installation areas and to run configure without extra options specified. The configure script accepts many additional options that you can use if you don't like the default configuration. To see them all, type

```
./configure --help
```

Some of the more commonly used ones are:

```
      --prefix=BASEDIR   Selects a different base directory
for the
                         installation of the Postgres
configuration.
                         The default is /usr/local/pgsql.
      --with-template=TEMPLATE
                         Use template file TEMPLATE - the
template
                         files are assumed to be in the
directory
                         src/template, so look there for
proper values.
      --with-tcl         Build interface libraries and
programs requiring
                         Tcl/Tk, including libpgtcl, pgtclsh,
and pgtksh.
      --with-perl        Build the Perl interface library.
```

[3] mailto:scrappy@hub.org

```
      --with-odbc        Build the ODBC driver package.
      --enable-hba       Enables Host Based Authentication
(DEFAULT)
      --disable-hba      Disables Host Based Authentication
      --enable-locale    Enables USE_LOCALE
      --enable-cassert   Enables ASSERT_CHECKING
      --with-CC=compiler
                         Use a specific C compiler that the
configure
                         script cannot find.
      --with-CXX=compiler
      --without-CXX
                         Use a specific C++ compiler that the
configure
                         script cannot find, or exclude C++
compilation
                         altogether.   (This only affects
libpq++ at
    present.)
```

c. Here is the configure script used on a Sparc Solaris 2.5 system with `/opt/postgres` specified as the installation base directory:

```
$ ./configure --prefix=/opt/postgres \
    --with-template=sparc_solaris-gcc --with-pgport=5432 \
    --enable-hba --disable-locale
```

Tip

Of course, you may type these three lines all on the same line.

12. Install the man and HTML documentation. Type

```
$ cd /usr/src/pgsql/doc
$ gmake install
```

The documentation is also available in Postscript format. Look for files ending with `.ps.gz` in the same directory.

13. Compile the program. Type

```
$ cd /usr/src/pgsql/src
$ gmake all >& make.log &
$ tail -f make.log
```

The last line displayed will hopefully be

```
All of PostgreSQL is successfully made. Ready to install.
```

At this point, or earlier if you wish, type control-C to get out of tail. (If you have problems later on you may wish to examine file make.log for warning and error messages.)

Note

You will probably find a number of warning messages in make.log. Unless you have problems later on, these messages may be safely ignored.

If the compiler fails with a message stating that the flex command cannot be found then install flex as described earlier. Next, change directory back to this directory, type

```
$ make clean
```

then recompile again.

Compiler options, such as optimization and debugging, may be specified on the command line using the COPT variable. For example, typing

```
$ gmake COPT="-g" all >& make.log &
```

would invoke your compiler's `-g` option in all steps of the build. See `src/Makefile.global.in` for further details.

14. Install the program. Type

    ```
    $ cd /usr/src/pgsql/src
    $ gmake install >& make.install.log &
    $ tail -f make.install.log
    ```

    The last line displayed will be

    ```
    gmake[1]: Leaving directory `/usr/src/pgsql/src/man'
    ```

    At this point, or earlier if you wish, type control-C to get out of tail.

15. If necessary, tell your system how to find the new shared libraries. You can do *one* of the following, preferably the first:

    a. (Optional) As root, edit file `/etc/ld.so.conf`. Add a line

       ```
       /usr/local/pgsql/lib
       ```

       to the file. Then run command `/sbin/ldconfig`.

    b. (Optional) In a bash shell, type

       ```
       export LD_LIBRARY_PATH=/usr/local/pgsql/lib
       ```

    c. (Optional) In a csh shell, type

       ```
       setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
       ```

    Please note that the above commands may vary wildly for different operating systems. Check the platform specific notes, such as those for Ultrix4.x or and for non-ELF Linux.

    If, when you create the database, you get the message

    ```
    pg_id: can't load library 'libpq.so'
    ```

    then the above step was necessary. Simply do this step, then try to create the database again.

16. (Optional) If you used the `--with-perl` option to configure, check the install log to see whether the Perl module was actually installed. If you've followed our advice to make the Postgres files be owned by an unprivileged userid, then the Perl module won't have been installed, for lack of write privileges on the Perl library directories. You can complete its installation, either now or later, by becoming the user that does own the Perl library (often root) (via `su`) and doing

    ```
    $ cd /usr/src/pgsql/src/interfaces/perl5
    $ gmake install
    ```

17. If it has not already been done, then prepare account `postgres` for using Postgres. Any account that will use Postgres must be similarly prepared.

    There are several ways to influence the runtime environment of the Postgres server. Refer to the *Administrator's Guide* for more information.

    ## Note

    The following instructions are for a bash/sh shell. Adapt accordingly for other shells.

    a. Add the following lines to your login environment: shell, `~/.bash_profile`:

    ```
    PATH=$PATH:/usr/local/pgsql/bin
    MANPATH=$MANPATH:/usr/local/pgsql/man
    PGLIB=/usr/local/pgsql/lib
    PGDATA=/usr/local/pgsql/data
    export PATH MANPATH PGLIB PGDATA
    ```

    b. Several regression tests could failed if the user's locale collation scheme is different from that of standard C locale.

    If you configure and compile Postgres with the `--enable-locale` option then set locale environment to C (or unset all LC_* variables) by putting these additional lines to your login environment before starting postmaster:

    ```
    LC_COLLATE=C
    LC_CTYPE=C
    LC_COLLATE=C
    export LC_COLLATE LC_CTYPE LC_COLLATE
    ```

    c. Make sure that you have defined these variables before continuing with the remaining steps. The easiest way to do this is to type:

    ```
    $ source ~/.bash_profile
    ```

18. Create the database installation from your Postgres superuser account (typically account `postgres`). *Do not do the following as root!* This would be a major security hole. Type

    ```
    $ initdb
    ```

19. Set up permissions to access the database system. Do this by editing file `/usr/local/pgsql/data/pg_hba.conf`. The instructions are included in the file. (If your database is not located in the default location, i.e. if `PGDATA` is set to point elsewhere, then the location of this file will change accordingly.) This file should be made read only again once you are finished. If you are upgrading from v6.0 or later you can copy file `pg_hba.conf` from your old database on top of the one in your new database, rather than redoing the file from scratch.

20. Briefly test that the backend will start and run by running it from the command line.

    a. Start the postmaster daemon running in the background by typing

    ```
    $ cd
    $ postmaster -i
    ```

    b. Create a database by typing

    ```
    $ createdb
    ```

    c. Connect to the new database:

    ```
    $ psql
    ```

    d. And run a sample query:

    ```
    postgres=> SELECT datetime 'now';
    ```

    e. Exit psql:

    ```
    postgres=> \q
    ```

    f. Remove the test database (unless you will want to use it later for other tests):

    ```
    $ destroydb
    ```

21. Run postmaster in the background from your Postgres superuser account (typically account `postgres`). *Do not run postmaster from the root account!*

    Usually, you will want to modify your computer so that it will automatically start postmaster whenever it boots. It is not required; the Postgres server can be run successfully from non-privileged accounts without root intervention.

    Here are some suggestions on how to do this, contributed by various users.

    Whatever you do, postmaster must be run by the Postgres superuser (`postgres`?) *and not by root*. This is why all of the examples below start by switching user (su) to postgres. These commands also take into account the fact that environment variables like PATH and PGDATA may not be set properly. The examples are as follows. Use them with extreme caution.

    - If you are installing from a non-privileged account and have no root access, then start the postmaster and send it to the background:

      ```
      $ cd
      $ nohup postmaster > regress.log 2>&1 &
      ```

    - Edit file rc.local on NetBSD or file rc2.d on SPARC Solaris 2.5.1 to contain the following single line:

      ```
      su postgres -c "/usr/local/pgsql/bin/postmaster -S -D /usr/local/
      pgsql/data"
      ```

    - In FreeBSD 2.2-RELEASE edit /usr/local/etc/rc.d/pgsql.sh to contain the following lines and make it chmod 755 and chown root:bin.

      ```
      #!/bin/sh
      ```

```
[ -x /usr/local/pgsql/bin/postmaster ] && {
    su -l pgsql -c 'exec /usr/local/pgsql/bin/postmaster
        -D/usr/local/pgsql/data
        -S -o -F > /usr/local/pgsql/errlog' &
    echo -n ' pgsql'
}
```

You may put the line breaks as shown above. The shell is smart enough to keep parsing beyond end-of-line if there is an expression unfinished. The exec saves one layer of shell under the postmaster process so the parent is init.

- In RedHat Linux add a file `/etc/rc.d/init.d/postgres.init` which is based on the example in `contrib/linux/`. Then make a softlink to this file from `/etc/rc.d/rc5.d/S98postgres.init`.

- In RedHat Linux edit file /etc/inittab to add the following as a single line:

```
pg:2345:respawn:/bin/su - postgres -c
    "/usr/local/pgsql/bin/postmaster -D/usr/local/pgsql/data
    >> /usr/local/pgsql/server.log 2>&1 </dev/null"
```

(The author of this example says this example will revive the postmaster if it dies, but he doesn't know if there are other side effects.)

22. Run the regression tests. The file `/usr/src/pgsql/src/test/regress/README` has detailed instructions for running and interpreting the regression tests. A short version follows here:

    - Type

      ```
      $ cd /usr/src/pgsql/src/test/regress
      $ gmake clean
      $ gmake all runtest
      ```

      You do not need to type `gmake clean` if this is the first time you are running the tests.

      You should get on the screen (and also written to file `./regress.out`) a series of statements stating which tests passed and which tests failed. Please note that it can be normal for some tests to "fail" on some platforms. The script says a test has failed if there is any difference at all between the actual output of the test and the expected output. Thus, tests may "fail" due to minor differences in wording of error messages, small differences in floating-point roundoff, etc, between your system and the regression test reference platform. "Failures" of this type do not indicate a problem with Postgres. The file `./regression.diffs` contains the textual differences between the actual test output on your machine and the "expected" output (which is simply what the reference system produced). You should carefully examine each difference listed to see whether it appears to be a significant issue.

      For example,

      - For a i686/Linux-ELF platform, no tests failed since this is the v6.4 regression testing reference platform.

      - For the SPARC/Linux-ELF platform, using the 970525 beta version of Postgres v6.2 the following tests "failed": float8 and geometry "failed" due to minor precision differences in floating point numbers. select_views produces massively different output, but the differences are due to minor floating point differences.

Even if a test result clearly indicates a real failure, it may be a localized problem that will not affect you. An example is that the `int8` test will fail, producing obviously incorrect output, if your machine and C compiler do not provide a 64-bit integer data type (or if they do but configure didn't discover it). This is not something to worry about unless you need to store 64-bit integers.

Conclusion? If you do see failures, try to understand the nature of the differences and then decide if those differences will affect your intended use of Postgres. The regression tests are a helpful tool, but they may require some study to be useful.

After running the regression tests, type

```
$ destroydb regression
$ cd /usr/src/pgsql/src/test/regress
$ gmake clean
```

to recover the disk space used for the tests. (You may want to save the `regression.diffs` file in another place before doing this.)

23. If you haven't already done so, this would be a good time to modify your computer to do regular maintainence. The following should be done at regular intervals:

Minimal Backup Procedure

1. Run the SQL command `VACUUM`. This will clean up your database.
2. Back up your system. (You should probably keep the last few backups on hand.) Preferably, no one else should be using the system at the time.

Ideally, the above tasks should be done by a shell script that is run nightly or weekly by cron. Look at the man page for crontab for a starting point on how to do this. (If you do it, please e-mail us a copy of your shell script. We would like to set up our own systems to do this too.)

24. If you are upgrading an existing system then reinstall your old database. Type

```
$ cd
$ psql -e template1 < db.out
```

If your pre-v6.2 database uses either path or polygon geometric data types, then you will need to upgrade any columns containing those types. To do so, type (from within psql)

```
UPDATE FirstTable SET PathCol = UpgradePath(PathCol);
UPDATE SecondTable SET PathCol = UpgradePath(PathCol);
...
VACUUM;
```

UpgradePath() checks to see that a path value is consistant with the old syntax, and will not update a column which fails that examination. UpgradePoly() cannot verify that a polygon is in fact from an old syntax, but RevertPoly() is provided to reverse the effects of a mis-applied upgrade.

25. If you are a new user, you may wish to play with Postgres as described below.

26. Clean up after yourself. Type

```
$ rm -rf /usr/src/pgsql_6_0
$ rm -rf /usr/local/pgsql_6_0
# Also delete old database directory tree if it is not in
```

```
#  /usr/local/pgsql_6_0/data
$ rm ~/postgresql-v6.2.1.tar.gz
```

27. You will probably want to print out the documentation. If you have a Postscript printer, or have your machine already set up to accept Postscript files using a print filter, then to print the User's Guide simply type

    ```
    $ cd /usr/local/pgsql/doc
    $ gunzip user.ps.tz | lpr
    ```

    Here is how you might do it if you have Ghostscript on your system and are writing to a laserjet printer.

    ```
    $ alias gshp='gs -sDEVICE=laserjet -r300 -dNOPAUSE'
    $ export GS_LIB=/usr/share/ghostscript:/usr/share/ghostscript/
    fonts
    $ gunzip user.ps.gz
    $ gshp -sOUTPUTFILE=user.hp user.ps
    $ gzip user.ps
    $ lpr -l -s -r manpage.hp
    ```

28. The Postgres team wants to keep Postgres working on all of the supported platforms. We therefore ask you to let us know if you did or did not get Postgres to work on you system. Please send a mail message to pgsql-ports@postgresql.org[4] telling us the following:

    - The version of Postgres (v6.4, 6.3.2, beta 981014, etc.).
    - Your operating system (i.e. RedHat v5.1 Linux v2.0.34).
    - Your hardware (SPARC, i486, etc.).
    - Did you compile, install and run the regression tests cleanly? If not, what source code did you change (i.e. patches you applied, changes you made, etc.), what tests failed, etc. It is normal to get many warning when you compile. You do not need to report these.

29. Now create, access and manipulate databases as desired. Write client programs to access the database server. In other words, *enjoy*!

24.3. Playing with Postgres

After Postgres is installed, a database system is created, a postmaster daemon is running, and the regression tests have passed, you'll want to see Postgres do something. That's easy. Invoke the interactive interface to Postgres, psql:

```
% psql template1
```

(psql has to open a particular database, but at this point the only one that exists is the template1 database, which always exists. We will connect to it only long enough to create another one and switch to it.)

The response from psql is:

```
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
```

[4] mailto:pgsql-ports@postgresql.org

```
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: template1

template1=>
```

Create the database foo:

```
template1=> create database foo;
CREATEDB
```

(Get in the habit of including those SQL semicolons. Psql won't execute anything until it sees the semicolon or a "\g" and the semicolon is required to delimit multiple statements.)

Now connect to the new database:

```
template1=> \c foo
connecting to new database: foo
```

("slash" commands aren't SQL, so no semicolon. Use \? to see all the slash commands.)

And create a table:

```
foo=> create table bar (i int4, c char(16));
CREATE
```

Then inspect the new table:

```
foo=> \d bar

Table    = bar
+----------------------------------+----------------------------------
+-------+
|               Field              |              Type
 | Length|
+----------------------------------+----------------------------------
+-------+
| i                                | int4
 |     4 |
| c                                | (bp)char
 |    16 |
+----------------------------------+----------------------------------
+-------+
```

And so on. You get the idea.

24.4. The Next Step

Questions? Bugs? Feedback? First, read the files in directory `/usr/src/pgsql/doc/`. The FAQ in this directory may be particularly useful.

If Postgres failed to compile on your computer then fill out the form in file `/usr/src/pgsql/doc/bug.template` and mail it to the location indicated at the top of the form.

Check on the web site at http://www.postgresql.org For more information on the various support mailing lists.

24.5. Porting Notes

Note

Check for any platform-specific FAQs in the `doc/` directory of the source distribution. For some ports, the notes below may be out of date.

24.5.1. Ultrix4.x

Note

There have been no recent reports of Ultrix usage with Postgres.

You need to install the libdl-1.1 package since Ultrix 4.x doesn't have a dynamic loader. It's available in s2k-ftp.CS.Berkeley.EDU:pub/personal/andrew/libdl-1.1.tar.Z

24.5.2. Linux

24.5.2.1. Linux ELF

Thomas G. Lockhart

The regression test reference machine is a linux-2.0.30/libc-5.3.12/RedHat-4.2 installation running on a dual processor i686. The linux-elf port installs cleanly. See the Linux FAQ for more details.

24.5.2.2. Linux a.out

For non-ELF Linux, the dld library MUST be obtained and installed on the system. It enables dynamic link loading capability to the Postgres port. The dld library can be obtained from the sunsite linux distributions. The current name is dld-3.2.5. Jalon Q. Zimmerman[5]

24.5.3. BSD/OS

For BSD/OS 2.0 and 2.01, you will need to get the GNU dld library.

24.5.4. NeXT

The NeXT port for v1.09 was supplied by Tom R. Hageman[6]. It requires a SysV IPC emulation library and header files for shared libary and semaphore stuff. Tom just happens to sell such a product so contact him for information. He has also indicated that binary releases of Postgres for NEXTSTEP will be made available to the general public. Contact Info@RnA.nl for information.

We have no recent reports of successful NeXT installations (as of v6.2.1). However, the client-side libraries should work even if the backend is not supported.

[5] sneaker@powergrid.electriciti.com
[6] mailto:tom@basil.icce.rug.nl

Chapter 25. Runtime Environment

This chapter outlines the interaction between Postgres and the operating system.

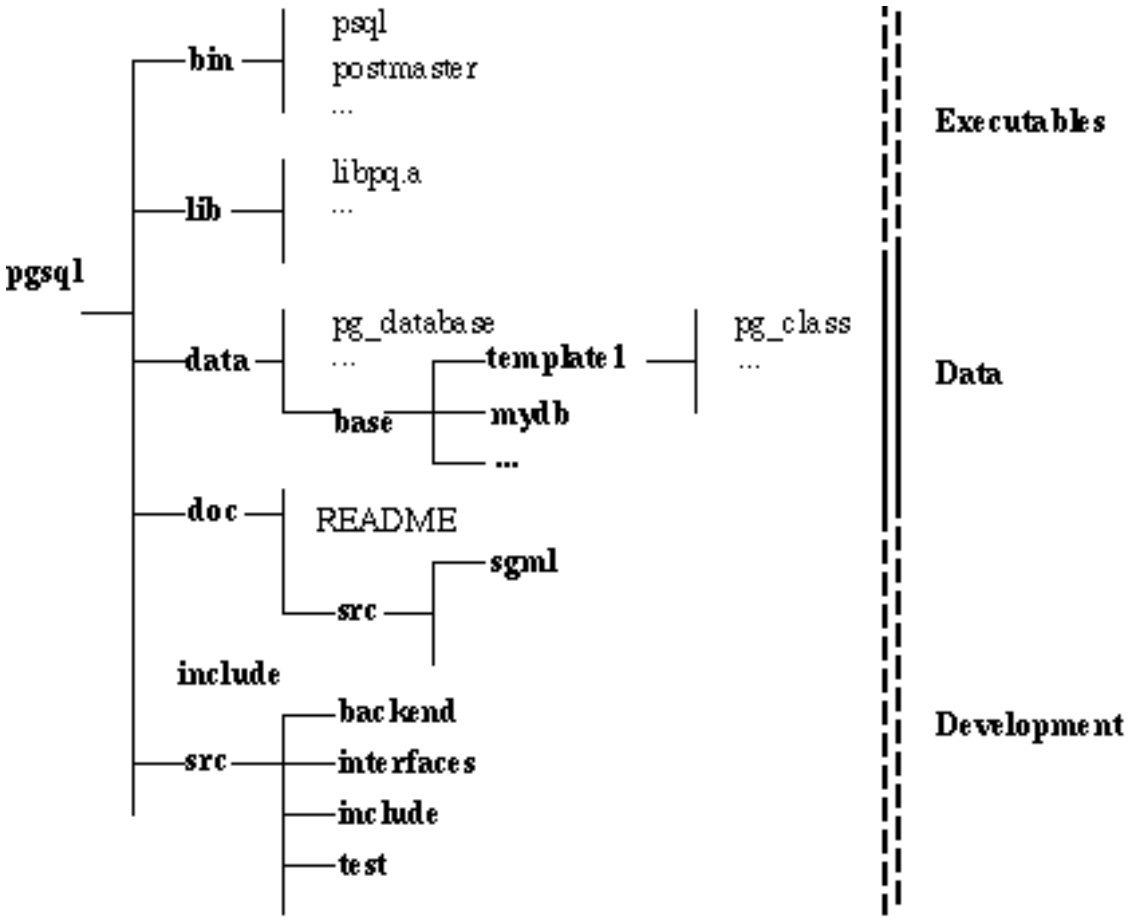
25.1. Using Postgres from Unix

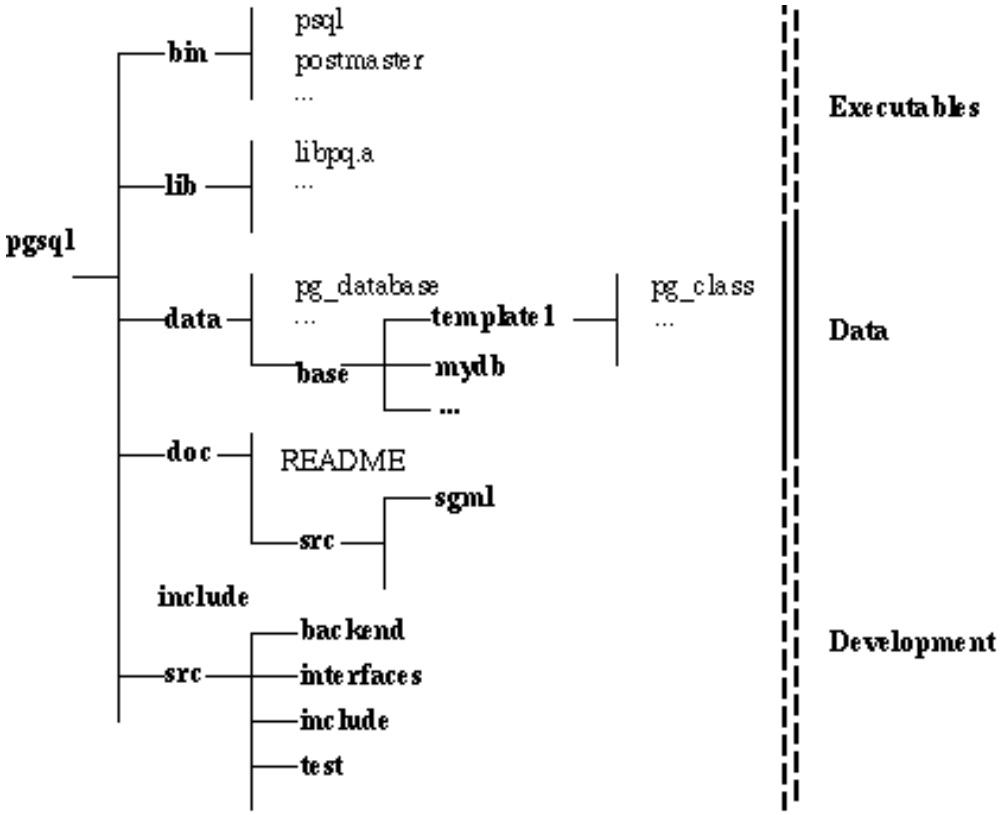
All Postgres commands that are executed directly from a Unix shell are found in the directory ".../bin". Including this directory in your search path will make executing the commands easier.

A collection of system catalogs exist at each site. These include a class (`pg_user`) that contains an instance for each valid Postgres user. The instance specifies a set of Postgres privileges, such as the ability to act as Postgres super-user, the ability to create/destroy databases, and the ability to update the system catalogs. A Unix user cannot do anything with Postgres until an appropriate instance is installed in this class. Further information on the system catalogs is available by running queries on the appropriate classes.

Chapter 26. System Layout

Figure 26.1. Postgres file layout





shows how the Postgres distribution is laid out when installed in the default way. For simplicity, we will assume that Postgres has been installed in the directory `/usr/local/pgsql`. Therefore, wherever you see the directory `/usr/local/pgsql` you should substitute the name of the directory where Postgres is actually installed. All Postgres commands are installed in the directory `/usr/local/pgsql/bin`. Therefore, you should add this directory to your shell command path. If you use a variant of the Berkeley C shell, such as csh or tcsh, you would add

```
set path = ( /usr/local/pgsql/bin path )
```

in the .login file in your home directory. If you use a variant of the Bourne shell, such as sh, ksh, or bash, then you would add

```
PATH=/usr/local/pgsql/bin PATH
export PATH
```

to the .profile file in your home directory. From now on, we will assume that you have added the Postgres bin directory to your path. In addition, we will make frequent reference to "setting a shell variable" or "setting an environment variable" throughout this document. If you did not fully understand the last paragraph on modifying your search path, you should consult the UNIX manual pages that describe your shell before going any further.

If you have not set things up in the default way, you may have some more work to do. For example, if the database server machine is a remote machine, you will need to set the `PGHOST` environment variable to the name of the database server machine. The environment variable `PGPORT` may also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the postmaster, you must go back and make sure that your environment is properly set up.

Chapter 27. Using pg_options

Massimo Dal Zotto

Note

Contributed by Massimo Dal Zotto[1]

The optional file `data/pg_options` contains runtime options used by the backend to control trace messages and other backend tunable parameters. What makes this file interesting is the fact that it is re-read by a backend when it receives a SIGHUP signal, making thus possible to change run-time options on the fly without needing to restart Postgres. The options specified in this file may be debugging flags used by the trace package (`backend/utils/misc/trace.c`) or numeric parameters which can be used by the backend to control its behaviour. New options and parameters must be defined in `backend/utils/misc/trace.c` and `backend/include/utils/trace.h`.

For example suppose we want to add conditional trace messages and a tunable numeric parameter to the code in file `foo.c`. All we need to do is to add the constant TRACE_FOO and OPT_FOO_PARAM into `backend/include/utils/trace.h`:

```
/* file trace.h */
enum pg_option_enum {
    ...
    TRACE_FOO,   /* trace foo functions */
    OPT_FOO_PARAM,  /* foo tunable parameter */

    NUM_PG_OPTIONS              /* must be the last item of enum */
};
```

and a corresponding line in `backend/utils/misc/trace.c`:

```
/* file trace.c */
static char *opt_names[] = {
    ...
    "foo",              /* trace foo functions */
    "fooparam"           /* foo tunable parameter */
};
```

Options in the two files must be specified in exactly the same order. In the foo source files we can now reference the new flags with:

```
/* file foo.c */
#include "trace.h"
#define foo_param pg_options[OPT_FOO_PARAM]

int
foo_function(int x, int y)
{
    TPRINTF(TRACE_FOO, "entering foo_function, foo_param=%d",
 foo_param);
    if (foo_param > 10) {
        do_more_foo(x, y);
```

[1] mailto:dz@cs.unitn.it

```
    }
}
```

Existing files using private trace flags can be changed by simply adding the following code:

```
#include "trace.h"
/* int my_own_flag = 0; -- removed */
#define my_own_flag pg_options[OPT_MY_OWN_FLAG]
```

All pg_options are initialized to zero at backend startup. If we need a different default value we must add some initialization code at the beginning of `PostgresMain`. Now we can set the foo_param and enable foo trace by writing values into the `data/pg_options` file:

```
# file pg_options
...
foo=1
fooparam=17
```

The new options will be read by all new backends when they are started. To make effective the changes for all running backends we need to send a SIGHUP to the postmaster. The signal will be automatically sent to all the backends. We can also activate the changes only for a specific backend by sending the SIGHUP directly to it.

pg_options can also be specified with the `-T` switch of Postgres:

```
postgres options -T "verbose=2,query,hostlookup-"
```

The functions used for printing errors and debug messages can now make use of the *syslog(2)* facility. Message printed to stdout or stderr are prefixed by a timestamp containing also the backend pid:

```
#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

This format improves readability of the logs and allows people to understand exactly which backend is doing what and at which time. It also makes easier to write simple awk or perl scripts which monitor the log to detect database errors or problem, or to compute transaction time statistics.

Messages printed to syslog use the log facility LOG_LOCAL0. The use of syslog can be controlled with the syslog pg_option. Unfortunately many functions call directly `printf()` to print their messages to stdout or stderr and this output can't be redirected to syslog or have timestamps in it. It would be advisable that all calls to printf would be replaced with the PRINTF macro and output to stderr be changed to use EPRINTF instead so that we can control all output in a uniform way.

The new pg_options mechanism is more convenient than defining new backend option switches because:

- we don't have to define a different switch for each thing we want to control. All options are defined as keywords in an external file stored in the data directory.
- we don't have to restart Postgres to change the setting of some option. Normally backend options are specified to the postmaster and passed to each backend when it is started. Now they are read from a file.

- we can change options on the fly while a backend is running. We can thus investigate some problem by activating debug messages only when the problem appears. We can also try different values for tunable parameters.

The format of the `pg_options` file is as follows:

```
# comment
option=integer_value  # set value for option
option                # set option = 1
option+               # set option = 1
option-               # set option = 0
```

Note that *keyword* can also be an abbreviation of the option name defined in `backend/utils/misc/trace.c`.

The options currently defined in `backend/utils/misc/trace.c` are the following:

all

Global trace flag. Allowed values are:

0

Trace messages enabled individually

1

Enable all trace messages

-1

Disable all trace messages

verbose

Verbosity flag. Allowed values are:

0

No messages. This is the default.

1

Print information messages.

2

Print more information messages.

query

Query trace flag. Allowed values are:

0

Don't print query.

1

Print a condensed query in one line.

4

Print the full query.

plan

Print query plan.

parse

Print parser output.

rewritten

Print rewritten query.

parserstats

Print parser statistics.

plannerstats

Print planner statistics.

executorstats

Print executor statistics.

shortlocks

Currently unused but needed to enable features in the future.

locks

Trace locks.

userlocks

Trace user locks.

spinlocks

Trace spin locks.

notify

Trace notify functions.

malloc

Currently unused.

palloc

Currently unused.

lock_debug_oidmin

Minimum relation oid traced by locks.

lock_debug_relid

oid, if not zero, of relation traced by locks.

lock_read_priority

Currently unused.

deadlock_timeout

Deadlock check timer.

syslog

syslog flag. Allowed values are:

0

Messages to stdout/stderr.

1

Messages to stdout/stderr and syslog.

2

Messages only to syslog.

hostlookup

Enable hostname lookup in ps_status.

showportnumber

Show port number in ps_status.

notifyunlock

Unlock of pg_listener after notify.

notifyhack

Remove duplicate tuples from pg_listener.

For example my pg_options file contains the following values:

```
verbose=2
query
hostlookup
showportnumber
```

Some of the existing code using private variables and option switches has been changed to make use of the pg_options feature, mainly in `postgres.c`. It would be advisable to modify all existing code in this way, so that we can get rid of many of the switches on the Postgres command line and can have more tunable options with a unique place to put option values.

Chapter 28. Starting postmaster

Nothing can happen to a database unless the postmaster process is running. As the site administrator, there are a number of things you should remember before starting the postmaster. These are discussed in the installation and configuration sections of this manual. However, if Postgres has been installed by following the installation instructions exactly as written, the following simple command is all you should need to start the postmaster:

```
% postmaster
```

The postmaster occasionally prints out messages which are often helpful during troubleshooting. If you wish to view debugging messages from the postmaster, you can start it with the -d option and redirect the output to the log file:

```
% postmaster -d >& pm.log &
```

If you do not wish to see these messages, you can type

```
% postmaster -S
```

and the postmaster will be "S"ilent. Notice that there is no ampersand ("&") at the end of the last example.

Chapter 29. Adding and Deleting Users

createuser enables specific users to access Postgres. destroyuser removes users and prevents them from accessing Postgres. Note that these commands only affect users with respect to Postgres; they have no effect on users other privileges or status with regards to the underlying operating system.

Chapter 30. Disk Management

30.1. Alternate Locations

It is possible to create a database in a location other than the default location for the installation. Remember that all database access actually occurs through the database backend, so that any location specified must be accessible by the backend.

Either an absolute path name or an environment variable may be specified as a location. Note that for security and integrity reasons, all paths and environment variables so specified have some additional path fields appended.

Note

The environment variable style of specification is to be preferred since it allows the site administrator more flexibility in managing disk storage.

Remember that database creation is actually performed by the database backend. Therefore, any environment variable specifying an alternate location must have been defined before the backend was started. To define an alternate location PGDATA2 pointing to `/home/postgres/data`, type

```
% setenv PGDATA2 /home/postgres/data
```

Usually, you will want to define this variable in the Postgres superuser's `.profile` or `.cshrc` initialization file to ensure that it is defined upon system startup.

To create a data storage area in `/home/postgres/data`, ensure that `/home/postgres` already exists and is writable. From the command line, type

```
% initlocation $PGDATA2
Creating Postgres database system directory /home/postgres/data

Creating Postgres database system directory /home/postgres/data/base
```

To test the new location, create a database test by typing

```
% createdb -D PGDATA2 test
% destroydb test
```

Chapter 31. Troubleshooting

Assuming that your site administrator has properly started the postmaster process and authorized you to use the database, you (as a user) may begin to start up applications. As previously mentioned, you should add `/usr/local/pgsql/bin` to your shell search path. In most cases, this is all you should have to do in terms of preparation.

If you get the following error message from a Postgres command (such as psql or createdb):

```
connectDB() failed: Is the postmaster running at 'localhost' on port
 '5432'?
```

it is usually because either the postmaster is not running, or you are attempting to connect to the wrong server host. If you get the following error message:

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)
```

it means that the site administrator started the postmaster as the wrong user. Tell him to restart it as the Postgres superuser.

Chapter 32. Managing a Database

Now that Postgres is up and running we can create some databases to experiment with. Here, we describe the basic commands for managing a database.

32.1. Creating a Database

Let's say you want to create a database named mydb. You can do this with the following command:

```
% createdb mydb
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 16 characters in length. Not every user has authorization to become a database administrator. If Postgres refuses to create databases for you, then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs.

32.2. Accessing a Database

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

- running the Postgres terminal monitor program (psql) which allows you to interactively enter, edit, and execute SQL commands.
- writing a C program using the `libpq` subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in the *PostgreSQL Programmer's Guide*.

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

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to the Postgres interactive sql monitor:

  type \? for help on slash commands
  type \q to quit
  type \g or terminate with semicolon to execute query
You are currently connected to the database: mydb

mydb=>
```

This prompt indicates that the terminal monitor is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, "\". For example, you can get help on the syntax of various Postgres SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the backslash-g is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say myFile, instead of entering them interactively, type:

```
mydb=> \i fileName
```

To get out of psql and return to UNIX, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type backslash-h at the monitor prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by "--". Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by "/* ... */"

32.3. Destroying a Database

If you are the database administrator for the database mydb, you can destroy it using the following UNIX command:

```
% destroydb mydb
```

This action physically removes all of the UNIX files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 33. Database Recovery

This section needs to be written. Volunteers?

Chapter 34. Regression Test

Regression test instructions and analysis.

The PostgreSQL regression tests are a comprehensive set of tests for the SQL implementation embedded in PostgreSQL developed by Jolly Chen and Andrew Yu. It tests standard SQL operations as well as the extended capabilities of PostgreSQL.

These tests have recently been revised by Marc Fournier and Thomas Lockhart and are now packaged as functional units which should make them easier to run and easier to interpret. From PostgreSQL v6.1 onward the regression tests are current for every official release.

Some properly installed and fully functional PostgreSQL installations can fail some of these regression tests due to artifacts of floating point representation and time zone support. The current tests are evaluated using a simple "diff" algorithm, and are sensitive to small system differences. For apparently failed tests, examining the differences may reveal that the differences are not significant.

The regression testing notes below assume the following (except where noted):

- Commands are Unix-compatible. See note below.
- Defaults are used except where noted.
- User postgres is the Postgres superuser.
- The source path is /usr/src/pgsql (other paths are possible).
- The runtime path is /usr/local/pgsql (other paths are possible).

34.1. Regression Environment

The regression test is invoked by the `make` command which compiles a C program into a shared library in the current directory. Localized shell scripts are also created in the current directory. The output file templates are massaged into the `./expected/*.out` files. The localization replaces macros in the source files with absolute pathnames and user names.

Normally, the regression test should be run as the pg_superuser since the 'src/test/regress' directory and sub-directories are owned by the pg_superuser. If you run the regression test as another user the 'src/test/regress' directory tree should be writeable to that user.

The postmaster should be invoked with the system time zone set for Berkeley, California. This is done automatically by the regression test script. However, it does require machine support for the PST8PDT time zone.

To verify that your machine does have this support, type the following:

```
setenv TZ PST8PDT
date
```

The "date" command above should have returned the current system time in the PST8PDT time zone. If the PST8PDT database is not available, then your system may have returned the time in GMT. If the PST8PDT time zone is not available, you can set the time zone rules explicitly:

```
setenv PGTZ PST8PDT7,M04.01.0,M10.05.03
```

34.2. Directory Layout

Note

This should become a table in the previous section.

```
 input/ .... .source files that are converted using 'make all' into
             some of the .sql files in the 'sql' subdirectory

 output/ ... .source files that are converted using 'make all' into
             .out files in the 'expected' subdirectory

 sql/ ...... .sql files used to perform the regression tests

 expected/ . .out files that represent what we *expect* the results
to
             look like

 results/ .. .out files that represent what the results *actually*
look
             like. Also used as temporary storage for table copy
testing.
```

34.3. Regression Test Procedure

Commands were tested on RedHat Linux version 4.2 using the bash shell. Except where noted, they will probably work on most systems. Commands like `ps` and `tar` vary wildly on what options you should use on each platform. *Use common sense* before typing in these commands.

Postgres Regression Configuration

For a fresh install or upgrading from previous releases of Postgres:

1. Build the regression test. Type

   ```
   cd /usr/src/pgsql/src/test/regress
   gmake all
   ```

2. (Optional) If you have prevously invoked the regression test, clean up the working directory with:

   ```
   cd /usr/src/pgsql/src/test/regress
   make clean
   ```

3. The file /usr/src/pgsql/src/test/regress/README has detailed instructions for running and interpreting the regression tests. A short version follows here:

   If the postmaster is not already running, start the postmaster on an available window by typing

   ```
   postmaster
   ```

   or start the postmaster daemon running in the background by typing

   ```
   cd
   nohup postmaster > regress.log 2>&1 &
   ```

Run postmaster from your Postgres super user account (typically account postgres).

Note

Do not run `postmaster` from the root account.

4. Run the regression tests. Type

```
cd /usr/src/pgsql/src/test/regress
gmake runtest
```

You do not need to type "gmake clean" if this is the first time you are running the tests.

5. You should get on the screen (and also written to file ./regress.out) a series of statements stating which tests passed and which tests failed. Please note that it can be normal for some of the tests to "fail". For the failed tests, use diff to compare the files in directories ./results and ./expected. If float8 failed, type something like:

```
cd /usr/src/pgsql/src/test/regress
diff -w expected/float8.out results
```

6. After running the tests, type

```
destroydb regression
cd /usr/src/pgsql/src/test/regress
gmake clean
```

34.4. Regression Analysis

“Failed” tests may have failed due to slightly different error messages, math libraries, or output formatting. "Failures" of this type do not indicate a problem with Postgres.

For a i686/Linux-ELF platform, no tests failed since this is the v6.2.1 regression testing reference platform.

For the SPARC/Linux-ELF platform, using the 970525 beta version of Postgres v6.2 the following tests "failed": float8 and geometry "failed" due to minor precision differences in floating point numbers. select_views produces massively different output, but the differences are due to minor floating point differences.

Conclusion? If you do see failures, try to understand the nature of the differences and then decide if those differences will affect your intended use of Postgres. However, keep in mind that this is likely to be the most solid release of Postgres to date, incorporating many bug fixes from v6.1, and that previous versions of Postgres have been in use successfully for some time now.

34.4.1. Comparing expected/actual output

The results are in files in the ./results directory. These results can be compared with results in the ./expected directory using 'diff'. The files might not compare exactly. The following paragraphs attempt to explain the differences.

34.4.2. Error message differences

Some of the regression tests involve intentional invalid input values. Error messages can come from either the Postgres code or from the host platform system routines. In the latter case, the messages may vary

between platforms, but should reflect similar information. These differences in messages will result in a "failed" regression test which can be validated by inspection.

34.4.3. OID differences

There are several places where PostgreSQL OID (object identifiers) appear in 'regress.out'. OID's are unique 32-bit integers which are generated by the PostgreSQL backend whenever a table row is inserted or updated. If you run the regression test on a non-virgin database or run it multiple times, the OID's reported will have different values. The following SQL statements in 'misc.out' have shown this behavior: QUERY: SELECT user_relns() AS user_relns ORDER BY user_relns; The 'a,523676' row is composed from an OID.

34.4.4. Date and time differences

On many supported platforms, you can force PostgreSQL to believe that it is running in the same time zone as Berkeley, California. See details in the section on how to run the regression tests. If you do not explicitly set your time zone environment to PST8PDT, then most of the date and time results will reflect your local time zone and will fail the regression testing. There appears to be some systems which do not accept the recommended syntax for explicitly setting the local time zone rules. Some systems using the public domain time zone package exhibit minor problems with pre-1970 PDT times, representing them in PST instead.

34.4.5. Floating point differences

Some of the tests involve computing 64-bit (`float8`) number from table columns. Differences in results involving mathematical functions of `float8` columns have been observed. These differences occur where different operating systems are used on the same platform ie: BSDI and SOLARIS on Intel/86, and where the same operating system is used used on different platforms, ie: SOLARIS on SPARC and Intel/86. Human eyeball comparison is needed to determine the real significance of these differences which are usually 10 places to the right of the decimal point. Some systems signal errors from pow() and exp() differently from the mechanism expected by the current Postgres code.

34.4.6. Polygon differences

Several of the tests involve operations on geographic date about the Oakland/Berkley CA street map. The map data is expressed as polygons whose vertices are represented as pairs of `float8` numbers (decimal latitude and longitude). Initially, some tables are created and loaded with geographic data, then some views are created which join two tables using the polygon intersection operator (##), then a select is done on the view. When comparing the results from different platforms, differences occur in the 2nd or 3rd place to the right of the decimal point. The SQL statements where these problems occur are the folowing:

```
QUERY: SELECT * from street;
QUERY: SELECT * from iexit;
```

34.4.7. Random differences

There is at least one test case in random.out which is intended to produce random results. This causes random to fail the regression testing. Typing

```
diff results/random.out expected/random.out
```

should produce only one or a few lines of differences for this reason, but other floating point differences on dissimilar architectures might cause many more differences. See the release notes below.

34.4.8. The “expected” files

The `./expected/*.out` files were adapted from the original monolithic `expected.input` file provided by Jolly Chen et al. Newer versions of these files generated on various development machines have been substituted after careful (?) inspection. Many of the development machines are running a Unix OS variant (FreeBSD, Linux, etc) on Ix86 hardware. The original `expected.input` file was created on a SPARC Solaris 2.4 system using the `postgres5-1.02a5.tar.gz` source tree. It was compared with a file created on an I386 Solaris 2.4 system and the differences were only in the floating point polygons in the 3rd digit to the right of the decimal point. (see below) The original `sample.regress.out` file was from the postgres-1.01 release constructed by Jolly Chen and is included here for reference. It may have been created on a DEC ALPHA machine as the `Makefile.global` in the postgres-1.01 release has PORTNAME=alpha.

Chapter 35. Release Notes

35.1. Release 6.4

There are *many* new features and improvements in this release. Thanks to our developers and maintainers, nearly every aspect of the system has received some attention since the previous release. Here is a brief, incomplete summary:

- Views and rules are now functional thanks to extensive new code in the rewrite rules system from Jan Wieck. He also wrote a chapter on it for the *Programmer's Guide*.
- Jan also contributed a second procedural language, PL/pgSQL, to go with the original PL/pgTCL procedural language he contributed last release.
- We have optional multiple-byte character set support from Tatsuo Iishi to complement our existing locale support.
- Client/server communications has been cleaned up, with better support for asynchronous messages and interrupts thanks to Tom Lane.
- The parser will now perform automatic type coersion to match arguments to available operators and functions, and to match columns and expressions with target columns. This uses a generic mechanism which supports the type extensibility features of Postgres. There is a new chapter in the *User's Guide* which covers this topic.
- Three new data types have been added. Two types, `inet` and `cidr`, support various forms of IP network, subnet, and machine addressing. There is now an 8-byte integer type available on some platforms. See the chapter on data types in the *User's Guide* for details. A fourth type, `serial`, is now supported by the parser as an amalgam of the `int4` type, a sequence, and a unique index.
- Several more SQL92-compatible syntax features have been added, including `INSERT DEFAULT VALUES`
- The automatic configuration and installation system has received some attention, and should be more robust for more platforms than it has ever been.

35.1.1. Migration to v6.4

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of Postgres.

35.1.2. Detailed Change List

```
Bug Fixes
---------
Fix for a tiny memory leak in PQsetdb/PQfinish(Bryan)
Remove char2-16 data types, use char/varchar(Darren)
Pqfn not handles a NOTICE message(Anders)
Reduced busywaiting overhead for spinlocks with many backends (dg)
Stuck spinlock detection (dg)
Fix up "ISO-style" timespan decoding and encoding(Thomas)
Fix problem with table drop after rollback of transaction(Vadim)
Change error message and remove non-functional update message(Vadim)
Fix for COPY array checking
Fix for SELECT 1 UNION SELECT NULL
Fix for buffer leaks in large object calls(Pascal)
Change owner from oid to int4 type(Bruce)
```

```
Fix a bug in the oracle compatibility functions btrim() ltrim() and
 rtrim()
Fix for shared invalidation cache overflow(Massimo)
Prevent file descriptor leaks in failed COPY's(Bruce)
Fix memory leak in libpgtcl's pg_select(Constantin)
Fix problems with username/passwords over 8 characters(Tom)
Fix problems with handling of asynchronous NOTIFY in backend(Tom)
Fix of many bad system table entries(Tom)

Enhancements
------------
Upgrade ecpg and ecpglib,see src/interfaces/ecpc/ChangeLog(Michael)
Show the index used in an EXPLAIN(Zeugswetter)
EXPLAIN  invokes  rule system and shows plan(s) for rewritten
 queries(Jan)
Multi-byte awareness of many data types and functions, via
 configure(Tatsuo)
New configure --with-mb option(Tatsuo)
New initdb --pgencoding option(Tatsuo)
New createdb -E multibyte option(Tatsuo)
Select version(); now returns PostgreSQL version(Jeroen)
Libpq now allows asynchronous clients(Tom)
Allow cancel from client of backend query(Tom)
Psql now cancels query with Control-C(Tom)
Libpq users need not issue dummy queries to get NOTIFY messages(Tom)
NOTIFY now sends sender's PID, so you can tell whether it was your
 own(Tom)
PGresult struct now includes associated error message, if any(Tom)
Define "tz_hour" and "tz_minute" arguments to date_part()(Thomas)
Add routines to convert between varchar and bpchar(Thomas)
Add routines to allow sizing of varchar and bpchar into target
 columns(Thomas)
Add bit flags to support timezonehour and minute in data
 retrieval(Thomas)
Allow more variations on valid floating point numbers (e.g. ".1",
 "1e6")(Thomas)
Fixes for unary minus parsing with leading spaces(Thomas)
Implement TIMEZONE_HOUR, TIMEZONE_MINUTE per SQL92 specs(Thomas)
Check for and properly ignore FOREIGN KEY column constraints(Thomas)
Define USER as synonym for CURRENT_USER per SQL92 specs(Thomas)
Enable HAVING clause but no fixes elsewhere yet.
Make "char" type a synonym for "char(1)" (actually implemented as
 bpchar)(Thomas)
Save string type if specified for DEFAULT clause handling(Thomas)
Coerce operations involving different data types(Thomas)
Allow some index use for columns of different types(Thomas)
Add capabilities for automatic type conversion(Thomas)
Cleanups for large objects, so file is truncated on open(Peter)
Readline cleanups(Tom)
Allow psql  \f \ to make spaces as delimiter(Bruce)
Pass pg_attribute.atttypmod to the frontend for column field
 lengths(Tom,Bruce)
Msql compatibility library in /contrib(Aldrin)
Remove the requirement that ORDER/GROUP BY clause identifiers be
```

```
included in the target list(David)
Convert columns to match columns in UNION clauses(Thomas)
Remove fork()/exec() and only do fork()(Bruce)
Jdbc cleanups(Peter)
Show backend status on ps command line(only works on some platforms)
(Bruce)
Pg_hba.conf now has a sameuser option in the database field
Make lo_unlink take oid param, not int4
New DISABLE_COMPLEX_MACRO for compilers that can't handle our
 macros(Bruce)
Libpgtcl now handles NOTIFY as a Tcl event, need not send dummy
 queries(Tom)
libpgtcl cleanups(Tom)
Add -error option to libpgtcl's pg_result command(Tom)
New locale patch, see docs/README/locale(Oleg)
Fix for pg_dump so CONSTRAINT and CHECK syntax is correct(ccb)
New contrib/lo code for large object orphan removal(Peter)
New psql command "SET CLIENT_ENCODING TO 'encoding'" for multi-bytes
feature, see /doc/README.mb(Tatsuo)
/contrib/noupdate code to revoke update permission on a column
Libpq can now be compiled on win32(Magnus)
Add PQsetdbLogin() in libpq
New 8-byte integer type, checked by configure for OS support(Thomas)
Better support for quoted table/column names(Thomas)
Surround table and column names with double-quotes in pg_dump(Thomas)
PQreset() now works with passwords(Tom)
Handle case of GROUP BY target list column number out of range(David)
Allow UNION in subselects
Add auto-size to screen to \d? commands(Bruce)
Use UNION to show all \d? results in one query(Bruce)
Add \d? field search feature(Bruce)
Pg_dump issues fewer \connect requests(Tom)
Make pg_dump -z flag work better, document it in manual page(Tom)
Add HAVING clause with full support for subselects and unions(Stephan)
Full text indexing routines in contrib/fulltextindex(Maarten)
Transaction ids now stored in shared memory(Vadim)
New PGCLIENTENCODING when issuing COPY command(Tatsuo)
Support for SQL92 syntax "SET NAMES"(Tatsuo)
Support for LATIN2-5(Tatsuo)
Add UNICODE regression test case(Tatsuo)
Lock manager cleanup, new locking modes for LLL(Vadim)
Allow index use with OR clauses(Bruce)
Allows "SELECT NULL ORDER BY 1;"
Explain VERBOSE prints the plan, and now pretty-prints the plan to
the postmaster log file(Bruce)
Add Indices display to \d command(Bruce)
Allow GROUP BY on functions(David)
New pg_class.relkind for large objects(Bruce)
New way to send libpq NOTICE messages to a different location(Tom)
New \w write command to psql(Bruce)
New /contrib/findoidjoins scans oid columns to find join
 relationships(Bruce)
Allow binary-compatible indices to be considered when checking for
 valid
```

```
indices for restriction clauses containing a constant(Thomas)
New ISBN/ISSN code in /contrib/isbn_issn
Allow NOT LIKE, IN, NOT IN, BETWEEN, and NOT BETWEEN
 constraint(Thomas)
New rewrite system fixes many problems with rules and views(Jan)
 * Rules on relations work
 * Event qualifications on insert/update/delete work
 * New OLD variable to reference CURRENT, CURRENT will be remove in
 future
 * Update rules can reference NEW and OLD in rule qualifications/
actions
 * Insert/update/delete rules on views work
 * Multiple rule actions are now supported, surrounded by parentheses
 * Regular users can create views/rules on tables they have RULE
 permits
 * Rules and views inherit the permissions on the creator
 * No rules at the column level
 * No UPDATE NEW/OLD rules
 * New pg_tables, pg_indexes, pg_rules and pg_views system views
 * Only a single action on SELECT rules
 * Total rewrite overhaul, perhaps for 6.5
 * handle subselects
 * handle aggregates on views
 * handle insert into select from view works
System indexes are now multi-key(Bruce)
Oidint2, oidint4, and oidname types are removed(Bruce)
Use system cache for more system table lookups(Bruce)
New backend programming language PL/pgSQL in backend/pl(Jan)
New SERIAL data type, auto-creates sequence/index(Thomas)
Enable assert checking without a recompile(Massimo)
User lock enhancements(Massimo)
New setval() command to set sequence value(Massimo)
Auto-remove unix socket file on startup if no postmaster
 running(Massimo)
Conditional trace package(Massimo)
New UNLISTEN command(Massimo)
Psql and libpq now compile under win32 using win32.mak(Magnus)
Lo_read no longer stores trailing NULL(Bruce)
Identifiers are now truncated to 31 characters internally(Bruce)
Createuser options now availble on the command line
Code for 64-bit integer supported added, configure tested, int8
 type(Thomas)
Prevent file descriptor leaf from failed COPY(Bruce)
New pg_upgrade command(Bruce)
Updated /contrib directories(Massimo)
New CREATE TABLE DEFAULT VALUES statement available(Thomas)
New INSERT INTO TABLE DEFAULT VALUES statement available(Thomas)
New DECLARE and FETCH feature(Thomas)
libpq's internal structures now not exported(Tom)
Allow up to 8 key indexes(Bruce)
Remove ARCHIVE keyword, that is no longer used(Thomas)
pg_dump -n flag to supress quotes around indentifiers
disable system columns for views(Jan)
new INET and CIDR types for network addresses(TomH, Paul)
```

```
no more double quotes in psql output
pg_dump now dumps views(Terry)
new SET QUERY_LIMIT(Tatsuo,Jan)

Source Tree Changes
-------------------
/contrib cleanup(Jun)
Inline some small functions called for every row(Bruce)
Alpha/linux fixes
Hp/UX cleanups(Tom)
Multi-byte regression tests(Soonmyung.)
Remove --disabled options from configure
Define PGDOC to use POSTGRESDIR by default
Make regression optional
Remove extra braces code to pgindent(Bruce)
Add bsdi shared library support(Bruce)
New --without-CXX support configure option(Brook)
New FAQ_CVS
Update backend flowchart in tools/backend(Bruce)
Change atttypmod from int16 to int32(Bruce, Tom)
Getrusage() fix for platforms that do not have it(Tom)
Add PQconnectdb, PGUSER, PGPASSWORD to libpq man page
NS32K platform fixes(Phil Nelson, John Buller)
Sco 7/UnixWare 2.x fixes(Billy,others)
Sparc/Solaris 2.5 fixes(Ryan)
Pgbuiltin.3 is obsolete, move to doc files(Thomas)
Even more documention(Thomas)
Nextstep support(Jacek)
Aix support(David)
pginterface manual page(Bruce)
shared libraries all have version numbers
merged all OS-specific shared library defines into one file
smarter TCL/TK configuration checking(Billy)
smarter perl configuration(Brook)
configure uses supplied install-sh if no install script found(Tom)
new Makefile.shlib for shared library configuration(Tom)
```

35.2. Release 6.3.2

This is a bugfix release for 6.3.x. Refer to the release notes for v6.3 for a more complete summary of new features.

Summary:

- Repairs automatic configuration support for some platforms, including Linux, from breakage inadvertently introduced in v6.3.1.
- Correctly handles function calls on the left side of BETWEEN and LIKE clauses.

A dump/restore is NOT required for those running 6.3 or 6.3.1. A 'make distclean', 'make', and 'make install' is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use Postgres libraries.

For upgrades from pre-v6.3 installations, refer to the installation and migration instructions for v6.3.

35.2.1. Detailed Change List

```
Changes
-------
Configure detection improvements for tcl/tk(Brook Milligan, Alvin)
Manual page improvements(Bruce)
BETWEEN and LIKE fix(Thomas)
fix for psql \connect used by pg_dump(Oliver Elphick)
New odbc driver
pgaccess, version 0.86
qsort removed, now uses libc version, cleanups(Jeroen)
fix for buffer over-runs detected(Maurice Gittens)
fix for buffer overrun in libpgtcl(Randy Kunkee)
fix for UNION with DISTINCT or ORDER BY(Bruce)
gettimeofday configure check(Doug Winterburn)
Fix "indexes not used" bug(Vadim)
docs additions(Thomas)
Fix for backend memory leak(Bruce)
libreadline cleanup(Erwan MAS)
Remove DISTDIR(Bruce)
Makefile dependency cleanup(Jeroen van Vianen)
ASSERT fixes(Bruce)
```

35.3. Release 6.3.1

Summary:

- Additional support for multi-byte character sets.
- Repair byte ordering for mixed-endian clients and servers.
- Minor updates to allowed SQL syntax.
- Improvements to the configuration autodetection for installation.
-

A dump/restore is NOT required for those running 6.3. A 'make distclean', 'make', and 'make install' is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use Postgres libraries.

For upgrades from pre-v6.3 installations, refer to the installation and migration instructions for v6.3.

35.3.1. Detailed Change List

```
Changes
-------
ecpg cleanup/fixes, now version 1.1(Michael Meskes)
pg_user cleanup(Bruce)
large object fix for pg_dump and tclsh (alvin)
LIKE fix for multiple adjacent underscores
fix for redefining builtin functions(Thomas)
ultrix4 cleanup
upgrade to pg_access 0.83
updated CLUSTER manual page
multi-byte character set support, see doc/README.mb(Tatsuo)
```

```
configure --with-pgport fix
pg_ident fix
big-endian fix for backend communications(Kataoka)
SUBSTR() and substring() fix(Jan)
several jdbc fixes(Peter)
libpgtcl improvements, see libptcl/README(Randy Kunkee)
Fix for "Datasize = 0" error(Vadim)
Prevent \do from wrapping(Bruce)
Remove duplicate Russian character set entries
Sunos4 cleanup
Allow optional TABLE keyword in LOCK and SELECT INTO(Thomas)
CREATE SEQUENCE options to allow a negative integer(Thomas)
Add "PASSWORD" as an allowed column identifier(Thomas)
Add checks for UNION target fields(Bruce)
Fix Alpha port(Dwayne Bailey)
Fix for text arrays containing quotes(Doug Gibson)
Solaris compile fix(Albert Chin-A-Young)
Better identify tcl and tk libs and includes(Bruce)
```

35.4. Release 6.3

There are *many* new features and improvements in this release. Here is a brief, incomplete summary:

- Many new SQL features, including full SQL92 subselect capability (everything is here but target-list subselects).
- Support for client-side environment variables to specify time zone and date style.
- Socket interface for client/server connection. This is the default now so you may need to start postmaster with the “-i” flag.
- Better password authorization mechanisms. Default table permissions have changed.
- Old-style “time travel” has been removed. Performance has been improved.

Note

Bruce Momjian wrote the following notes to introduce the new release.

There are some general 6.3 issues that I want to mention. These are only the big items that can not be described in one sentence. A review of the detailed changes list is still needed.

First, we now have subselects. Now that we have them, I would like to mention that without subselects, SQL is a very limited language. Subselects are a major feature, and you should review your code for places where subselects provide a better solution for your queries. I think you will find that there are more uses for subselects than you may think. Vadim has put us on the big SQL map with subselects, and fully functional ones too. The only thing you can't do with subselects is to use them in the target list.

Second, 6.3 uses unix domain sockets rather than TCP/IP by default. To enable connections from other machines, you have to use the new postmaster -i option, and of course edit pg_hba.conf. Also, for this reason, the format of pg_hba.conf has changed.

Third, char() fields will now allow faster access than varchar() or text. Specifically, the text and varchar() have a penalty for access to any columns after the first column of this type. char() used to also have this access penalty, but it no longer does. This may suggest that you redesign some of your tables, especially if you have short character columns that you have defined as varchar() or text. This and other changes make 6.3 even faster than earlier releases.

We now have passwords definable independent of any Unix file. There are new SQL USER commands. See the pg_hba.conf manual page for more information. There is a new table, pg_shadow, which is used to store user information and user passwords, and it by default only SELECT-able by the postgres super-user. pg_user is now a view of pg_shadow, and is SELECT-able by PUBLIC. You should keep using pg_user in your application without changes.

User-created tables now no longer have SELECT permission to PUBLIC by default. This was done because the ANSI standard requires it. You can of course GRANT any permissions you want after the table is created. System tables continue to be SELECT-able by PUBLIC.

We also have real deadlock detection code. No more sixty-second timeouts. And the new locking code implements a FIFO better, so there should be less resource starvation during heavy use.

Many complaints have been made about inadequate documenation in previous releases. Thomas has put much effort into many new manuals for this release. Check out the doc/ directory.

For performance reasons, time travel is gone, but can be implemented using triggers (see pgsql/contrib/spi/README). Please check out the new \d command for types, operators, etc. Also, views have their own permissions now, not based on the underlying tables, so permissions on them have to be set separately. Check /pgsql/interfaces for some new ways to talk to Postgres.

This is the first release that really required an explanation for existing users. In many ways, this was necessary because the new release removes many limitations, and the work-arounds people were using are no longer needed.

35.4.1. Migration to v6.3

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of Postgres.

35.4.2. Detailed Change List

```
Bug Fixes
---------
Fix binary cursors broken by MOVE implementation(Vadim)
Fix for tcl library crash(Jan)
Fix for array handling, from Gerhard Hintermayer
Fix acl error, and remove duplicate pqtrace(Bruce)
Fix psql \e for empty file(Bruce)
Fix for textcat on varchar() fields(Bruce)
Fix for DBT Sendproc (Zeugswetter Andres)
Fix vacuum analyze syntax problem(Bruce)
Fix for international identifiers(Tatsuo)
Fix aggregates on inherited tables(Bruce)
Fix substr() for out-of-bounds data
Fix for select 1=1 or 2=2, select 1=1 and 2=2, and select sum(2+2)
(Bruce)
Fix notty output to show status result.  -q option still turns it
 off(Bruce)
Fix for count(*), aggs with views and multiple tables and sum(3)
(Bruce)
Fix cluster(Bruce)
Fix for PQtrace start/stop several times(Bruce)
Fix a variety of locking problems like newer lock waiters getting
```

```
 lock before older waiters, and having readlock people not share
 locks if a writer is waiting for a lock, and waiting writers not
 getting priority over waiting readers(Bruce)
Fix crashes in psql when executing queries from external files(James)
Fix problem with multiple order by columns, with the first one having
 NULL values(Jeroen)
Use correct hash table support functions for float8 and int4(Thomas)
Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
Change precedence for boolean operators to match expected
 behavior(Thomas)
Generate elog(ERROR) on over-large integer(Bruce)
Allow multiple-argument functions in constraint clauses(Thomas)
Check boolean input literals for 'true','false','yes','no','1','0'
 and throw elog(ERROR) if unrecognized(Thomas)
Major large objects fix
Fix for GROUP BY showing duplicates(Vadim)
Fix for index scans in MergeJion(Vadim)

Enhancements
------------
Subselects with EXISTS, IN, ALL, ANY keywords (Vadim, Bruce, Thomas)
New User Manual(Thomas, others)
Speedup by inlining some frequently-called functions
Real deadlock detection, no more timeouts(Bruce)
Add SQL92 "constants" CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP,
 CURRENT_USER(Thomas)
Modify constraint syntax to be SQL92-compliant(Thomas)
Implement SQL92 PRIMARY KEY and UNIQUE clauses using indices(Thomas)
Recognize SQL92 syntax for FOREIGN KEY. Throw elog notice(Thomas)
Allow NOT NULL UNIQUE constraint clause (each allowed separately
 before)(Thomas)
Allow Postgres-style casting ("::") of non-constants(Thomas)
Add support for SQL3 TRUE and FALSE boolean constants(Thomas)
Support SQL92 syntax for IS TRUE/IS FALSE/IS NOT TRUE/IS NOT
 FALSE(Thomas)
Allow shorter strings for boolean literals (e.g. "t", "tr", "tru")
(Thomas)
Allow SQL92 delimited identifiers(Thomas)
Implement SQL92 binary and hexadecimal string decoding (b'10' and
 x'1F')(Thomas)
Support SQL92 syntax for type coercion of literal strings
 (e.g. "DATETIME 'now'")(Thomas)
Add conversions for int2, int4, and OID types to and from text(Thomas)
Use shared lock when building indices(Vadim)
Free memory allocated for an user query inside transaction block after
 this query is done, was turned off in <= 6.2.1(Vadim)
New SQL statement CREATE PROCEDURAL LANGUAGE(Jan)
New Postgres Procedural Language (PL) backend interface(Jan)
Rename pg_dump -H option to -h(Bruce)
Add Java support for passwords, European dates(Peter)
Use indices for LIKE and ~, !~ operations(Bruce)
Add hash functions for datetime and timespan(Thomas)
Time Travel removed(Vadim, Bruce)
Add paging for \d and \z, and fix \i(Bruce)
```

```
Add Unix domain socket support to backend and to frontend
 library(Goran)
Implement CREATE DATABASE/WITH LOCATION and initlocation
 utility(Thomas)
Allow more SQL92 and/or Postgres reserved words as column
 identifiers(Thomas)
Augment support for SQL92 SET TIME ZONE...(Thomas)
SET/SHOW/RESET TIME ZONE uses TZ backend environment variable(Thomas)
Implement SET keyword = DEFAULT and SET TIME ZONE DEFAULT(Thomas)
Enable SET TIME ZONE using TZ environment variable(Thomas)
Add PGDATESTYLE environment variable to frontend and backend
 initialization(Thomas)
Add PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGGEQO
 frontend library initialization environment variables(Thomas)
Regression tests time zone automatically set with "setenv PGTZ
 PST8PDT"(Thomas)
Add pg_description table for info on tables, columns, operators,
 types, and
 aggregates(Bruce)
Increase 16 char limit on system table/index names to 32
 characters(Bruce)
Rename system indices(Bruce)
Add 'GERMAN' option to SET DATESTYLE(Thomas)
Define an "ISO-style" timespan output format with "hh:mm:ss"
 fields(Thomas)
Allow fractional values for delta times (e.g. '2.5 days')(Thomas)
Validate numeric input more carefully for delta times(Thomas)
Implement day of year as possible input to date_part()(Thomas)
Define timespan_finite() and text_timespan() functions(Thomas)
Remove archive stuff(Bruce)
Allow for a pg_password authentication database that is separate from
 the system password file(Todd)
Dump ACLs, GRANT, REVOKE permissions(Matt)
Define text, varchar, and bpchar string length functions(Thomas)
Fix Query handling for inheritance, and cost computations(Bruce)
Implement CREATE TABLE/AS SELECT (alternative to SELECT/INTO)(Thomas)
Allow NOT, IS NULL, IS NOT NULL in constraints(Thomas)
Implement UNIONs for SELECT(Bruce)
Add UNION, GROUP, DISTINCT to INSERT(Bruce)
varchar() stores only necessary bytes on disk(Bruce)
Fix for BLOBs(Peter)
Mega-Patch for JDBC...see README_6.3 for list of changes(Peter)
Remove unused "option" from PQconnectdb()
New LOCK command and lock manual page describing deadlocks(Bruce)
Add new psql \da, \dd, \df, \do, \dS, and \dT commands(Bruce)
Enhance psql \z to show sequences(Bruce)
Show NOT NULL and DEFAULT in psql \d table(Bruce)
New psql .psqlrc file startup(Andrew)
Modify sample startup script in contrib/linux to show syslog(Thomas)
New types for IP and MAC addresses in contrib/ip_and_mac(TomH)
Unix system time conversions with date/time types in contrib/
unixdate(Thomas)
Update of contrib stuff(Massimo)
Add Unix socket support to DBD::Pg(Goran)
```

```
New python interface (PyGreSQL 2.0)(D'Arcy)
New frontend/backend protocol has a version number, network byte
 order(Phil)
Security features in pg_hba.conf enhanced and documented, many
 cleanups(Phil)
CHAR() now faster access than VARCHAR() or TEXT
ecpg embedded SQL preprocessor
Reduce system column overhead(Vadmin)
Remove pg_time table(Vadim)
Add pg_type attribute to identify types that need length (bpchar,
 varchar)
Add report of offending line when COPY command fails
Allow VIEW permissions to be set separately from the underlying
 tables.
 For security, use GRANT/REVOKE on views as appropriate(Jan)
Tables now have no default GRANT SELECT TO PUBLIC.  You must
 explicitly grant such permissions.
Clean up tutorial examples(Darren)

Source Tree Changes
-------------------
Add new html development tools, and flow chart in /tools/backend
Fix for SCO compiles
Stratus computer port Robert Gillies
Added support for shlib for BSD44_derived & i386_solaris
Make configure more automated(Brook)
Add script to check regression test results
Break parser functions into smaller files, group together(Bruce)
Rename heap_create to heap_create_and_catalog, rename heap_creatr
 to heap_create()(Bruce)
Sparc/Linux patch for locking(TomS)
Remove PORTNAME and reorganize port-specific stuff(Marc)
Add optimizer README file(Bruce)
Remove some recursion in optimizer and clean up some code there(Bruce)
Fix for NetBSD locking(Henry)
Fix for libptcl make(Tatsuo)
AIX patch(Darren)
Change IS TRUE, IS FALSE, ... to expressions using "=" rather than
 function calls to istrue() or isfalse() to allow optimization(Thomas)
Various fixes NetBSD/Sparc related(TomH)
Alpha linux locking(Travis,Ryan)
Change elog(WARN) to elog(ERROR)(Bruce)
FAQ for FreeBSD(Marc)
Bring in the PostODBC source tree as part of our standard
 distribution(Marc)
A minor patch for HP/UX 10 vs 9(Stan)
New pg_attribute.atttypmod for type-specific info like varchar
 length(Bruce)
Unixware patches(Billy)
New i386 'lock' for spin lock asm(Billy)
Support for multiplexed backends is removed
Start an OpenBSD port
Start an AUX port
Start a Cygnus port
```

```
Add string functions to regression suite(Thomas)
Expand a few function names formerly truncated to 16
 characters(Thomas)
Remove un-needed malloc() calls and replace with palloc()(Bruce)
```

35.5. Release 6.2.1

v6.2.1 is a bug-fix and usability release on v6.2.

Summary:

- Allow strings to span lines, per SQL92.
- Include example trigger function for inserting user names on table updates.

This is a minor bug-fix release on v6.2. For upgrades from pre-v6.2 systems, a full dump/reload is required. Refer to the v6.2 release notes for instructions.

35.5.1. Migration from v6.2 to v6.2.1

This is a minor bug-fix release. A dump/reload is not required from v6.2, but is required from any release prior to v6.2.

In upgrading from v6.2, if you choose to dump/reload you will find that avg(money) is now calculated correctly. All other bug fixes take effect upon updating the executables.

Another way to avoid dump/reload is to use the following SQL command from psql to update the existing system table:

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
 where aggname = 'avg' and aggbasetype = 790;
```

This will need to be done to every existing database, including template1.

35.5.2. Detailed Change List

```
Changes in this release
-----------------------
Allow TIME and TYPE column names(Thomas)
Allow larger range of true/false as boolean values(Thomas)
Support output of "now" and "current"(Thomas)
Handle DEFAULT with INSERT of NULL properly(Vadim)
Fix for relation reference counts problem in buffer manager(Vadim)
Allow strings to span lines, like ANSI(Thomas)
Fix for backward cursor with ORDER BY(Vadim)
Fix avg(cash) computation(Thomas)
Fix for specifying a column twice in ORDER/GROUP BY(Vadim)
Documented new libpq function to return affected rows,
 PQcmdTuples(Bruce)
Trigger function for inserting user names for INSERT/UPDATE(Brook
 Milligan)
```

35.6. Release 6.2

A dump/restore is required for those wishing to migrate data from previous releases of Postgres.

35.6.1. Migration from v6.1 to v6.2

This migration requires a complete dump of the 6.1 database and a restore of the database in 6.2.

Note that the pg_dump and pg_dumpall utility from 6.2 should be used to dump the 6.1 database.

35.6.2. Migration from v1.x to v6.2

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

35.6.3. Detailed Change List

```
Bug Fixes
---------
Fix problems with pg_dump for inheritance, sequences, archive
 tables(Bruce)
Fix compile errors on overflow due to shifts, unsigned, and bad
 prototypes
  from Solaris(Diab Jerius)
Fix bugs in geometric line arithmetic (bad intersection calculations)
(Thomas)
Check for geometric intersections at endpoints to avoid rounding
 ugliness(Thomas)
Catch non-functional delete attempts(Vadim)
Change time function names to be more consistent(Michael Reifenberg)
Check for zero divides(Michael Reifenberg)
Fix very old bug which made tuples changed/inserted by a commnd
 visible to the command itself (so we had multiple update of
 updated tuples, etc)(Vadim)
Fix for SELECT null, 'fail' FROM pg_am (Patrick)
SELECT NULL as EMPTY_FIELD now allowed(Patrick)
Remove un-needed signal stuff from contrib/pginterface
Fix OR (where x <> 1 or x isnull didn't return tuples with x NULL)
 (Vadim)
Fix time_cmp function (Vadim)
Fix handling of functions with non-attribute first argument in
 WHERE clauses (Vadim)
Fix GROUP BY when order of entries is different from order
 in target list (Vadim)
Fix pg_dump for aggregates without sfunc1 (Vadim)

Enhancements
------------
Default genetic optimizer GEQO parameter is now 8(Bruce)
Allow use parameters in target list having aggregates in
 functions(Vadim)
Added JDBC driver as an interface(Adrian & Peter)
pg_password utility
Return number of tuples inserted/affected by INSERT/UPDATE/DELETE etc.
(Vadim)
Triggers implemented with CREATE TRIGGER (SQL3)(Vadim)
SPI (Server Programming Interface) allows execution of queries inside
```

```
 C-functions (Vadim)
NOT NULL implemented (SQL92)(Robson Paniago de Miranda)
Include reserved words for string handling, outer joins, and
 unions(Thomas)
Implement extended comments ("/* ... */") using exclusive
 states(Thomas)
Add "//" single-line comments(Bruce)
Remove some restrictions on characters in operator names(Thomas)
DEFAULT and CONSTRAINT for tables implemented (SQL92)(Vadim & Thomas)
Add text concatenation operator and function (SQL92)(Thomas)
Support WITH TIME ZONE syntax (SQL92)(Thomas)
Support INTERVAL unit TO unit syntax (SQL92)(Thomas)
Define types DOUBLE PRECISION, INTERVAL, CHARACTER,
 and CHARACTER VARYING (SQL92)(Thomas)
Define type FLOAT(p) and rudimentary DECIMAL(p,s), NUMERIC(p,s)
 (SQL92)(Thomas)
Define EXTRACT(), POSITION(), SUBSTRING(), and TRIM() (SQL92)(Thomas)
Define CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92)(Thomas)
Add syntax and warnings for UNION, HAVING, INNER and OUTER JOIN
 (SQL92)(Thomas)
Add more reserved words, mostly for SQL92 compliance(Thomas)
Allow hh:mm:ss time entry for timespan/reltime types(Thomas)
Add center() routines for lseg, path, polygon(Thomas)
Add distance() routines for circle-polygon, polygon-polygon(Thomas)
Check explicitly for points and polygons contained within polygons
 using an axis-crossing algorithm(Thomas)
Add routine to convert circle-box(Thomas)
Merge conflicting operators for different geometric data types(Thomas)
Replace distance operator "<===>" with "<->"(Thomas)
Replace "above" operator "!^" with ">^" and "below" operator "!|" with
 "<^"(Thomas)
Add routines for text trimming on both ends, substring, and string
 position(Thomas)
Added conversion routines circle(box) and poly(circle)(Thomas)
Allow internal sorts to be stored in memory rather than in files(Bruce
 & Vadim)
Allow functions and operators on internally-identical types to
 succeed(Bruce)
Speed up backend startup after profiling analysis(Bruce)
Inline frequently called functions for performance(Bruce)
Reduce open() calls(Bruce)
psql:  Add PAGER for \h and \?,\C fix
Fix for psql pager when no tty(Bruce)
New entab utility(Bruce)
General trigger functions for referential integrity (Vadim)
General trigger functions for time travel (Vadim)
General trigger functions for AUTOINCREMENT/IDENTITY feature (Vadim)
MOVE implementation (Vadim)

Source Tree Changes
-------------------
HPUX 10 patches (Vladimir Turin)
Added SCO support, (Daniel Harris)
mkLinux patches (Tatsuo Ishii)
```

```
Change geometric box terminology from "length" to "width"(Thomas)
Deprecate temporary unstored slope fields in geometric code(Thomas)
Remove restart instructions from INSTALL(Bruce)
Look in /usr/ucb first for install(Bruce)
Fix c++ copy example code(Thomas)
Add -o to psql manual page(Bruce)
Prevent relname unallocated string length from being copied into
 database(Bruce)
Cleanup for NAMEDATALEN use(Bruce)
Fix pg_proc names over 15 chars in output(Bruce)
Add strNcpy() function(Bruce)
remove some (void) casts that are unnecessary(Bruce)
new interfaces directory(Marc)
Replace fopen() calls with calls to fd.c functions(Bruce)
Make functions static where possible(Bruce)
enclose unused functions in #ifdef NOT_USED(Bruce)
Remove call to difftime() in timestamp support to fix SunOS(Bruce &
 Thomas)
Changes for Digital Unix
Portability fix for pg_dumpall(Bruce)
Rename pg_attribute.attnvals to attdisbursion(Bruce)
"intro/unix" manual page now "pgintro"(Bruce)
"built-in" manual page now "pgbuiltin"(Bruce)
"drop" manual page now "drop_table"(Bruce)
Add "create_trigger", "drop_trigger" manual pages(Thomas)
Add constraints regression test(Vadim & Thomas)
Add comments syntax regression test(Thomas)
Add PGINDENT and support program(Bruce)
Massive commit to run PGINDENT on all *.c and *.h files(Bruce)
Files moved to /src/tools directory(Bruce)
SPI and Trigger programming guides (Vadim & D'Arcy)
```

35.7. Release 6.1.1

35.7.1. Migration from v6.1 to v6.1.1

This is a minor bug-fix release. A dump/reload is not required from v6.1, but is required from any release prior to v6.1. Refer to the release notes for v6.1 for more details.

35.7.2. Detailed Change List

```
Changes in this release
-----------------------
fix for SET with options (Thomas)
allow pg_dump/pg_dumpall to preserve ownership of all tables/
objects(Bruce)
new psql \connect option allows changing usernames without chaning
 databases
fix for initdb --debug option(Yoshihiko Ichikawa))
lextest cleanup(Bruce)
hash fixes(Vadim)
fix date/time month boundary arithmetic(Thomas)
```

```
fix timezone daylight handling for some ports(Thomas, Bruce, Tatsuo)
timestamp overhauled to use standard functions(Thomas)
other code cleanup in date/time routines(Thomas)
psql's \d now case-insensitive(Bruce)
psql's backslash commands can now have trailing semicolon(Bruce)
fix memory leak in psql when using \g(Bruce)
major fix for endian handling of communication to server(Thomas,
 Tatsuo)
Fix for Solaris assembler and include files(Yoshihiko Ichikawa)
allow underscores in usernames(Bruce)
pg_dumpall now returns proper status, portability fix(Bruce)
```

35.8. Release 6.1

The regression tests have been adapted and extensively modified for the v6.1 release of Postgres.

Three new data types (datetime, timespan, and circle) have been added to the native set of Postgres types. Points, boxes, paths, and polygons have had their output formats made consistant across the data types. The polygon output in misc.out has only been spot-checked for correctness relative to the original regression output.

Postgres v6.1 introduces a new, alternate optimizer which uses *genetic* algorithms. These algorithms introduce a random behavior in the ordering of query results when the query contains multiple qualifiers or multiple tables (giving the optimizer a choice on order of evaluation). Several regression tests have been modified to explicitly order the results, and hence are insensitive to optimizer choices. A few regression tests are for data types which are inherently unordered (e.g. points and time intervals) and tests involving those types are explicitly bracketed with `set geqo to 'off'` and `reset geqo`.

The interpretation of array specifiers (the curly braces around atomic values) appears to have changed sometime after the original regression tests were generated. The current `./expected/*.out` files reflect this new interpretation, which may not be correct!

The float8 regression test fails on at least some platforms. This is due to differences in implementations of pow() and exp() and the signaling mechanisms used for overflow and underflow conditions.

The "random" results in the random test should cause the "random" test to be "failed", since the regression tests are evaluated using a simple diff. However, "random" does not seem to produce random results on my test machine (Linux/gcc/i686).

35.8.1. Migration to v6.1

This migration requires a complete dump of the 6.0 database and a restore of the database in 6.1.

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

35.8.2. Detailed Change List

```
Bug Fixes
---------
packet length checking in library routines
lock manager priority patch
check for under/over flow of float8(Bruce)
multi-table join fix(Vadim)
```

```
SIGPIPE crash fix(Darren)
large object fixes(Sven)
allow btree indexes to handle NULLs(Vadim)
timezone fixes(D'Arcy)
select SUM(x) can return NULL on no rows(Thomas)
internal optimizer, executor bug fixes(Vadim)
fix problem where inner loop in < or <= has no rows(Vadim)
prevent re-commuting join index clauses(Vadim)
fix join clauses for multiple tables(Vadim)
fix hash, hashjoin for arrays(Vadim)
fix btree for abstime type(Vadim)
large object fixes(Raymond)
fix buffer leak in hash indices (Vadim)
fix rtree for use in inner scan (Vadim)
fix gist for use in inner scan, cleanups (Vadim, Andrea)
avoid unnecessary local buffers allocation (Vadim, Massimo)
fix local buffers leak in transaction aborts (Vadim)
fix file manager memmory leaks, cleanups (Vadim, Massimo)
fix storage manager memmory leaks (Vadim)
fix btree duplicates handling (Vadim)
fix deleted tuples re-incarnation caused by vacuum (Vadim)
fix SELECT varchar()/char() INTO TABLE made zero-length fields(Bruce)
many psql, pg_dump, and libpq memory leaks fixed using Purify (Igor)

Enhancements
------------
attribute optimization statistics(Bruce)
much faster new btree bulk load code(Paul)
BTREE UNIQUE added to bulk load code(Vadim)
new lock debug code(Massimo)
massive changes to libpg++(Leo)
new GEQO optimizer speeds table multi-table optimization(Martin)
new WARN message for non-unique insert into unique key(Marc)
update x=-3, no spaces, now valid(Bruce)
remove case-sensitive identifier handling(Bruce,Thomas,Dan)
debug backend now pretty-prints tree(Darren)
new Oracle character functions(Edmund)
new plaintext password functions(Dan)
no such class or insufficient privilege changed to distinct
 messages(Dan)
new ANSI timestamp function(Dan)
new ANSI Time and Date types (Thomas)
move large chunks of data in backend(Martin)
multi-column btree indexes(Vadim)
new SET var TO value command(Martin)
update transaction status on reads(Dan)
new locale settings for character types(Oleg)
new SEQUENCE serial number generator(Vadim)
GROUP BY function now possible(Vadim)
re-organize regression test(Thomas,Marc)
new optimizer operation weights(Vadim)
new psql \z grant/permit option(Marc)
new MONEY data type(D'Arcy,Thomas)
tcp socket communication speed improved(Vadim)
```

```
new VACUUM option for attribute statistics, and for certain columns
 (Vadim)
many geometric type improvements(Thomas,Keith)
additional regression tests(Thomas)
new datestyle variable(Thomas,Vadim,Martin)
more comparison operators for sorting types(Thomas)
new conversion functions(Thomas)
new more compact btree format(Vadim)
allow pg_dumpall to preserve database ownership(Bruce)
new SET GEQO=# and R_PLANS variable(Vadim)
old (!GEQO) optimizer can use right-sided plans (Vadim)
typechecking improvement in SQL parser(Bruce)
new SET, SHOW, RESET commands(Thomas,Vadim)
new \connect database USER option
new destroydb -i option (Igor)
new \dt and \di psql commands (Darren)
SELECT "\n" now escapes newline (A. Duursma)
new geometry conversion functions from old format (Thomas)

Source tree changes
-------------------
new configuration script(Marc)
readline configuration option added(Marc)
OS-specific configuration options removed(Marc)
new OS-specific template files(Marc)
no more need to edit Makefile.global(Marc)
re-arrange include files(Marc)
nextstep patches (Gregor Hoffleit)
removed WIN32-specific code(Bruce)
removed postmaster -e option, now only postgres -e option (Bruce)
merge duplicate library code in front/backends(Martin)
now works with eBones, international Kerberos(Jun)
more shared library support
c++ include file cleanup(Bruce)
warn about buggy flex(Bruce)
DG-UX, Ultrix, Irix, AIX portability fixes
```

35.9. Release v6.0

A dump/restore is required for those wishing to migrate data from previous releases of Postgres.

35.9.1. Migration from v1.09 to v6.0

This migration requires a complete dump of the 1.09 database and a restore of the database in 6.0.

35.9.2. Migration from pre-v1.09 to v6.0

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

35.9.3. Detailed Change List

```
Bug Fixes
```

```
---------
ALTER TABLE bug - running postgress process needs to re-read table
 definition
Allow vacuum to be run on one table or entire database(Bruce)
Array fixes
Fix array over-runs of memory writes(Kurt)
Fix elusive btree range/non-range bug(Dan)
Fix for hash indexes on some types like time and date
Fix for pg_log size explosion
Fix permissions on lo_export()(Bruce)
Fix unitialized reads of memory(Kurt)
Fixed ALTER TABLE ... char(3) bug(Bruce)
Fixed a few small memory leaks
Fixed EXPLAIN handling of options and changed full_path option name
Fixed output of group acl permissions
Memory leaks (hunt and destroy with tools like Purify(Kurt)
Minor improvements to rules system
NOTIFY fixes
New asserts for run-checking
Overhauled parser/analyze code to properly report errors and increase
 speed
Pg_dump -d now handles NULL's properly(Bruce)
Prevent SELECT NULL from crashing server (Bruce)
Properly report errors when INSERT ... SELECT columns did not match
Properly report errors when insert column names were not correct
Psql \g filename now works(Bruce)
Psql fixed problem with multiple statements on one line with multiple
 outputs
Removed duplicate system oid's
SELECT * INTO TABLE . GROUP/ORDER BY gives unlink error if table
 exists(Bruce)
Several fixes for queries that crashed the backend
Starting quote in insert string errors(Bruce)
Submitting an empty query now returns empty status, not just " "
 query(Bruce)

Enhancements
------------
Add EXPLAIN manual page(Bruce)
Add UNIQUE index capability(Dan)
Add hostname/user level access control rather than just hostname and
 user
Add synonym of != for <>(Bruce)
Allow "select oid,* from table"
Allow BY,ORDER BY to specify columns by number, or by non-alias
 table.column(Bruce)
Allow COPY from the frontend(Bryan)
Allow GROUP BY to use alias column name(Bruce)
Allow actual compression, not just reuse on the same page(Vadim)
Allow installation-configuration option to auto-add all local
 users(Bryan)
Allow libpq to distinguish between text value '' and null(Bruce)
Allow non-postgres users with createdb privs to destroydb's
Allow restriction on who can create C functions(Bryan)
```

```
Allow restriction on who can do backend COPY(Bryan)
Can shrink tables, pg_time and pg_log(Vadim & Erich)
Change debug level 2 to print queries only, changed debug heading
 layout(Bruce)
Change default decimal constant representation from float4 to
 float8(Bruce)
European date format now set when postmaster is started
Execute lowercase function names if not found with exact case
Fixes for aggregate/GROUP processing, allow 'select sum(func(x),sum(x
+y) from z'
Gist now included in the distrubution(Marc)
Idend authentication of local users(Bryan)
Implement BETWEEN qualifier(Bruce)
Implement IN qualifier(Bruce)
Libpq has PQgetisnull()(Bruce)
Libpq++ improvements
New options to initdb(Bryan)
Pg_dump allow dump of oid's(Bruce)
Pg_dump create indexes after tables are loaded for speed(Bruce)
Pg_dumpall dumps all databases, and the user table
Pginterface additions for NULL values(Bruce)
Prevent postmaster from being run as root
Psql \h and \? is now readable(Bruce)
Psql allow backslashed, semicolons anywhere on the line(Bruce)
Psql changed command prompt for lines in query or in quotes(Bruce)
Psql char(3) now displays as (bp)char in \d output(Bruce)
Psql return code now more accurate(Bryan?)
Psql updated help syntax(Bruce)
Re-visit and fix vacuum(Vadim)
Reduce size of regression diffs, remove timezone name
 difference(Bruce)
Remove compile-time parameters to enable binary distributions(Bryan)
Reverse meaning of HBA masks(Bryan)
Secure Authentication of local users(Bryan)
Speed up vacuum(Vadim)
Vacuum now had VERBOSE option(Bruce)

Source tree changes
-------------------
All functions now have prototypes that are compared against the calls
Allow asserts to be disabled easly from Makefile.global(Bruce)
Change oid constants used in code to #define names
Decoupled sparc and solaris defines(Kurt)
Gcc -Wall compiles cleanly with warnings only from unfixable
 constructs
Major include file reorganization/reduction(Marc)
Make now stops on compile failure(Bryan)
Makefile restructuring(Bryan, Marc)
Merge bsdi_2_1 to bsdi(Bruce)
Monitor program removed
Name change from Postgres95 to PostgreSQL
New config.h file(Marc, Bryan)
PG_VERSION now set to 6.0 and used by postmaster
Portability additions, including Ultrix, DG/UX, AIX, and Solaris
```

```
Reduced the number of #define's, centeralized #define's
Remove duplicate OIDS in system tables(Dan)
Remove duplicate system catalog info or report mismatches(Dan)
Removed many os-specific #define's
Restructured object file generation/location(Bryan, Marc)
Restructured port-specific file locations(Bryan, Marc)
Unused/uninialized variables corrected
```

35.10. Release v1.09

Sorry, we stopped keeping track of changes from 1.02 to 1.09. Some of the changes listed in 6.0 were actually included in the 1.02.1 to 1.09 releases.

35.11. Release v1.02

35.11.1. Migration from v1.02 to v1.02.1

Here is a new migration file for 1.02.1. It includes the 'copy' change and a script to convert old ascii files.

Note

The following notes are for the benefit of users who want to migrate databases from postgres95 1.01 and 1.02 to postgres95 1.02.1.

If you are starting afresh with postgres95 1.02.1 and do not need to migrate old databases, you do not need to read any further.

In order to upgrade older postgres95 version 1.01 or 1.02 databases to version 1.02.1, the following steps are required:

1. Start up a new 1.02.1 postmaster

2. Add the new built-in functions and operators of 1.02.1 to 1.01 or 1.02 databases. This is done by running the new 1.02.1 server against your own 1.01 or 1.02 database and applying the queries attached at the end of thie file. This can be done easily through psql. If your 1.01 or 1.02 database is named "testdb" and you have cut the commands from the end of this file and saved them in addfunc.sql:

   ```
   % psql testdb -f addfunc.sql
   ```

   Those upgrading 1.02 databases will get a warning when executing the last two statements in the file because they are already present in 1.02. This is not a cause for concern.

35.11.2. Dump/Reload Procedure

If you are trying to reload a pg_dump or text-mode 'copy tablename to stdout' generated with a previous version, you will need to run the attached sed script on the ASCII file before loading it into the database. The old format used '.' as end-of-data, while '\.' is now the end-of-data marker. Also, empty strings are now loaded in as '' rather than NULL. See the copy manual page for full details.

```
sed 's/^\.$/\\./g' <in_file >out_file
```

If you are loading an older binary copy or non-stdout copy, there is no end-of-data character, and hence no conversion necessary.

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure =
 texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure =
 texticregexne);
```

35.11.3. Detailed Change List

```
Source code maintenance and development
 * worldwide team of volunteers
 * the source tree now in CVS at ftp.ki.net

Enhancements
 * psql (and underlying libpq library) now has many more options for
   formatting output, including HTML
 * pg_dump now output the schema and/or the data, with many fixes to
   enhance completeness.
 * psql used in place of monitor in administration shell scripts.
   monitor to be depreciated in next release.
 * date/time functions enhanced
 * NULL insert/update/comparison fixed/enhanced
 * TCL/TK lib and shell fixed to work with both tck7.4/tk4.0 and
 tcl7.5/tk4.1

Bug Fixes (almost too numerous to mention)
 * indexes
 * storage management
 * check for NULL pointer before dereferencing
 * Makefile fixes

New Ports
 * added SolarisX86 port
 * added BSDI 2.1 port
 * added DGUX port
```

35.12. Release v1.01

35.12.1. Migration from v1.0 to v1.01

The following notes are for the benefit of users who want to migrate databases from postgres95 1.0 to postgres95 1.01.

If you are starting afresh with postgres95 1.01 and do not need to migrate old databases, you do not need to read any further.

In order to postgres95 version 1.01 with databases created with postgres95 version 1.0, the following steps are required:

1. Set the definition of NAMEDATALEN in src/Makefile.global to 16 and OIDNAMELEN to 20.

2. Decide whether you want to use Host based authentication.

   a. If you do, you must create a file name "pg_hba" in your top-level data directory (typically the value of your $PGDATA). src/libpq/pg_hba shows an example syntax.

   b. If you do not want host-based authentication, you can comment out the line

      ```
      HBA = 1
      ```

      in src/Makefile.global

      Note that host-based authentication is turned on by default, and if you do not take steps A or B above, the out-of-the-box 1.01 will not allow you to connect to 1.0 databases.

3. Compile and install 1.01, but DO NOT do the initdb step.

4. Before doing anything else, terminate your 1.0 postmaster, and backup your existing $PGDATA directory.

5. Set your PGDATA environment variable to your 1.0 databases, but set up path up so that 1.01 binaries are being used.

6. Modify the file $PGDATA/PG_VERSION from 5.0 to 5.1

7. Start up a new 1.01 postmaster

8. Add the new built-in functions and operators of 1.01 to 1.0 databases. This is done by running the new 1.01 server against your own 1.0 database and applying the queries attached and saving in the file 1.0_to_1.01.sql. This can be done easily through psql. If your 1.0 database is name "testdb":

   ```
   % psql testdb -f 1.0_to_1.01.sql
   ```

   and then execute the following commands (cut and paste from here):

   ```
   -- add builtin functions that are new to 1.01

   create function int4eqoid (int4, oid) returns bool as 'foo'
   language 'internal';
   create function oideqint4 (oid, int4) returns bool as 'foo'
   language 'internal';
   create function char2icregexeq (char2, text) returns bool as 'foo'
   language 'internal';
   create function char2icregexne (char2, text) returns bool as 'foo'
   language 'internal';
   create function char4icregexeq (char4, text) returns bool as 'foo'
   language 'internal';
   create function char4icregexne (char4, text) returns bool as 'foo'
   language 'internal';
   create function char8icregexeq (char8, text) returns bool as 'foo'
   language 'internal';
   create function char8icregexne (char8, text) returns bool as 'foo'
   language 'internal';
   create function char16icregexeq (char16, text) returns bool as
    'foo'
   language 'internal';
   ```

```
create function char16icregexne (char16, text) returns bool as
 'foo'
language 'internal';
create function texticregexeq (text, text) returns bool as 'foo'
language 'internal';
create function texticregexne (text, text) returns bool as 'foo'
language 'internal';

-- add builtin functions that are new to 1.01

create operator = (leftarg = int4, rightarg = oid, procedure =
 int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure =
 oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure =
 char2icregexeq);
create operator !~* (leftarg = char2, rightarg = text, procedure =
 char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure =
 char4icregexeq);
create operator !~* (leftarg = char4, rightarg = text, procedure =
 char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure =
 char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure =
 char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure =
 char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure
 = char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure =
 texticregexne);
```

35.12.2. Detailed Change List

```
Incompatibilities:
 * 1.01 is backwards compatible with 1.0 database provided the user
   follow the steps outlined in the MIGRATION_from_1.0_to_1.01 file.
   If those steps are not taken, 1.01 is not compatible with 1.0
 database.

Enhancements:
 * added PQdisplayTuples() to libpq and changed monitor and psql to
 use it
 * added NeXT port (requires SysVIPC implementation)
 * added CAST .. AS ... syntax
 * added ASC and DESC keywords
 * added 'internal' as a possible language for CREATE FUNCTION
   internal functions are C functions which have been statically
 linked
   into the postgres backend.
```

```
 * a new type "name" has been added for system identifiers (table
 names,
   attribute names, etc.)  This replaces the old char16 type.   The
   of name is set by the NAMEDATALEN #define in src/Makefile.global
 * a readable reference manual that describes the query language.
 * added host-based access control.  A configuration file ($PGDATA/
pg_hba)
   is used to hold the configuration data.  If host-based access
 control
   is not desired, comment out HBA=1 in src/Makefile.global.
 * changed regex handling to be uniform use of Henry Spencer's regex
 code
   regardless of platform.  The regex code is included in the
 distribution
 * added functions and operators for case-insensitive regular
 expressions.
   The operators are ~* and !~*.
 * pg_dump uses COPY instead of SELECT loop for better performance

Bug fixes:
 * fixed an optimizer bug that was causing core dumps when
   functions calls were used in comparisons in the WHERE clause
 * changed all uses of getuid to geteuid so that effective uids are
 used
 * psql now returns non-zero status on errors when using -c
 * applied public patches 1-14
```

35.13. Release v1.0

35.13.1. Detailed Change List

```
Copyright change:
 * The copyright of Postgres 1.0 has been loosened to be freely
 modifiable
   and modifiable for any purpose.  Please read the COPYRIGHT file.
   Thanks to Professor Michael Stonebraker for making this possible.

Incompatibilities:
 *  date formats have to be MM-DD-YYYY (or DD-MM-YYYY if you're using
   EUROPEAN STYLE).  This follows SQL-92 specs.
 *  "delimiters" is now a keyword

Enhancements:
 *  sql LIKE syntax has been added
 *  copy command now takes an optional USING DELIMITER specification.
   delimiters can be any single-character string.
 *  IRIX 5.3 port has been added.
   Thanks to Paul Walmsley and others.
 *  updated pg_dump to work with new libpq
 *  \d has been added psql
   Thanks to Keith Parks
 *  regexp performance for architectures that use POSIX regex has been
   improved due to caching of precompiled patterns.
```

```
    Thanks to Alistair Crooks
 *  a new version of libpq++
    Thanks to William Wanders

Bug fixes:
 *  arbitrary userids can be specified in the createuser script
 *  \c to connect to other databases in psql now works.
 *  bad pg_proc entry for float4inc() is fixed
 *  users with usecreatedb field set can now create databases without
    having to be usesuper
 *  remove access control entries when the entry no longer has any
    permissions
 *  fixed non-portable datetimes implementation
 *  added kerberos flags to the src/backend/Makefile
 *  libpq now works with kerberos
 *  typographic errors in the user manual have been corrected.
 *  btrees with multiple index never worked, now we tell you they
 don't
    work when you try to use them
```

35.14. Postgres95 Beta 0.03

35.14.1. Detailed Change List

```
Incompatible changes:
 * BETA-0.3 IS INCOMPATIBLE WITH DATABASES CREATED WITH PREVIOUS
 VERSIONS
    (due to system catalog changes and indexing structure changes).
 * double-quote (") is deprecated as a quoting character for string
 literals;
    you need to convert them to single quotes (').
 * name of aggregates (eg. int4sum) are renamed in accordance with the
    SQL standard (eg. sum).
 * CHANGE ACL syntax is replaced by GRANT/REVOKE syntax.
 * float literals (eg. 3.14) are now of type float4 (instead of float8
 in
    previous releases); you might have to do typecasting if you depend
 on it
    being of type float8.  If you neglect to do the typecasting and you
 assign
    a float literal to a field of type float8, you may get incorrect
 values
    stored!
 * LIBPQ has been totally revamped so that frontend applications
    can connect to multiple backends
 * the usesysid field in pg_user has been changed from int2 to int4 to
    allow wider range of Unix user ids.
 * the netbsd/freebsd/bsd o/s ports have been consolidated into a
    single BSD44_derived port.  (thanks to Alistair Crooks)

SQL standard-compliance (the following details changes that makes
 postgres95
more compliant to the SQL-92 standard):
```

```
 * the following SQL types are now built-in: smallint, int(eger),
 float, real,
   char(N), varchar(N), date and time.

   The following are aliases to existing postgres types:
  smallint -> int2
  integer, int -> int4
  float, real  -> float4
   char(N) and varchar(N) are implemented as truncated text types. In
   addition, char(N) does blank-padding.
 * single-quote (') is used for quoting string literals; '' (in
 addition to
   \') is supported as means of inserting a single quote in a string
 * SQL standard aggregate names (MAX, MIN, AVG, SUM, COUNT) are used
   (Also, aggregates can now be overloaded, i.e. you can define your
   own MAX aggregate to take in a user-defined type.)
 * CHANGE ACL removed. GRANT/REVOKE syntax added.
   - Privileges can be given to a group using the "GROUP" keyword.
 For example:
  GRANT SELECT ON foobar TO GROUP my_group;
 The keyword 'PUBLIC' is also supported to mean all users.

 Privileges can only be granted or revoked to one user or group
 at a time.

 "WITH GRANT OPTION" is not supported.  Only class owners can change
 access control
   - The default access control is to to grant users readonly access.
     You must explicitly grant insert/update access to users.  To
 change
     this, modify the line in
  src/backend/utils/acl.h
     that defines ACL_WORLD_DEFAULT

Bug fixes:
 * the bug where aggregates of empty tables were not run has been
 fixed. Now,
   aggregates run on empty tables will return the initial conditions
 of the
   aggregates. Thus, COUNT of an empty table will now properly return
 0.
   MAX/MIN of an empty table will return a tuple of value NULL.
 * allow the use of \; inside the monitor
 * the LISTEN/NOTIFY asynchronous notification mechanism now work
 * NOTIFY in rule action bodies now work
 * hash indices work, and access methods in general should perform
 better.
   creation of large btree indices should be much faster.  (thanks to
 Paul
   Aoki)

Other changes and enhancements:
 * addition of an EXPLAIN statement used for explaining the query
 execution
```

```
  plan (eg. "EXPLAIN SELECT * FROM EMP" prints out the execution plan
for
  the query).
* WARN and NOTICE messages no longer have timestamps on them. To turn
on
  timestamps of error messages, uncomment the line in
  src/backend/utils/elog.h:
/* define ELOG_TIMESTAMPS */
* On an access control violation, the message
"Either no such class or insufficient privilege"
  will be given.  This is the same message that is returned when
  a class is not found.  This dissuades non-privileged users from
  guessing the existence of privileged classes.
* some additional system catalog changes have been made that are not
  visible to the user.

libpgtcl changes:
* The -oid option has been added to the "pg_result" tcl command.
  pg_result -oid returns oid of the last tuple inserted.   If the
  last command was not an INSERT, then pg_result -oid returns "".
* the large object interface is available as pg_lo* tcl commands:
  pg_lo_open, pg_lo_close, pg_lo_creat, etc.

Portability enhancements and New Ports:
* flex/lex problems have been cleared up.  Now, you should be able to
use
  flex instead of lex on any platforms.  We no longer make
assumptions of
  what lexer you use based on the platform you use.
* The Linux-ELF port is now supported.  Various configuration have
been
  tested:  The following configuration is known to work:
kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
  with everything in ELF format,

New utilities:
* ipcclean added to the distribution
  ipcclean usually does not need to be run, but if your backend
crashes
  and leaves shared memory segments hanging around, ipcclean will
  clean them up for you.

New documentation:
* the user manual has been revised and libpq documentation added.
```

35.15. Postgres95 Beta 0.02

35.15.1. Detailed Change List

```
Incompatible changes:
* The SQL statement for creating a database is 'CREATE DATABASE'
instead
```

```
    of 'CREATEDB'. Similarly, dropping a database is 'DROP DATABASE'
  instead
    of 'DESTROYDB'. However, the names of the executables 'createdb'
  and
    'destroydb' remain the same.

New tools:
  * pgperl - a Perl (4.036) interface to Postgres95
  * pg_dump - a utility for dumping out a postgres database into a
  script file containing query commands. The script files are in a
  ASCII
  format and can be used to reconstruct the database, even on other
  machines and other architectures. (Also good for converting
  a Postgres 4.2 database to Postgres95 database.)

The following ports have been incorporated into postgres95-beta-0.02:
  * the NetBSD port by Alistair Crooks
  * the AIX port by Mike Tung
  * the Windows NT port by Jon Forrest (more stuff but not done yet)
  * the Linux ELF port by Brian Gallew

The following bugs have been fixed in postgres95-beta-0.02:
  * new lines not escaped in COPY OUT and problem with COPY OUT when
  first
    attribute is a '.'
  * cannot type return to use the default user id in createuser
  * SELECT DISTINCT on big tables crashes
  * Linux installation problems
  * monitor doesn't allow use of 'localhost' as PGHOST
  * psql core dumps when doing \c or \l
  * the "pgtclsh" target missing from src/bin/pgtclsh/Makefile
  * libpgtcl has a hard-wired default port number
  * SELECT DISTINCT INTO TABLE hangs
  * CREATE TYPE doesn't accept 'variable' as the internallength
  * wrong result using more than 1 aggregate in a SELECT
```

35.16. Postgres95 Beta 0.01

Initial release.

35.17. Timing Results

These timing results are from running the regression test with the commands

```
% cd src/test/regress
% make all
% time make runtest
```

Timing under Linux 2.0.27 seems to have a roughly 5% variation from run to run, presumably due to the scheduling vagaries of multitasking systems.

35.17.1. v6.4beta

The times for this release are not directly comparable to those for previous releases since some additional regression tests have been included. In general, however, v6.4 should be slightly faster than the previous release (thanks, Bruce!).

```
  Time   System
  02:26  Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc
 2.7.2.1 -O2 -m486
```

35.17.2. v6.3

The times for this release are not directly comparable to those for previous releases since some additional regression tests have been included and some obsolete tests involving time travel have been removed. In general, however, v6.3 is substantially faster than previous releases (thanks, Bruce!).

```
  Time   System
  02:30  Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc
 2.7.2.1 -O2 -m486
  04:12  Dual Pentium Pro 180, 96MB, EIDE, Linux 2.0.30, gcc 2.7.2.1 -
O2 -m486
```

35.17.3. v6.1

```
  Time   System
  06:12  Pentium Pro 180, 32MB, EIDE, Linux 2.0.30, gcc 2.7.2 -O2 -
m486
  12:06  P-100, 48MB, Linux 2.0.29, gcc
  39:58  Sparc IPC 32MB, Solaris 2.5, gcc 2.7.2.1 -O -g
```

Part IV. Programmer's Guide

Information for extending Postgres.

Table of Contents

Chapter 36. Introduction

This document is the programmer's manual for the PostgreSQL[1] database management system, originally developed at the University of California at Berkeley. PostgreSQL is based on Postgres release 4.2[2]. The Postgres project, led by Professor Michael Stonebraker, has been sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

The first part of this manual explains the Postgres approach to extensibility and describe how users can extend Postgres by adding user-defined types, operators, aggregates, and both query language and programming language functions. After a discussion of the Postgres rule system, we discuss the trigger and SPI interfaces. The manual concludes with a detailed description of the programming interfaces and support libraries for various languages.

We assume proficiency with UNIX and C programming.

36.1. Resources

This manual set is organized into several parts:

Tutorial

An introduction for new users. Does not cover advanced features.

User's Guide

General information for users, including available commands and data types.

Programmer's Guide

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

Administrator's Guide

Installation and management information. List of supported machines.

Developer's Guide

Information for Postgres developers. This is intended for those who are contributing to the Postgres project; application development information should appear in the *Programmer's Guide*. Currently included in the *Programmer's Guide*.

Reference Manual

Detailed reference information on command syntax. Currently included in the *User's Guide*.

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

man pages

The man pages have general information on command syntax.

[1] http://postgresql.org/
[2] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

FAQs

The Frequently Asked Questions (FAQ) documents address both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

Web Site

The Postgres[3] web site has some information not appearing in the distribution. There is a mhonarc catalog of mailing list traffic which is a rich resource for many topics.

Mailing Lists

The Postgres Questions[4] mailing list is a good place to have user questions answered. Other mailing lists are available; consult the web page for details.

Yourself!

Postgres is an open source product. As such, it depends on the user community for ongoing support. As you begin to use Postgres, 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 it. 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 Postgres Documentation[5] mailing list is the place to get going.

36.2. Terminology

In the following documentation, *site* may be interpreted as the host machine on which Postgres is installed. Since it is possible to install more than one set of Postgres databases on a single host, this term more precisely denotes any particular set of installed Postgres binaries and databases.

The Postgres *superuser* is the user named `postgres` who owns the Postgres binaries and database files. As the database superuser, all protection mechanisms may be bypassed and any data accessed arbitrarily. In addition, the Postgres superuser is allowed to execute some support programs which are generally not available to all users. Note that the Postgres superuser is *not* the same as the Unix superuser (which will be referred to as *root*). The superuser should have a non-zero user identifier (*UID*) for security reasons.

The *database administrator* or DBA, is the person who is responsible for installing Postgres with mechanisms to enforce a security policy for a site. The DBA can add new users by the method described below and maintain a set of template databases for use by createdb.

The postmaster is the process that acts as a clearing-house for requests to the Postgres system. Frontend applications connect to the postmaster, which keeps tracks of any system errors and communication between the backend processes. The postmaster can take several command-line arguments to tune its behavior. However, supplying arguments is necessary only if you intend to run multiple sites or a non-default site.

The Postgres backend (the actual executable program postgres) may be executed directly from the user shell by the Postgres super-user (with the database name as an argument). However, doing this bypasses

[3] postgresql.org
[4] mailto:questions@postgresql.org
[5] mailto:docs@postgresql.org

the shared buffer pool and lock table associated with a postmaster/site, therefore this is not recommended in a multiuser site.

36.3. Notation

"..." or `/usr/local/pgsql/` at the front of a file name is used to represent the path to the Postgres superuser's home directory.

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.

In examples, parentheses ("(" and ")") are used to group boolean expressions. "|" is the boolean operator OR.

Examples will show commands executed from various accounts and programs. Commands executed from the root account will be preceeded with ">". Commands executed from the Postgres superuser account will be preceeded with "%", while commands executed from an unprivileged user's account will be preceeded with "$". SQL commands will be preceeded with "=>" or will have no leading prompt, depending on the context.

Note

At the time of writing (Postgres v6.4) the notation for flagging commands is not universally consistant throughout the documentation set. Please report problems to the Documentation Mailing List[6].

36.4. Y2K Statement

Author

Written by Thomas Lockhart[7] on 1998-10-22.

The PostgreSQL Global Development Team provides the Postgres software code tree as a public service, without warranty and without liability for it's behavior or performance. However, at the time of writing:

- The author of this statement, a volunteer on the Postgres support team since November, 1996, is not aware of any problems in the Postgres code base related to time transitions around Jan 1, 2000 (Y2K).
- The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of Postgres. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.
- To the best of the author's knowledge, the assumptions Postgres makes about dates specified with a two-digit year are documented in the current User's Guide[8] in the chapter on data types. For two-digit years, the significant transition year is 1970, not 2000; e.g. "70-01-01" is interpreted as "1970-01-01", whereas "69-01-01" is interpreted as "2069-01-01".
- Any Y2K problems in the underlying OS related to obtaining "the current time" may propagate into apparent Y2K problems in Postgres.

[6] mailto:docs@postgresql.org
[7] mailto:lockhart@alumni.caltech.edu
[8] http://www.postgresql.org/docs/user/datatype.htm

Refer to The Gnu Project[9] and The Perl Institute[10] for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

36.5. Copyrights and Trademarks

[9] http://www.gnu.org/software/year2000.html
[10] http://language.perl.com/news/y2k.html

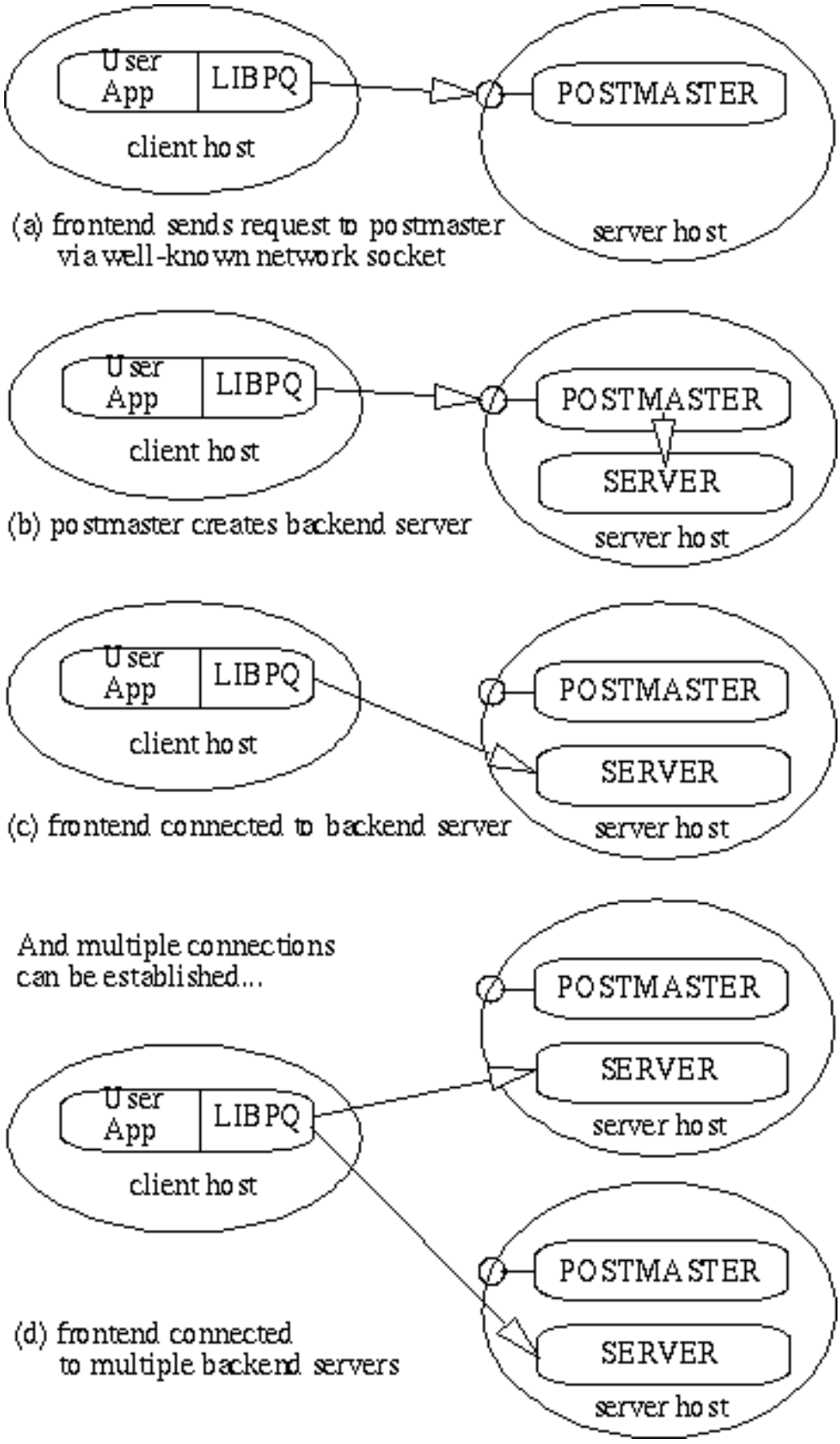
Chapter 37. Architecture

37.1. Postgres Architectural Concepts

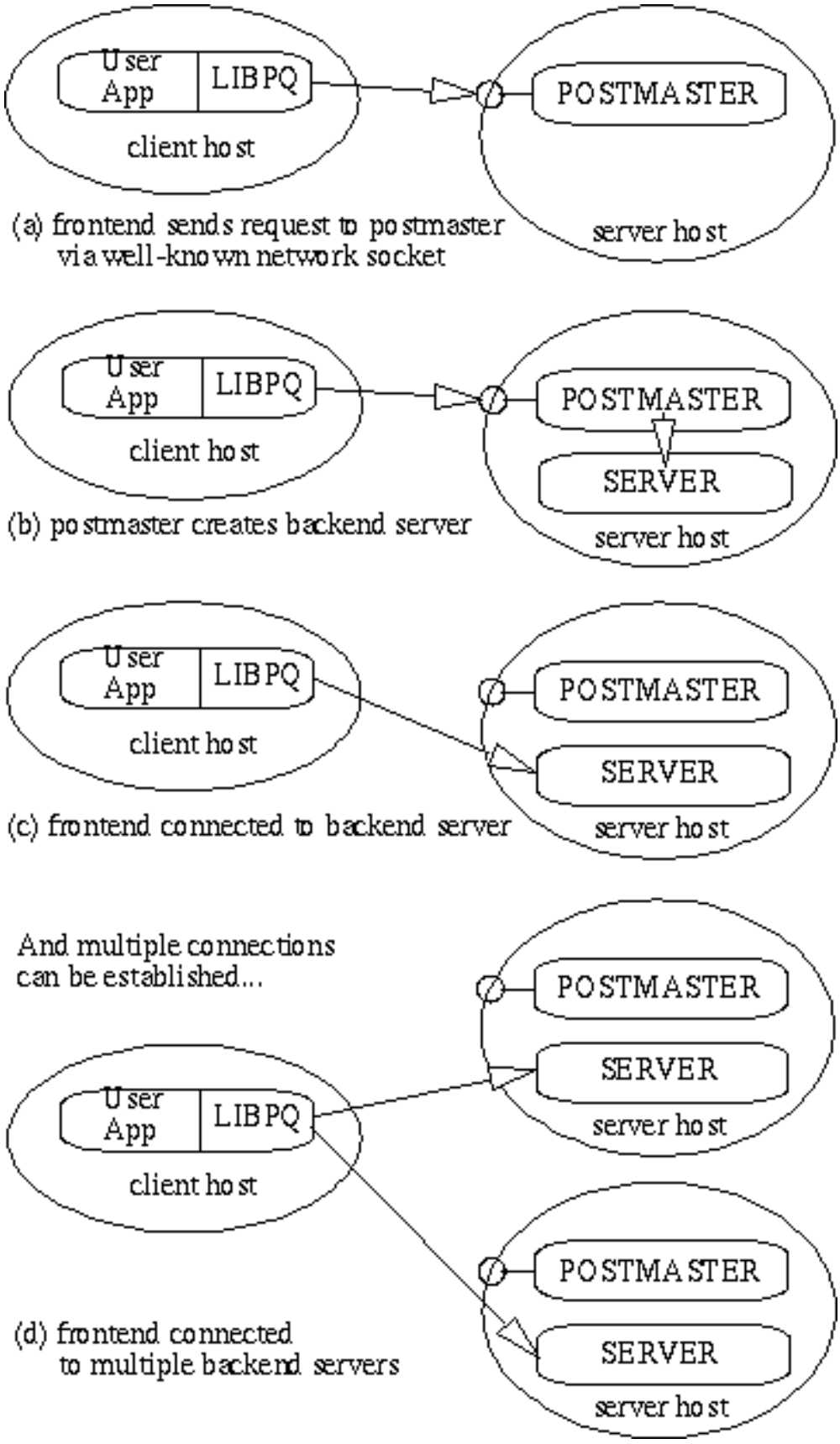
Before we continue, you should understand the basic Postgres system architecture. Understanding how the parts of Postgres interact will make the next chapter somewhat clearer. In database jargon, Postgres uses a simple "process per-user" client/server model. A Postgres session consists of the following cooperating UNIX processes (programs):

- A supervisory daemon process (postmaster),
- the user's frontend application (e.g., the psql program), and
- the one or more backend database servers (the postgres process itself).

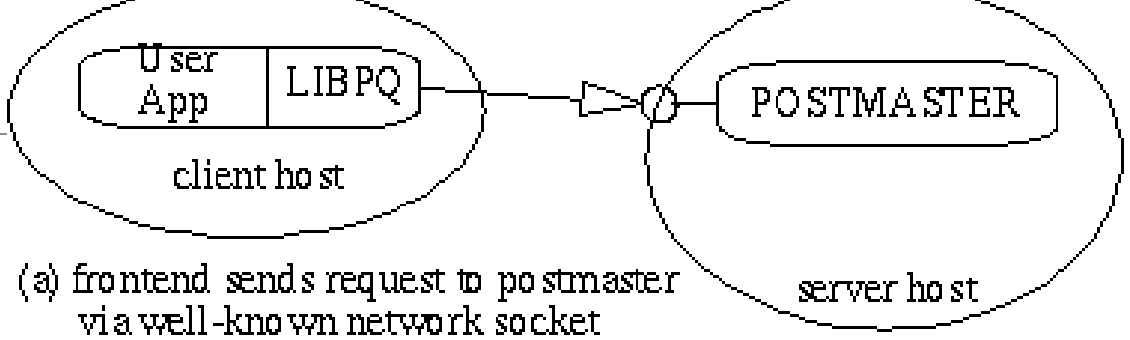
A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called an installation or site. Frontend applications that wish to access a given database within an installation make calls to the library. The library sends user requests over the network to the postmaster (


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

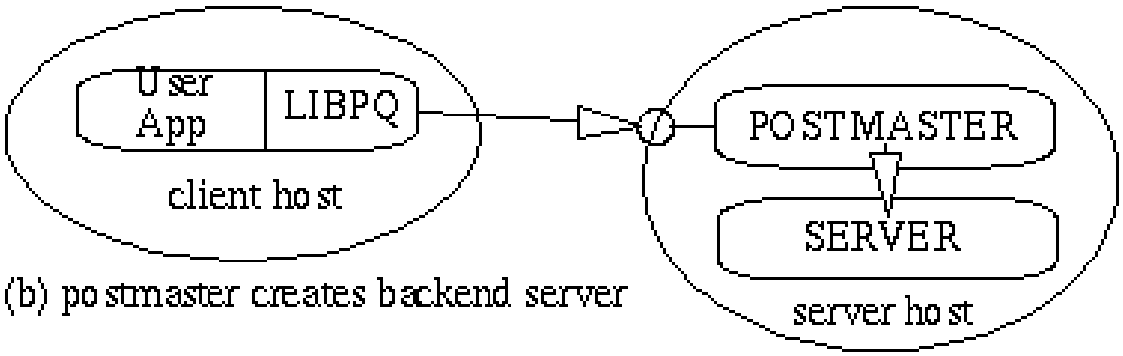
(a)), which in turn starts a new backend server process (


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

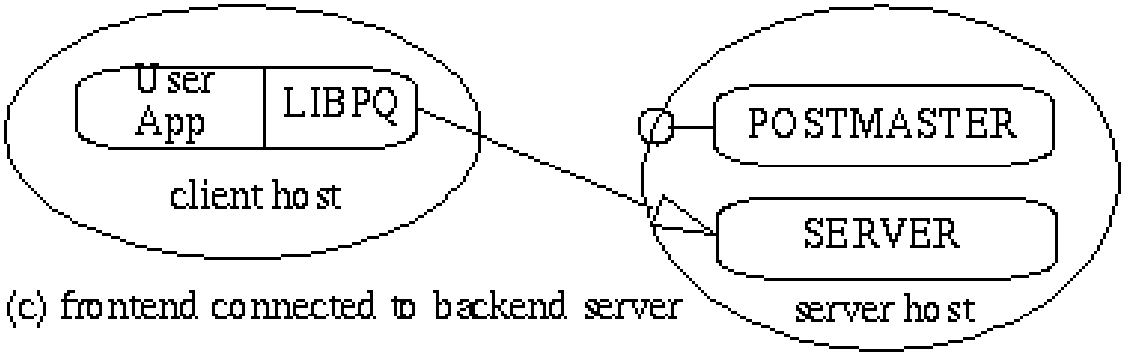
(b))



(a) frontend sends request to postmaster via well-known network socket

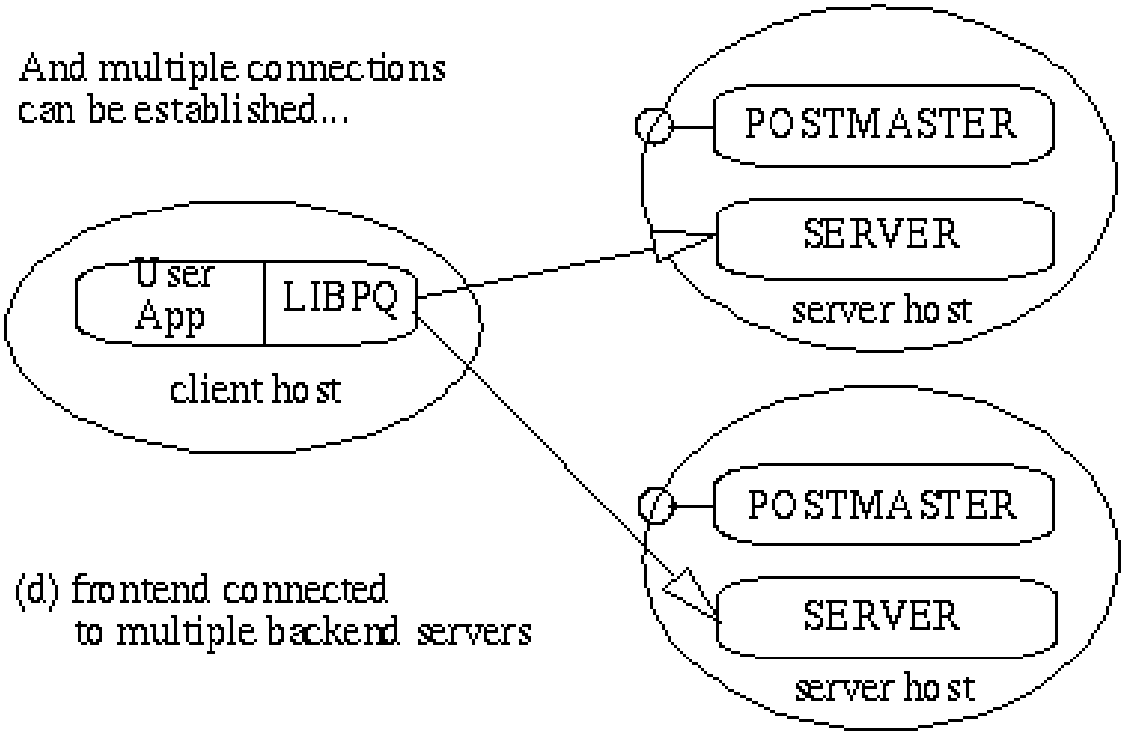


(b) postmaster creates backend server



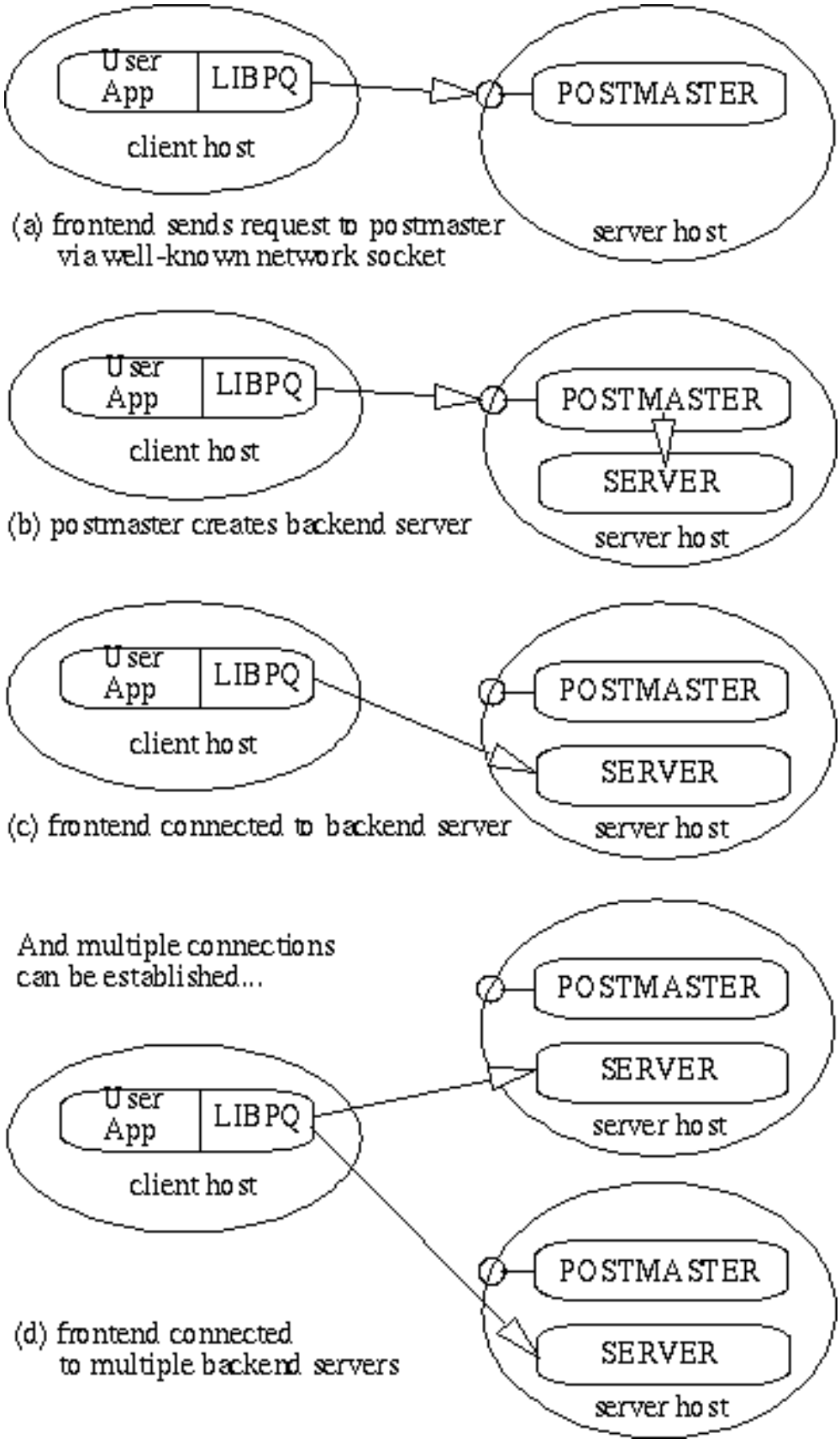
(c) frontend connected to backend server

And multiple connections can be established...



(d) frontend connected to multiple backend servers

and connects the frontend process to the new server (


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

(c)). From that point on, the frontend process and the backend server communicate without intervention by the postmaster. Hence, the postmaster is always running, waiting for requests, whereas frontend and backend processes come and go. The `libpq` library allows a single frontend to make multiple connections to backend processes. However, the frontend application is still a single-threaded process. Multithreaded frontend/backend connections are not currently supported in `libpq`. One implication of this architecture is that the postmaster and the backend always run on the same machine (the database server), while the frontend application may run anywhere. You should keep this in mind, because the files that can be accessed on a client machine may not be accessible (or may only be accessed using a different filename) on the database server machine. You should also be aware that the postmaster and postgres servers run with the user-id of the Postgres "superuser." Note that the Postgres superuser does not have to be a special user (e.g., a user named "postgres"), although many systems are installed that way. Furthermore, the Postgres superuser should definitely not be the UNIX superuser, "root"! In any case, all files relating to a database should belong to this Postgres superuser.

Chapter 38. Extending SQL: An Overview

In the sections that follow, we will discuss how you can extend the Postgres SQL query language by adding:

- functions
- types
- operators
- aggregates

38.1. How Extensibility Works

Postgres is extensible because its operation is catalog-driven. If you are familiar with standard relational systems, you know that they store information about databases, tables, columns, etc., in what are commonly known as system catalogs. (Some systems call this the data dictionary). The catalogs appear to the user as classes, like any other, but the DBMS stores its internal bookkeeping in them. One key difference between Postgres and standard relational systems is that Postgres stores much more information in its catalogs -- not only information about tables and columns, but also information about its types, functions, access methods, and so on. These classes can be modified by the user, and since Postgres bases its internal operation on these classes, this means that Postgres can be extended by users. By comparison, conventional database systems can only be extended by changing hardcoded procedures within the DBMS or by loading modules specially-written by the DBMS vendor.

Postgres is also unlike most other data managers in that the server can incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a compiled .o file or shared library) that implements a new type or function and Postgres will load it as required. Code written in SQL are even more trivial to add to the server. This ability to modify its operation "on the fly" makes Postgres uniquely suited for rapid prototyping of new applications and storage structures.

38.2. The Postgres Type System

The Postgres type system can be broken down in several ways. Types are divided into base types and composite types. Base types are those, like *int4*, that are implemented in a language such as C. They generally correspond to what are often known as "abstract data types"; Postgres can only operate on such types through methods provided by the user and only understands the behavior of such types to the extent that the user describes them. Composite types are created whenever the user creates a class. EMP is an example of a composite type.

Postgres stores these types in only one way (within the file that stores all instances of the class) but the user can "look inside" at the attributes of these types from the query language and optimize their retrieval by (for example) defining indices on the attributes. Postgres base types are further divided into built-in types and user-defined types. Built-in types (like *int4*) are those that are compiled into the system. User-defined types are those created by the user in the manner to be described below.

38.3. About the Postgres System Catalogs

Having introduced the basic extensibility concepts, we can now take a look at how the catalogs are actually laid out. You can skip this section for now, but some later sections will be incomprehensible without the

information given here, so mark this page for later reference. All system catalogs have names that begin with *pg_*. The following classes contain information that may be useful to the end user. (There are many other system catalogs, but there should rarely be a reason to query them directly.)

Table 38.1. Postgres System Catalogs

Catalog Name	Description
pg_database	databases
pg_class	classes
pg_attribute	class attributes
pg_index	secondary indices
pg_proc	procedures (both C and SQL)
pg_type	types (both base and complex)
pg_operator	operators
pg_aggregate	aggregates and aggregate functions
pg_am	access methods
pg_amop	access method operators
pg_amproc	access method support functions
pg_opclass	access method operator classes

Figure 38.1. The major Postgres system catalogs

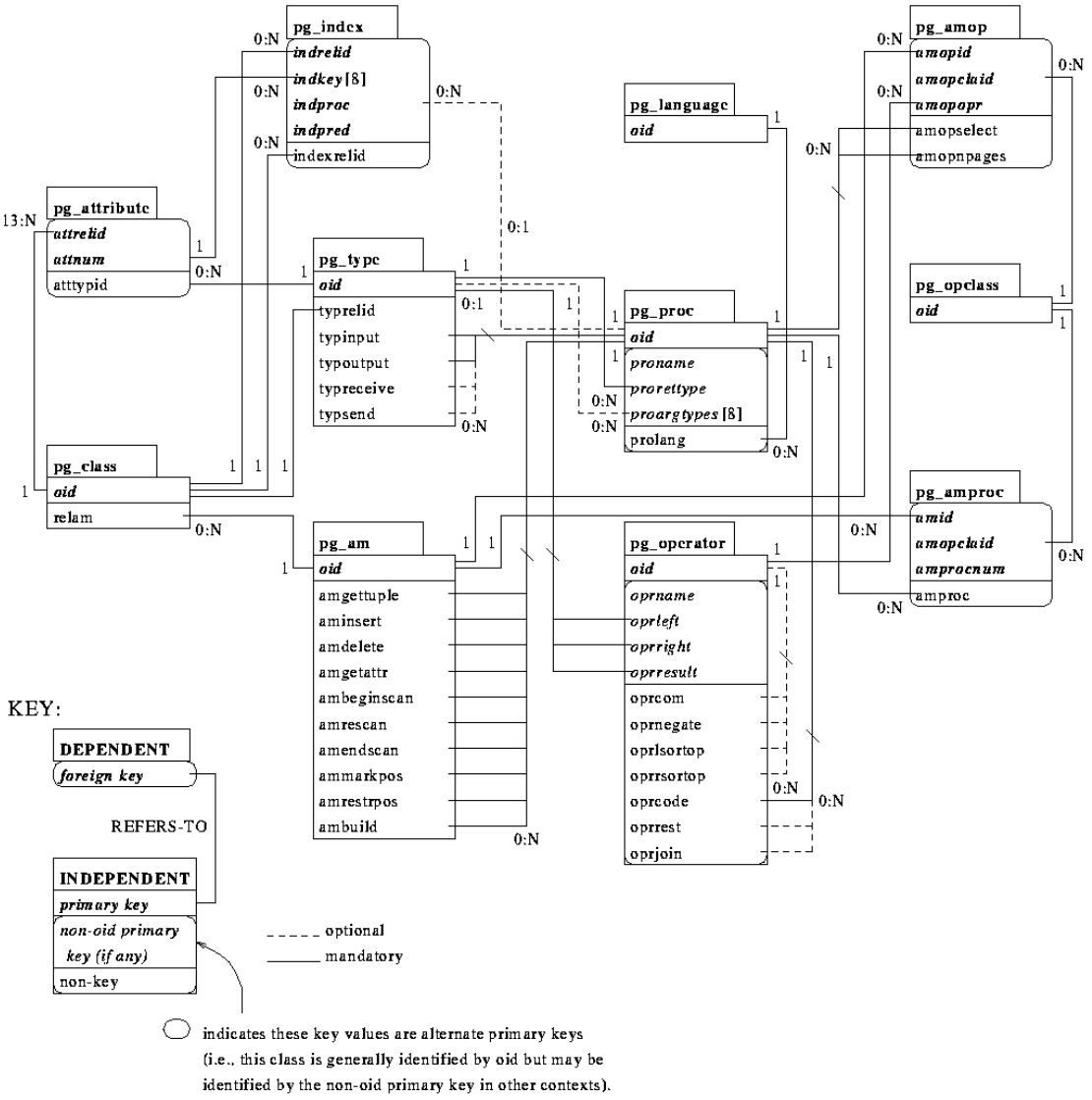


Figure 3. The major POSTGRES system catalogs.

The Reference Manual gives a more detailed explanation of these catalogs and their attributes. However,

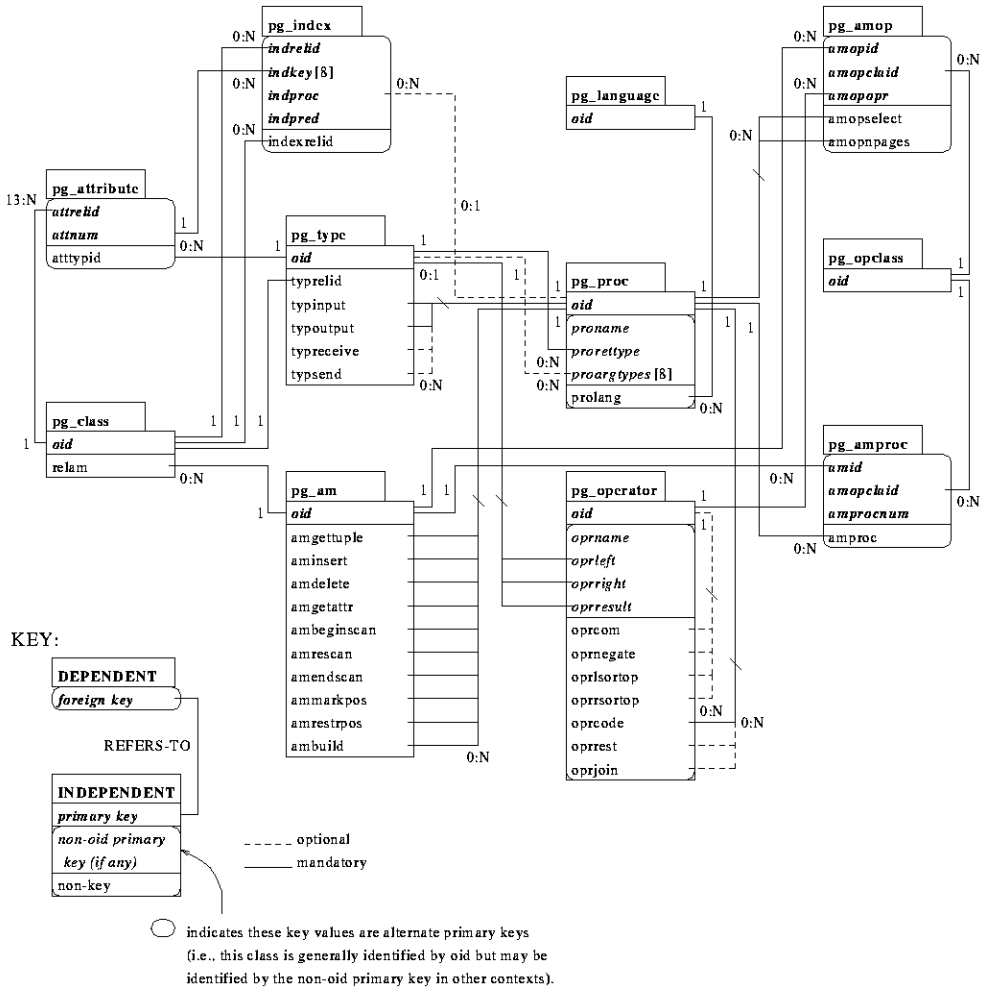


Figure 3. The major POSTGRES system catalogs.

shows the major entities and their relationships in the system catalogs. (Attributes that do not refer to other entities are not shown unless they are part of a primary key.) This diagram is more or less incomprehensible until you actually start looking at the contents of the catalogs and see how they relate to each other. For now, the main things to take away from this diagram are as follows:

- In several of the sections that follow, we will present various join queries on the system catalogs that display information we need to extend the system. Looking at this diagram should make some of these join queries (which are often three- or four-way joins) more understandable, because you will be able to see that the attributes used in the queries form foreign keys in other classes.
- Many different features (classes, attributes, functions, types, access methods, etc.) are tightly integrated in this schema. A simple create command may modify many of these catalogs.
- Types and procedures are central to the schema.

Note

We use the words *procedure* and *function* more or less interchangably.

Nearly every catalog contains some reference to instances in one or both of these classes. For example, Postgres frequently uses type signatures (e.g., of functions and operators) to identify unique instances of other catalogs.

- There are many attributes and relationships that have obvious meanings, but there are many (particularly those that have to do with access methods) that do not. The relationships between pg_am, pg_amop, pg_amproc, pg_operator and pg_opclass are particularly hard to understand and will be described in depth (in the section on interfacing types and operators to indices) after we have discussed basic extensions.

Chapter 39. Extending SQL: Functions

As it turns out, part of defining a new type is the definition of functions that describe its behavior. Consequently, while it is possible to define a new function without defining a new type, the reverse is not true. We therefore describe how to add new functions to Postgres before describing how to add new types. Postgres SQL provides two types of functions: query language functions (functions written in SQL and programming language functions (functions written in a compiled programming language such as C.) Either kind of function can take a base type, a composite type or some combination as arguments (parameters). In addition, both kinds of functions can return a base type or a composite type. It's easier to define SQL functions, so we'll start with those. Examples in this section can also be found in `funcs.sql` and `C-code/funcs.c`.

39.1. Query Language (SQL) Functions

39.1.1. SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as int4:

```
CREATE FUNCTION one() RETURNS int4
 AS 'SELECT 1 as RESULT' LANGUAGE 'sql';

SELECT one() AS answer;

     +-------+
     |answer |
     +-------+
     |1      |
     +-------+
```

Notice that we defined a target list for the function (with the name RESULT), but the target list of the query that invoked the function overrode the function's target list. Hence, the result is labelled answer instead of one.

It's almost as easy to define SQL functions that take base types as arguments. In the example below, notice how we refer to the arguments within the function as $1 and $2.

```
CREATE FUNCTION add_em(int4, int4) RETURNS int4
 AS 'SELECT $1 + $2;' LANGUAGE 'sql';

SELECT add_em(1, 2) AS answer;

     +-------+
     |answer |
     +-------+
     |3      |
     +-------+
```

39.1.2. SQL Functions on Composite Types

When specifying functions with arguments of composite types (such as EMP), we must not only specify which argument we want (as we did above with $1 and $2) but also the attributes of that argument. For example, take the function double_salary that computes what your salary would be if it were doubled.

```
CREATE FUNCTION double_salary(EMP) RETURNS int4
 AS 'SELECT $1.salary * 2 AS salary;' LANGUAGE 'sql';

SELECT name, double_salary(EMP) AS dream
 FROM EMP
 WHERE EMP.dept = 'toy';

     +-----+-------+
     |name | dream |
     +-----+-------+
     |Sam  | 2400  |
     +-----+-------+
```

Notice the use of the syntax $1.salary. Before launching into the subject of functions that return composite types, we must first introduce the function notation for projecting attributes. The simple way to explain this is that we can usually use the notation attribute(class) and class.attribute interchangably.

```
--
-- this is the same as:
--  SELECT EMP.name AS youngster FROM EMP WHERE EMP.age < 30
--
SELECT name(EMP) AS youngster
 FROM EMP
 WHERE age(EMP) < 30;

     +----------+
     |youngster |
     +----------+
     |Sam       |
     +----------+
```

As we shall see, however, this is not always the case. This function notation is important when we want to use a function that returns a single instance. We do this by assembling the entire instance within the function, attribute by attribute. This is an example of a function that returns a single EMP instance:

```
CREATE FUNCTION new_emp() RETURNS EMP
 AS 'SELECT \'None\'::text AS name,
  1000 AS salary,
  25 AS age,
  \'none\'::text AS dept;'
  LANGUAGE 'sql';
```

In this case we have specified each of the attributes with a constant value, but any computation or expression could have been substituted for these constants. Defining a function like this can be tricky. Some of the more important caveats are as follows:

- The target list order must be exactly the same as that in which the attributes appear in the CREATE TABLE statement (or when you execute a .* query).
- You must typecast the expressions (using ::) very carefully or you will see the following error:

```
            WARN::function declared to return type EMP does not
 retrieve (EMP.*)
```

- When calling a function that returns an instance, we cannot retrieve the entire instance. We must either project an attribute out of the instance or pass the entire instance into another function.

```
SELECT name(new_emp()) AS nobody;

        +-------+
        |nobody |
        +-------+
        |None   |
        +-------+
```

- The reason why, in general, we must use the function syntax for projecting attributes of function return values is that the parser just doesn't understand the other (dot) syntax for projection when combined with function calls.

```
SELECT new_emp().name AS nobody;
WARN:parser: syntax error at or near "."
```

Any collection of commands in the SQL query language can be packaged together and defined as a function. The commands can include updates (i.e., insert, update and delete) as well as select queries. However, the final command must be a select that returns whatever is specified as the function's returntype.

```
CREATE FUNCTION clean_EMP () RETURNS int4
 AS 'DELETE FROM EMP WHERE EMP.salary <= 0;
SELECT 1 AS ignore_this'
 LANGUAGE 'sql';

SELECT clean_EMP();

     +--+
     |x |
     +--+
     |1 |
     +--+
```

39.2. Programming Language Functions

39.2.1. Programming Language Functions on Base Types

Internally, Postgres regards a base type as a "blob of memory." The user-defined functions that you define over a type in turn define the way that Postgres can operate on it. That is, Postgres will only store and retrieve the data from disk and use your user-defined functions to input, process, and output the data. Base types can have one of three internal formats:

- pass by value, fixed-length
- pass by reference, fixed-length
- pass by reference, variable-length

By-value types can only be 1, 2 or 4 bytes in length (even if your computer supports by-value types of other sizes). Postgres itself only passes integer types by value. You should be careful to define your types such that they will be the same size (in bytes) on all architectures. For example, the long type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas int type is 4 bytes on most UNIX

machines (though not on most personal computers). A reasonable implementation of the int4 type on UNIX machines might be:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

On the other hand, fixed-length types of any size may be passed by-reference. For example, here is a sample implementation of a Postgres type:

```
/* 16-byte structure, passed by reference */
typedef struct {
    char data[16];
} char16;
```

Only pointers to such types can be used when passing them in and out of Postgres functions. Finally, all variable-length types must also be passed by reference. All variable-length types must begin with a length field of exactly 4 bytes, and all data to be stored within that type must be located in the memory immediately following that length field. The length field is the total length of the structure (i.e., it includes the size of the length field itself). We can define the text type as follows:

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

Obviously, the data field is not long enough to hold all possible strings -- it's impossible to declare such a structure in C. When manipulating variable-length types, we must be careful to allocate the correct amount of memory and initialize the length field. For example, if we wanted to store 40 bytes in a text structure, we might use a code fragment like this:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memmove(destination->data, buffer, 40);
...
```

Now that we've gone over all of the possible structures for base types, we can show some examples of real functions. Suppose `funcs.c` look like:

```
#include <string.h>
#include "postgres.h"
int
add_one(int arg)
{
    return(arg + 1);
}
text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) -
VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);
```

```
            memset((void *) new_text, 0, new_text_size);
            VARSIZE(new_text) = new_text_size;
            strncpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-
VARHDRSZ);
            strncat(VARDATA(new_text), VARDATA(arg2), VARSIZE(arg2)-
VARHDRSZ);
            return (new_text);
        }
        text *
        copytext(text *t)
        {
            /*
             * VARSIZE is the total size of the struct in bytes.
             */
            text *new_t = (text *) palloc(VARSIZE(t));
            memset(new_t, 0, VARSIZE(t));
            VARSIZE(new_t) = VARSIZE(t);
            /*
             * VARDATA is a pointer to the data region of the struct.
             */
            memcpy((void *) VARDATA(new_t), /* destination */
                   (void *) VARDATA(t),     /* source */
                   VARSIZE(t)-VARHDRSZ);        /* how many bytes */
            return(new_t);
        }
```

On OSF/1 we would type:

```
CREATE FUNCTION add_one(int4) RETURNS int4
     AS 'PGROOT/tutorial/obj/funcs.so' LANGUAGE 'c';

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'PGROOT/tutorial/obj/funcs.so' LANGUAGE 'c';

CREATE FUNCTION copytext(text) RETURNS text
     AS 'PGROOT/tutorial/obj/funcs.so' LANGUAGE 'c';
```

On other systems, we might have to make the filename end in .sl (to indicate that it's a shared library).

39.2.2. Programming Language Functions on Composite Types

Composite types do not have a fixed layout like C structures. Instances of a composite type may contain null fields. In addition, composite types that are part of an inheritance hierarchy may have different fields than other members of the same inheritance hierarchy. Therefore, Postgres provides a procedural interface for accessing fields of composite types from C. As Postgres processes a set of instances, each instance will be passed into your function as an opaque structure of type TUPLE. Suppose we want to write a function to answer the query

```
* SELECT name, c_overpaid(EMP, 1500) AS overpaid
  FROM EMP
  WHERE name = 'Bill' or name = 'Sam';
```

In the query above, we can define c_overpaid as:

```
#include "postgres.h"
#include "libpq-fe.h" /* for TUPLE */
bool
c_overpaid(TUPLE t,/* the current instance of EMP */
           int4 limit)
{
    bool isnull = false;
    int4 salary;
    salary = (int4) GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        return (false);
    return(salary > limit);
}
```

GetAttributeByName is the Postgres system function that returns attributes out of the current instance. It has three arguments: the argument of type TUPLE passed into the function, the name of the desired attribute, and a return parameter that describes whether the attribute is null. GetAttributeByName will align data properly so you can cast its return value to the desired type. For example, if you have an attribute name which is of the type name, the GetAttributeByName call would look like:

```
char *str;
...
str = (char *) GetAttributeByName(t, "name", &isnull)
```

The following query lets Postgres know about the c_overpaid function:

```
* CREATE FUNCTION c_overpaid(EMP, int4) RETURNS bool
     AS 'PGROOT/tutorial/obj/funcs.so' LANGUAGE 'c';
```

While there are ways to construct new instances or modify existing instances from within a C function, these are far too complex to discuss in this manual.

39.2.3. Caveats

We now turn to the more difficult task of writing programming language functions. Be warned: this section of the manual will not make you a programmer. You must have a good understanding of C (including the use of pointers and the malloc memory manager) before trying to write C functions for use with Postgres. While it may be possible to load functions written in languages other than C into Postgres, this is often difficult (when it is possible at all) because other languages, such as FORTRAN and Pascal often do not follow the same "calling convention" as C. That is, other languages do not pass argument and return values between functions in the same way. For this reason, we will assume that your programming language functions are written in C. The basic rules for building C functions are as follows:

- Most of the header (include) files for Postgres should already be installed in `PGROOT/include` (see Figure 2). You should always include

  ```
  -I$PGROOT/include
  ```

  on your cc command lines. Sometimes, you may find that you require header files that are in the server source itself (i.e., you need a file we neglected to install in include). In those cases you may need to add one or more of

  ```
  -I$PGROOT/src/backend
  -I$PGROOT/src/backend/include
  -I$PGROOT/src/backend/port/<PORTNAME>
  ```

```
-I$PGROOT/src/backend/obj
```

(where <PORTNAME> is the name of the port, e.g., alpha or sparc).

- When allocating memory, use the Postgres routines palloc and pfree instead of the corresponding C library routines malloc and free. The memory allocated by palloc will be freed automatically at the end of each transaction, preventing memory leaks.
- Always zero the bytes of your structures using memset or bzero. Several routines (such as the hash access method, hash join and the sort algorithm) compute functions of the raw bits contained in your structure. Even if you initialize all fields of your structure, there may be several bytes of alignment padding (holes in the structure) that may contain garbage values.
- Most of the internal Postgres types are declared in postgres.h, so it's a good idea to always include that file as well. Including postgres.h will also include elog.h and palloc.h for you.
- Compiling and loading your object code so that it can be dynamically loaded into Postgres always requires special flags. See Appendix A for a detailed explanation of how to do it for your particular operating system.

Chapter 40. Extending SQL: Types

As previously mentioned, there are two kinds of types in Postgres: base types (defined in a programming language) and composite types (instances). Examples in this section up to interfacing indices can be found in `complex.sql` and `complex.c`. Composite examples are in `funcs.sql`.

40.1. User-Defined Types

40.1.1. Functions Needed for a User-Defined Type

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-delimited character string as its input and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type and returns a null delimited character string. Suppose we want to define a complex type which represents complex numbers. Naturally, we choose to represent a complex in memory as the following C structure:

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

and a string of the form (x,y) as the external string representation. These functions are usually not hard to write, especially the output function. However, there are a number of points to remember:

- When defining your external (string) representation, remember that you must eventually write a complete and robust parser for that representation as your input function!

```
Complex *
complex_in(char *str)
{
    double x, y;
    Complex *result;
    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
{

        elog(WARN, "complex_in: error in parsing
        return NULL;
    }
    result = (Complex *)palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    return (result);
}
```

The output function can simply be:

```
char *
complex_out(Complex *complex)
{
    char *result;
    if (complex == NULL)
        return(NULL);
```

```
                    result = (char *) palloc(60);
                    sprintf(result, "(%g,%g)", complex->x, complex-
>y);
                    return(result);
                }
```

- You should try to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in (say, into someone else's database on another computer). This is a particularly common problem when floating-point numbers are involved.

To define the complex type, we need to create the two user-defined functions complex_in and complex_out before creating the type:

```
CREATE FUNCTION complex_in(opaque)
   RETURNS complex
   AS 'PGROOT/tutorial/obj/complex.so'
   LANGUAGE 'c';

CREATE FUNCTION complex_out(opaque)
   RETURNS opaque
   AS 'PGROOT/tutorial/obj/complex.so'
   LANGUAGE 'c';

CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out
);
```

As discussed earlier, Postgres fully supports arrays of base types. Additionally, Postgres supports arrays of user-defined types as well. When you define a type, Postgres automatically provides support for arrays of that type. For historical reasons, the array type has the same name as the user-defined type with the underscore character _ prepended. Composite types do not need any function defined on them, since the system already understands what they look like inside.

40.1.2. Large Objects

The types discussed to this point are all "small" objects -- that is, they are smaller than 8KB in size.

Note

1024 longwords == 8192 bytes. In fact, the type must be considerably smaller than 8192 bytes, since the Postgres tuple and page overhead must also fit into this 8KB limitation. The actual value that fits depends on the machine architecture.

If you require a larger type for something like a document retrieval system or for storing bitmaps, you will need to use the Postgres large object interface.

Chapter 41. Extending SQL: Operators

Postgres supports left unary, right unary and binary operators. Operators can be overloaded, or re-used with different numbers and types of arguments. If there is an ambiguous situation and the system cannot determine the correct operator to use, it will return an error and you may have to typecast the left and/or right operands to help it understand which operator you meant to use. To create an operator for adding two complex numbers can be done as follows. First we need to create a function to add the new types. Then, we can create the operator with the function.

```
CREATE FUNCTION complex_add(complex, complex)
   RETURNS complex
   AS '$PWD/obj/complex.so'
   LANGUAGE 'c';

CREATE OPERATOR + (
   leftarg = complex,
   rightarg = complex,
   procedure = complex_add,
   commutator = +
);
```

We've shown how to create a binary operator here. To create unary operators, just omit one of leftarg (for left unary) or rightarg (for right unary). If we give the system enough type information, it can automatically figure out which operators to use.

```
SELECT (a + b) AS c FROM test_complex;

+----------------+
|c               |
+----------------+
|(5.2,6.05)      |
+----------------+
|(133.42,144.95) |
+----------------+
```

Chapter 42. Extending SQL: Aggregates

Aggregates in Postgres are expressed in terms of state transition functions. That is, an aggregate can be defined in terms of state that is modified whenever an instance is processed. Some state functions look at a particular value in the instance when computing the new state (sfunc1 in the create aggregate syntax) while others only keep track of their own internal state (sfunc2). If we define an aggregate that uses only sfunc1, we define an aggregate that computes a running function of the attribute values from each instance. "Sum" is an example of this kind of aggregate. "Sum" starts at zero and always adds the current instance's value to its running total. We will use the int4pl that is built into Postgres to perform this addition.

```
CREATE AGGREGATE complex_sum (
    sfunc1 = complex_add,
    basetype = complex,
    stype1 = complex,
    initcond1 = '(0,0)'
);

SELECT complex_sum(a) FROM test_complex;

         +------------+
         |complex_sum |
         +------------+
         |(34,53.9)   |
         +------------+
```

If we define only sfunc2, we are specifying an aggregate that computes a running function that is independent of the attribute values from each instance. "Count" is the most common example of this kind of aggregate. "Count" starts at zero and adds one to its running total for each instance, ignoring the instance value. Here, we use the built-in int4inc routine to do the work for us. This routine increments (adds one to) its argument.

```
CREATE AGGREGATE my_count (
    sfunc2 = int4inc, -- add one
    basetype = int4,
    stype2 = int4,
    initcond2 = '0'
);

SELECT my_count(*) as emp_count from EMP;

         +----------+
         |emp_count |
         +----------+
         |5         |
         +----------+
```

"Average" is an example of an aggregate that requires both a function to compute the running sum and a function to compute the running count. When all of the instances have been processed, the final answer for the aggregate is the running sum divided by the running count. We use the int4pl and int4inc routines we used before as well as the Postgres integer division routine, int4div, to compute the division of the sum by the count.

```
CREATE AGGREGATE my_average (
    sfunc1 = int4pl,     --  sum
    basetype = int4,
    stype1 = int4,
    sfunc2 = int4inc,    -- count
    stype2 = int4,
    finalfunc = int4div, -- division
    initcond1 = '0',
    initcond2 = '0'
);

SELECT my_average(salary) as emp_average FROM EMP;

         +------------+
         |emp_average |
         +------------+
         |1640        |
         +------------+
```

Chapter 43. The Postgres Rule System

Production rule systems are conceptually simple, but there are many subtle points involved in actually using them. Some of these points and the theoretical foundations of the Postgres rule system can be found in [Stonebraker et al, ACM, 1990].

Some other database systems define active database rules. These are usually stored procedures and triggers and are implemented in Postgres as functions and triggers.

The query rewrite rule system (the "rule system" from now on) is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query optimizer for execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The power of this rule system is discussed in [Ong and Goh, 1990] as well as [Stonebraker et al, ACM, 1990].

43.1. What is a Querytree?

To understand how the rule system works it is necessary to know when it is invoked and what it's input and results are.

The rule system is located between the query parser and the optimizer. It takes the output of the parser, one querytree, and the rewrite rules from the `pg_rewrite` catalog, which are querytrees too with some extra information, and creates zero or many querytrees as result. So it's input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a querytree? It is an internal representation of an SQL statement where the single parts that built it are stored separately. These querytrees are visible when starting the Postgres backend with debuglevel 4 and typing queries into the interactive backend interface. The rule actions in the `pg_rewrite` system catalog are also stored as querytrees. They are not formatted like the debug output, but they contain exactly the same information.

Reading a querytree requires some experience and it was a hard time when I started to work on the rule system. I can remember that I was standing at the coffee machine and I saw the cup in a targetlist, water and coffee powder in a rangetable and all the buttons in a qualification expression. Since SQL representations of querytrees are sufficient to understand the rule system, this document will not teach how to read them. It might help to learn it and the naming conventions are required in the later following descriptions.

43.1.1. The Parts of a Querytree

When reading the SQL representations of the querytrees in this document it is necessary to be able to identify the parts the statement is broken into when it is in the querytree structure. The parts of a querytree are

the commandtype

> This is a simple value telling which command (SELECT, INSERT, UPDATE, DELETE) produced the parsetree.

the rangetable

> The rangtable is a list of relations that are used in the query. In a SELECT statement that are the relations given after the FROM keyword.

Every rangetable entry identifies a table or view and tells by which name it is called in the other parts of the query. In the querytree the rangetable entries are referenced by index rather than by name, so here it doesn't matter if there are duplicate names as it would in an SQL statement. This can happen after the rangetables of rules have been merged in. The examples in this document will not have this situation.

the resultrelation

This is an index into the rangetable that identifies the relation where the results of the query go.

SELECT queries normally don't have a result relation. The special case of a SELECT INTO is mostly identical to a CREATE TABLE, INSERT ... SELECT sequence and is not discussed separately here.

On INSERT, UPDATE and DELETE queries the resultrelation is the table (or view!) where the changes take effect.

the targetlist

The targetlist is a list of expressions that define the result of the query. In the case of a SELECT, the expressions are what builds the final output of the query. They are the expressions between the SELECT and the FROM keywords (* is just an abbreviation for all the attribute names of a relation).

DELETE queries don't need a targetlist because they don't produce any result. In fact the optimizer will add a special entry to the empty targetlist. But this is after the rule system and will be discussed later. For the rule system the targetlist is empty.

In INSERT queries the targetlist describes the new rows that should go into the resultrelation. Missing columns of the resultrelation will be added by the optimizer with a constant NULL expression. It are the expressions in the VALUES clause or the ones from the SELECT clause on INSERT ... SELECT.

On UPDATE queries, it describes the new rows that should replace the old ones. Here now the optimizer will add missing columns by inserting expressions that put the values from the old rows into the new one. And it will add the special entry like for DELETE too. It are the expressions from the SET attribute = expression part of the query.

Every entry in the targetlist contains an expression that can be a constant value, a variable pointing to an attribute of one of the relations in the rangetable, a parameter or an expression tree made of function calls, constants, variables, operators etc.

the qualification

The queries qualification is an expression much like one of those contained in the targetlist entries. The result value of this expression is a boolean that tells if the operation (INSERT, UPDATE, DELETE or SELECT) for the final result row should be executed or not. It is the WHERE clause of an SQL statement.

the others

The other parts of the querytree like the ORDER BY clause arent of interest here. The rule system substitutes entries there while applying rules, but that doesn't have much to do with the fundamentals of the rule system. GROUP BY is a special thing when it appears in a view definition and still needs to be documented.

43.2. Views and the Rule System

43.2.1. Implementation of Views in Postgres

Views in Postgres are implemented using the rule system. In fact there is absolutely no difference between a

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

compared against the two commands

```
CREATE TABLE myview (same attribute list as for mytab);
CREATE RULE "_RETmyview" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

because this is exactly what the CREATE VIEW command does internally. This has some side effects. One of them is that the information about a view in the Postgres system catalogs is exactly the same as it is for a table. So for the query parsers, there is absolutely no difference between a table and a view. They are the same thing - relations. That is the important one for now.

43.2.2. How SELECT Rules Work

Rules ON SELECT are applied to all queries as the last step, even if the command given is an INSERT, UPDATE or DELETE. And they have different semantics from the others in that they modify the parsetree in place instead of creating a new one. So SELECT rules are described first.

Currently, there could be only one action and it must be a SELECT action that is INSTEAD. This restriction was required to make rules safe enough to open them for ordinary users and it restricts rules ON SELECT to real view rules.

The example for this document are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for INSERT, UPDATE and DELETE operations so that the final result will be a view that behaves like a real table with some magic functionality. It is not such a simple example to start from and this makes things harder to get into. But it's better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

The database needed to play on the examples is named al_bundy. You'll see soon why this is the database name. And it needs the procedural language PL/pgSQL installed, because we need a little min() function returning the lower of 2 integer values. We create that as

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS
    'BEGIN
        IF $1 < $2 THEN
            RETURN $1;
        END IF;
        RETURN $2;
    END;'
LANGUAGE 'plpgsql';
```

The real tables we need in the first two rule system descripitons are these:

```
CREATE TABLE shoe_data (
    shoename   char(10),      -- primary key
    sh_avail   integer,       -- available # of pairs
    slcolor    char(10),      -- preferred shoelace color
    slminlen   float,         -- miminum shoelace length
    slmaxlen   float,         -- maximum shoelace length
    slunit     char(8)        -- length unit
);
```

```
CREATE TABLE shoelace_data (
    sl_name    char(10),      -- primary key
    sl_avail   integer,       -- available # of pairs
    sl_color   char(10),      -- shoelace color
    sl_len     float,         -- shoelace length
    sl_unit    char(8)        -- length unit
);

CREATE TABLE unit (
    un_name    char(8),       -- the primary key
    un_fact    float          -- factor to transform to cm
);
```

I think most of us wear shoes and can realize that this is really useful data. Well there are shoes out in the world that don't require shoelaces, but this doesn't make Al's life easier and so we ignore it.

The views are created as

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

The CREATE VIEW command for the `shoelace` view (which is the simplest one we have) will create a relation shoelace and an entry in `pg_rewrite` that tells that there is a rewrite rule that must be applied whenever the relation shoelace is referenced in a queries rangetable. The rule has no rule qualification

(discussed in the non SELECT rules since SELECT rules currently cannot have them) and it is INSTEAD. Note that rule qualifications are not the same as query qualifications! The rules action has a qualification.

The rules action is one querytree that is an exact copy of the SELECT statement in the view creation command.

Note

The two extra range table entries for NEW and OLD (named *NEW* and *CURRENT* for historical reasons in the printed querytree) you can see in the `pg_rewrite` entry aren't of interest for SELECT rules.

Now we populate `unit`, `shoe_data` and `shoelace_data` and Al types the first SELECT in his life:

```
al_bundy=> INSERT INTO unit VALUES ('cm', 1.0);
al_bundy=> INSERT INTO unit VALUES ('m', 100.0);
al_bundy=> INSERT INTO unit VALUES ('inch', 2.54);
al_bundy=>
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh1', 2, 'black', 70.0, 90.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh2', 0, 'black', 30.0, 40.0, 'inch');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
al_bundy=>
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl1', 5, 'black', 80.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl2', 6, 'black', 100.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl3', 0, 'black', 35.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl4', 8, 'black', 40.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl5', 4, 'brown', 1.0 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl6', 0, 'brown', 0.9 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl7', 7, 'brown', 60 , 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl8', 1, 'brown', 40 , 'inch');
al_bundy=>
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       7|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
```

```
(8 rows)
```

It's the simplest SELECT Al can do on our views, so we take this to explain the basics of view rules. The 'SELECT * FROM shoelace' was interpreted by the parser and produced the parsetree

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

and this is given to the rule system. The rule system walks through the rangetable and checks if there are rules in `pg_rewrite` for any relation. When processing the rangetable entry for `shoelace` (the only one up to now) it finds the rule '_RETshoelace' with the parsetree

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       float8mul(s.sl_len, u.un_fact) AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*,
       shoelace_data s, unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

Note that the parser changed the calculation and qualification into calls to the appropriate functions. But in fact this changes nothing. The first step in rewriting is merging the two rangetables. The resulting parsetree then reads

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data s,
       unit u;
```

In step 2 it adds the qualification from the rule action to the parsetree resulting in

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data s,
       unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

And in step 3 it replaces all the variables in the parsetree, that reference the rangetable entry (the one for `shoelace` that is currently processed) by the corresponding targetlist expressions from the rule action. This results in the final query

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len,
       s.sl_unit, float8mul(s.sl_len, u.un_fact) AS sl_len_cm
  FROM shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data s,
       unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

Turning this back into a real SQL statement a human user would type reads

```
SELECT s.sl_name, s.sl_avail,
```

```
        s.sl_color, s.sl_len,
        s.sl_unit, s.sl_len * u.un_fact AS sl_len_cm
   FROM shoelace_data s, unit u
  WHERE s.sl_unit = u.un_name;
```

That was the first rule applied. While this was done, the rangetable has grown. So the rule system continues checking the range table entries. The next one is number 2 (shoelace *OLD*). Relation `shoelace` has a rule, but this rangetable entry isn't referenced in any of the variables of the parsetree, so it is ignored. Since all the remaining rangetable entries either have no rules in `pg_rewrite` or aren't referenced, it reaches the end of the rangetable. Rewriting is complete and the above is the final result given into the optimizer. The optimizer ignores the extra rangetable entries that aren't referenced by variables in the parsetree and the plan produced by the planner/optimizer would be exactly the same as if Al had typed the above SELECT query instead of the view selection.

Now we face Al with the problem that the Blues Brothers appear in his shop and want to buy some new shoes, and as the Blues Brothers are, they want to wear the same shoes. And they want to wear them immediately, so they need shoelaces too.

Al needs to know for which shoes currently in the store he has the matching shoelaces (color and size) and where the total number of exactly matching pairs is greater or equal to two. We theach him how to do and he asks his database:

```
al_bundy=> SELECT * FROM shoe_ready WHERE total_avail >= 2;
shoename  |sh_avail|sl_name   |sl_avail|total_avail
----------+--------+----------+--------+-----------
sh1       |       2|sl1       |       5|          2
sh3       |       4|sl7       |       7|          4
(2 rows)
```

Al is a shoe guru and so he knows that only shoes of type sh1 would fit (shoelace sl7 is brown and shoes that need brown shoelaces aren't shoes the Blues Brothers would ever wear).

The output of the parser this time is the parsetree

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

The first rule applied will be that one for the `shoe_ready` relation and it results in the parsetree

```
SELECT rsh.shoename, rsh.sh_avail,
       rsl.sl_name, rsl.sl_avail,
       min(rsh.sh_avail, rsl.sl_avail) AS total_avail
  FROM shoe_ready shoe_ready, shoe_ready *OLD*,
       shoe_ready *NEW*, shoe rsh,
       shoelace rsl
 WHERE int4ge(min(rsh.sh_avail, rsl.sl_avail), 2)
   AND (bpchareq(rsl.sl_color, rsh.slcolor)
        AND float8ge(rsl.sl_len_cm, rsh.slminlen_cm)
        AND float8le(rsl.sl_len_cm, rsh.slmaxlen_cm)
       );
```

In reality the AND clauses in the qualification will be operator nodes of type AND with a left and right expression. But that makes it lesser readable as it already is, and there are more rules to apply. So I only

put them into some parantheses to group them into logical units in the order they where added and we continue with the rule for relation `shoe` as it is the next rangetable entry that is referenced and has a rule. The result of applying it is

```
SELECT sh.shoename, sh.sh_avail,
       rsl.sl_name, rsl.sl_avail,
       min(sh.sh_avail, rsl.sl_avail) AS total_avail,
  FROM shoe_ready shoe_ready, shoe_ready *OLD*,
       shoe_ready *NEW*, shoe rsh,
       shoelace rsl, shoe *OLD*,
       shoe *NEW*, shoe_data sh,
       unit un
 WHERE (int4ge(min(sh.sh_avail, rsl.sl_avail), 2)
        AND (bpchareq(rsl.sl_color, sh.slcolor)
             AND float8ge(rsl.sl_len_cm,
                          float8mul(sh.slminlen, un.un_fact))
             AND float8le(rsl.sl_len_cm,
                          float8mul(sh.slmaxlen, un.un_fact))
            )
       )
   AND bpchareq(sh.slunit, un.un_name);
```

And finally we apply the already well known rule for `shoelace` (this time on a parsetree that is a little more complex) and get

```
SELECT sh.shoename, sh.sh_avail,
       s.sl_name, s.sl_avail,
       min(sh.sh_avail, s.sl_avail) AS total_avail
  FROM shoe_ready shoe_ready, shoe_ready *OLD*,
       shoe_ready *NEW*, shoe rsh,
       shoelace rsl, shoe *OLD*,
       shoe *NEW*, shoe_data sh,
       unit un, shoelace *OLD*,
       shoelace *NEW*, shoelace_data s,
       unit u
 WHERE (    (int4ge(min(sh.sh_avail, s.sl_avail), 2)
             AND (bpchareq(s.sl_color, sh.slcolor)
                  AND float8ge(float8mul(s.sl_len, u.un_fact),
                               float8mul(sh.slminlen, un.un_fact))
                  AND float8le(float8mul(s.sl_len, u.un_fact),
                               float8mul(sh.slmaxlen, un.un_fact))
                 )
            )
        AND bpchareq(sh.slunit, un.un_name)
       )
   AND bpchareq(s.sl_unit, u.un_name);
```

Again we reduce it to a real SQL statement that is equivalent to the final output of the rule system:

```
SELECT sh.shoename, sh.sh_avail,
       s.sl_name, s.sl_avail,
       min(sh.sh_avail, s.sl_avail) AS total_avail
  FROM shoe_data sh, shoelace_data s, unit u, unit un
 WHERE min(sh.sh_avail, s.sl_avail) >= 2
   AND s.sl_color = sh.slcolor
```

```
AND s.sl_len * u.un_fact >= sh.slminlen * un.un_fact
AND s.sl_len * u.un_fact <= sh.slmaxlen * un.un_fact
AND sh.sl_unit = un.un_name
AND s.sl_unit = u.un_name;
```

Recursive processing of rules rewrote one SELECT from a view into a parsetree, that is equivalent to exactly that what Al had to type if there would be no views at all.

Note

There is currently no recursion stopping mechanism for view rules in the rule system (only for the other rules). This doesn't hurt much, because the only way to push this into an endless loop (blowing up the backend until it reaches the memory limit) is to create tables and then setup the view rules by hand with CREATE RULE in such a way, that one selects from the other that selects from the one. This could never happen if CREATE VIEW is used because on the first CREATE VIEW, the second relation does not exist and thus the first view cannot select from the second.

43.2.3. View Rules in Non-SELECT Statements

Two details of the parsetree aren't touched in the description of view rules above. These are the commandtype and the resultrelation. In fact, view rules don't need these informations.

There are only a few differences between a parsetree for a SELECT and one for any other command. Obviously they have another commandtype and this time the resultrelation points to the rangetable entry where the result should go. Anything else is absolutely the same. So having two tables t1 and t2 with attributes a and b, the parsetrees for the two statements

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

are nearly identical.

- The rangetables contain entries for the tables t1 and t2.
- The targetlists contain one variable that points to attribute b of the rangetable entry for table t2.
- The qualification expressions compare the attributes a of both ranges for equality.

The consequence is, that both parsetrees result in similar execution plans. They are both joins over the two tables. For the UPDATE the missing columns from t1 are added to the targetlist by the optimizer and the final parsetree will read as

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

and thus the executor run over the join will produce exactly the same result set as a

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

will do. But there is a little problem in UPDATE. The executor does not care what the results from the join it is doing are meant for. It just produces a result set of rows. The difference that one is a SELECT command and the other is an UPDATE is handled in the caller of the executor. The caller still knows (looking at the parsetree) that this is an UPDATE, and he knows that this result should go into table t1. But which of the 666 rows that are there has to be replaced by the new row? The plan executed is a join with a qualification that potentially could produce any number of rows between 0 and 666 in unknown order.

To resolve this problem, another entry is added to the targetlist in UPDATE and DELETE statements. The current tuple ID (ctid). This is a system attribute with a special feature. It contains the block and position in the block for the row. Knowing the table, the ctid can be used to find one specific row in a 1.5GB sized table containing millions of rows by fetching one single data block. After adding the ctid to the targetlist, the final result set could be defined as

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Now another detail of Postgres enters the stage. At this moment, table rows aren't overwritten and this is why ABORT TRANSACTION is fast. In an UPDATE, the new result row is inserted into the table (after stripping ctid) and in the tuple header of the row that ctid pointed to the cmax and xmax entries are set to the current command counter and current transaction ID. Thus the old row is hidden and after the transaction commited the vacuum cleaner can really move it out.

Knowing that all, we can simply apply view rules in absolutely the same way to any command. There is no difference.

43.2.4. The Power of Views in Postgres

The above demonstrates how the rule system incorporates view definitions into the original parsetree. In the second example a simple SELECT from one view created a final parsetree that is a join of 4 tables (unit is used twice with different names).

43.2.4.1. Benefits

The benefit of implementing views with the rule system is, that the optimizer has all the information about which tables have to be scanned plus the relationships between these tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single parsetree. And this is still the situation when the original query is already a join over views. Now the optimizer has to decide which is the best path to execute the query. The more information the optimizer has, the better this decision can be. And the rule system as implemented in Postgres ensures, that this is all information available about the query up to now.

43.2.4.2. Concerns

There was a long time where the Postgres rule system was considered broken. The use of rules was not recommended and the only part working where view rules. And also these view rules made problems because the rule system wasn't able to apply them properly on other statements than a SELECT (for example an UPDATE that used data from a view didn't work).

During that time, development moved on and many features where added to the parser and optimizer. The rule system got more and more out of sync with their capabilities and it became harder and harder to start fixing it. Thus, noone did.

For 6.4, someone locked the door, took a deep breath and shuffled that damned thing up. What came out was a rule system with the capabilities described in this document. But there are still some constructs not handled and some where it fails due to things that are currently not supported by the Postgres query optimizer.

- Views with aggregate columns have bad problems. Aggregate expressions in qualifications must be used in subselects. Currently it is not possible to do a join of two views, each having an aggregate column, and compare the two aggregate values in the qualification. In the meantime it is possible to put these aggregate expressions into functions with the appropriate arguments and use them in the view definition.
- Views of unions are currently not supported. Well it's easy to rewrite a simple SELECT into a union. But it is a little difficult if the view is part of a join doing an update.

- ORDER BY clauses in view definitions aren't supported.
- DISTINCT isn't supported in view definitions.

There is no good reason why the optimizer should not handle parsetree constructs that the parser could never produce due to limitations in the SQL syntax. The author hopes that these items disappear in the future.

43.2.5. Implementation Side Effects

Using the described rule system to implement views has a funny side effect. The following does not seem to work:

```
al_bundy=> INSERT INTO shoe (shoename, sh_avail, slcolor)
al_bundy->     VALUES ('sh5', 0, 'black');
INSERT 20128 1
al_bundy=> SELECT shoename, sh_avail, slcolor FROM shoe_data;
shoename  |sh_avail|slcolor
----------+--------+----------
sh1       |       2|black
sh3       |       4|brown
sh2       |       0|black
sh4       |       3|brown
(4 rows)
```

The interesting thing is that the return code for INSERT gave us an object ID and told that 1 row has been inserted. But it doesn't appear in `shoe_data`. Looking into the database directory we can see, that the database file for the view relation `shoe` seems now to have a data block. And that is definitely the case.

We can also issue a DELETE and if it does not have a qualification, it tells us that rows have been deleted and the next vacuum run will reset the file to zero size.

The reason for that behaviour is, that the parsetree for the INSERT does not reference the `shoe` relation in any variable. The targetlist contains only constant values. So there is no rule to apply and it goes down unchanged into execution and the row is inserted. And so for the DELETE.

To change this we can define rules that modify the behaviour of non-SELECT queries. This is the topic of the next section.

43.3. Rules on INSERT, UPDATE and DELETE

43.3.1. Differences to View Rules

Rules that are defined ON INSERT, UPDATE and DELETE are totally different from the view rules described in the previous section. First, their CREATE RULE command allows more:

- They can have no action.
- They can have multiple actions.
- The keyword INSTEAD is optional.
- The pseudo relations NEW and OLD become useful.
- They can have rule qualifications.

Second, they don't modify the parsetree in place. Instead they create zero or many new parsetrees and can throw away the original one.

43.3.2. How These Rules Work

Keep the syntax

```
CREATE RULE rule_name AS ON event
    TO object [WHERE rule_qualification]
    DO [INSTEAD] [action | (actions) | NOTHING];
```

in mind. In the following, "update rules" means rules that are defined ON INSERT, UPDATE or DELETE.

Update rules get applied by the rule system when the result relation and the commandtype of a parsetree are equal to the object and event given in the CREATE RULE command. For update rules, the rule system creates a list of parsetrees. Initially the parsetree list is empty. There can be zero (NOTHING keyword), one or multiple actions. To simplify, we look at a rule with one action. This rule can have a qualification or not and it can be INSTEAD or not.

What is a rule qualification? It is a restriction that tells when the actions of the rule should be done and when not. This qualification can only reference the NEW and/or OLD pseudo relations which are basically the relation given as object (but with a special meaning).

So we have four cases that produce the following parsetrees for a one-action rule.

- No qualification and not INSTEAD:
  - The parsetree from the rule action where the original parsetrees qualification has been added.
- No qualification but INSTEAD:
  - The parsetree from the rule action where the original parsetrees qualification has been added.
- Qualification given and not INSTEAD:
  - The parsetree from the rule action where the rule qualification and the original parsetrees qualification have been added.
- Qualification given and INSTEAD:
  - The parsetree from the rule action where the rule qualification and the original parsetrees qualification have been added.
  - The original parsetree where the negated rule qualification has been added.

Finally, if the rule is not INSTEAD, the unchanged original parsetree is added to the list. Since only qualified INSTEAD rules already add the original parsetree, we end up with a total maximum of two parsetrees for a rule with one action.

The parsetrees generated from rule actions are thrown into the rewrite system again and maybe more rules get applied resulting in more or less parsetrees. So the parsetrees in the rule actions must have either another commandtype or another resultrelation. Otherwise this recursive process will end up in a loop. There is a compiled in recursion limit of currently 10 iterations. If after 10 iterations there are still update rules to apply the rule system assumes a loop over multiple rule definitions and aborts the transaction.

The parsetrees found in the actions of the `pg_rewrite` system catalog are only templates. Since they can reference the rangetable entries for NEW and OLD, some substitutions have to be made before they

can be used. For any reference to NEW, the targetlist of the original query is searched for a corresponding entry. If found, that entries expression is placed into the reference. Otherwise NEW means the same as OLD. Any reference to OLD is replaced by a reference to the rangetable entry which is the resultrelation.

43.3.2.1. A First Rule Step by Step

We want to trace changes to the sl_avail column in the `shoelace_data` relation. So we setup a log table and a rule that writes us entries every time and UPDATE is performed on `shoelace_data`.

```
CREATE TABLE shoelace_log (
    sl_name    char(10),      -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    name,          -- who did it
    log_when   datetime       -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail != OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                       NEW.sl_name,
                                       NEW.sl_avail,
                                       getpgusername(),
                                       'now'::text
                                     );
```

One interesting detail is the casting of 'now' in the rules INSERT action to type text. Without that, the parser would see at CREATE RULE time, that the target type in `shoelace_log` is a datetime and tries to make a constant from it - with success. So a constant datetime value would be stored in the rule action and all log entries would have the time of the CREATE RULE statement. Not exactly what we want. The casting causes that the parser constructs a datetime('now'::text) from it and this will be evaluated when the rule is executed.

Now Al does

```
al_bundy=> UPDATE shoelace_data SET sl_avail = 6

al_bundy->     WHERE sl_name = 'sl7';
```

and we look at the logtable.

```
al_bundy=> SELECT * FROM shoelace_log;
sl_name   |sl_avail|log_who|log_when
----------+--------+-------+--------------------------------
sl7       |       6|Al     |Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

That's what we expected. What happened in the background is the following. The parser created the parsetree (this time the parts of the original parsetree are highlighted because the base of operations is the rule action for update rules).

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE bpchareq(shoelace_data.sl_name, 'sl7');
```

There is a rule 'log_shoelace' that is ON UPDATE with the rule qualification expression

```
int4ne(NEW.sl_avail, OLD.sl_avail)
```

and one action

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avail,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_log shoelace_log;
```

Don't trust the output of the pg_rules system view. It specially handles the situation that there are only references to NEW and OLD in the INSERT and outputs the VALUES format of INSERT. In fact there is no difference between an INSERT ... VALUES and an INSERT ... SELECT on parsetree level. They both have rangetables, targetlists and maybe qualifications etc. The optimizer later decides, if to create an execution plan of type result, seqscan, indexscan, join or whatever for that parsetree. If there are no references to rangetable entries leftin the parsetree , it becomes a result execution plan (the INSERT ... VALUES version). The rule action above can truely result in both variants.

The rule is a qualified non-INSTEAD rule, so the rule system has to return two parsetrees. The modified rule action and the original parsetree. In the first step the rangetable of the original query is incorporated into the rules action parsetree. This results in

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avai,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log;
```

In step 2 the rule qualification is added to it, so the result set is restricted to rows where sl_avail changes.

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avai,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail);
```

In step 3 the original parsetrees qualification is added, restricting the resultset further to only the rows touched by the original parsetree.

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avai,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 4 substitutes NEW references by the targetlist entries from the original parsetree or with the matching variable references from the result relation.

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
```

```
 WHERE int4ne(6, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 5 replaces OLD references into resultrelation references.

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(6, shoelace_data.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

That's it. So reduced to the max the return from the rule system is a list of two parsetrees that are the same as the statements:

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), 'now'
  FROM shoelace_data
 WHERE 6 != shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

These are executed in this order and that is exactly what the rule defines. The subtitutions and the qualifications added ensure, that if the original query would be an

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

No log entry would get written because due to the fact that this time the original parsetree does not contain a targetlist entry for sl_avail, NEW.sl_avail will get replaced by shoelace_data.sl_avail resulting in the extra query

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, shoelace_data.sl_avail,
       getpgusername(), 'now'
  FROM shoelace_data
 WHERE shoelace_data.sl_avail != shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

and that qualification will never be true. Since the is no difference on parsetree level between an INSERT ... SELECT, and an INSERT ... VALUES, it will also work if the original query modifies multiple rows. So if Al would issue the command

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

four rows in fact get updated (sl1, sl2, sl3 and sl4). But sl3 already has sl_avail = 0. This time, the original parsetrees qualification is different and that results in the extra parsetree

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 0,
       getpgusername(), 'now'
```

```
  FROM shoelace_data
 WHERE 0 != shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

This parsetree will surely insert three new log entries. And that's absolutely correct.

It is important, that the original parsetree is executed last. The Postgres "traffic cop" does a command counter increment between the execution of the two parsetrees so the second one can see changes made by the first. If the UPDATE would have been executed first, all the rows are already set to zero, so the logging INSERT would not find any row where 0 != shoelace_data.sl_avail.

43.3.3. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can INSERT, UPDATE and DELETE invisible data on them is to let those parsetrees get thrown away. We create the rules

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

If Al now tries to do any of these operations on the view relation `shoe`, the rule system will apply the rules. Since the rules have no actions and are INSTEAD, the resulting list of parsetrees will be empty and the whole query will become nothing because there is nothing left to be optimized or executed after the rule system is done with it.

Note

This fact might irritate frontend applications because absolutely nothing happened on the database and thus, the backend will not return anything for the query. Not even a PGRES_EMPTY_QUERY or so will be available in libpq. In psql, nothing happens. This might change in the future.

A more sophisticated way to use the rule system is to create rules that rewrite the parsetree into one that does the right operation on the real tables. To do that on the `shoelace` view, we create the following rules:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit);

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data SET
           sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
```

```
            sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

Now there is a pack of shoelaces arriving in Al's shop and it has a big partlist. Al is not that good in calculating and so we don't want him to manually update the shoelace view. Instead we setup two little tables, one where he can insert the items from the partlist and one with a special trick. The create commands for anything are:

```
CREATE TABLE shoelace_arrive (
    arr_name    char(10),
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     char(10),
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace SET
           sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Now Al can sit down and do whatever until

```
al_bundy=> SELECT * FROM shoelace_arrive;
arr_name  |arr_quant
----------+---------
sl3       |       10
sl6       |       20
sl8       |       20
(3 rows)
```

is exactly that what's on the part list. We take a quick look at the current data,

```
al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

move the arrived shoelaces in

```
al_bundy=> INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

and check the results

```
al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl4       |       8|black     |    40|inch    |    101.6
sl3       |      10|black     |    35|inch    |     88.9
sl8       |      21|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |      20|brown     |   0.9|m       |       90
(8 rows)

al_bundy=> SELECT * FROM shoelace_log;
sl_name   |sl_avail|log_who|log_when
----------+--------+------+---------------------------------
sl7       |       6|Al     |Tue Oct 20 19:14:45 1998 MET DST
sl3       |      10|Al     |Tue Oct 20 19:25:16 1998 MET DST
sl6       |      20|Al     |Tue Oct 20 19:25:16 1998 MET DST
sl8       |      21|Al     |Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

It's a long way from the one INSERT ... SELECT to these results. And it's description will be the last in this document (but not the last example :-). First there was the parsers output

```
INSERT INTO shoelace_ok SELECT
       shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Now the first rule 'shoelace_ok_ins' is applied and turns it into

```
UPDATE shoelace SET
       sl_avail = int4pl(shoelace.sl_avail,
shoelace_arrive.arr_quant)
     FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
          shoelace_ok *OLD*, shoelace_ok *NEW*,
          shoelace shoelace
    WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name);
```

and throws away the original INSERT on `shoelace_ok`. This rewritten query is passed to the rule system again and the second applied rule 'shoelace_upd' produced

```
UPDATE shoelace_data SET
       sl_name = shoelace.sl_name,
       sl_avail = int4pl(shoelace.sl_avail,
shoelace_arrive.arr_quant),
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok *OLD*, shoelace_ok *NEW*,
```

```
          shoelace shoelace, shoelace *OLD*,
          shoelace *NEW*, shoelace_data showlace_data
     WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name)
       AND bpchareq(shoelace_data.sl_name, shoelace.sl_name);
```

Again it's an INSTEAD rule and the previous parsetree is trashed. Note that this query sill uses the view `shoelace` But the rule system isn't finished with this loop so it continues and applies the rule '_RETshoelace' on it and we get

```
    UPDATE shoelace_data SET
           sl_name = s.sl_name,
           sl_avail = int4pl(s.sl_avail, shoelace_arrive.arr_quant),
           sl_color = s.sl_color,
           sl_len = s.sl_len,
           sl_unit = s.sl_unit
      FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
           shoelace_ok *OLD*, shoelace_ok *NEW*,
           shoelace shoelace, shoelace *OLD*,
           shoelace *NEW*, shoelace_data showlace_data,
           shoelace *OLD*, shoelace *NEW*,
           shoelace_data s, unit u
     WHERE bpchareq(s.sl_name, showlace_arrive.arr_name)
       AND bpchareq(shoelace_data.sl_name, s.sl_name);
```

Again an update rule has been applied and so the wheel turns on and we are in rewrite round 3. This time rule 'log_shoelace' gets applied what produces the extra parsetree

```
    INSERT INTO shoelace_log SELECT
           s.sl_name,
           int4pl(s.sl_avail, shoelace_arrive.arr_quant),
           getpgusername(),
           datetime('now'::text)
      FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
           shoelace_ok *OLD*, shoelace_ok *NEW*,
           shoelace shoelace, shoelace *OLD*,
           shoelace *NEW*, shoelace_data showlace_data,
           shoelace *OLD*, shoelace *NEW*,
           shoelace_data s, unit u,
           shoelace_data *OLD*, shoelace_data *NEW*
           shoelace_log shoelace_log
     WHERE bpchareq(s.sl_name,  showlace_arrive.arr_name)
       AND bpchareq(shoelace_data.sl_name, s.sl_name);
       AND int4ne(int4pl(s.sl_avail, shoelace_arrive.arr_quant),
                                                 s.sl_avail);
```

After that the rule system runs out of rules and returns the generated parsetrees. So we end up with two final parsetrees that are equal to the SQL statements

```
    INSERT INTO shoelace_log SELECT
           s.sl_name,
           s.sl_avail + shoelace_arrive.arr_quant,
           getpgusername(),
           'now'
      FROM shoelace_arrive shoelace_arrive, shoelace_data
shoelace_data,
```

```
            shoelace_data s
     WHERE s.sl_name = shoelace_arrive.arr_name
       AND shoelace_data.sl_name = s.sl_name
       AND s.sl_avail + shoelace_arrive.arr_quant != s.sl_avail;

    UPDATE shoelace_data SET
           sl_avail = shoelace_data.sl_avail +
 shoelace_arrive.arr_quant
     FROM shoelace_arrive shoelace_arrive, shoelace_data
 shoelace_data,
          shoelace_data s
    WHERE s.sl_name = shoelace_arrive.sl_name
      AND shoelace_data.sl_name = s.sl_name;
```

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that's a bit ugly. Looking at the two queries turns out, that the `shoelace_data` relation appears twice in the rangetable where it could definitely be reduced to one. The optimizer does not handle it and so the execution plan for the rule systems output of the INSERT will be

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

while omitting the extra rangetable entry would result in a

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

that totally produces the same entries in the log relation. Thus, the rule system caused one extra scan on the `shoelace_data` relation that is absolutely not necessary. And the same obsolete scan is done once more in the UPDATE. But it was a really hard job to make that all possible at all.

A final demonstration of the Postgres rule system and it's power. There is a cute blonde that sells shoelaces. And what Al could never realize, she's not only cute, she's smart too - a little too smart. Thus, it happens from time to time that Al orders shoelaces that are absolutely not sellable. This time he ordered 1000 pairs of magenta shoelaces and since another kind is currently not available but he committed to buy some, he also prepared his database for pink ones.

```
    al_bundy=> INSERT INTO shoelace VALUES
    al_bundy->     ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
    al_bundy=> INSERT INTO shoelace VALUES
    al_bundy->     ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Since this happens often, we must lookup for shoelace entries, that fit for absolutely no shoe sometimes. We could do that in a complicated statement every time, or we can setup a view for it. The view for this is

```
CREATE VIEW shoelace_obsolete AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

It's output is

```
al_bundy=> SELECT * FROM shoelace_obsolete;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl9       |       0|pink      |    35|inch    |     88.9
sl10      |    1000|magenta   |    40|inch    |    101.6
```

For the 1000 magenta shoelaces we must debt Al before we can throw 'em away, but that's another problem. The pink entry we delete. To make it a little harder for Postgres, we don't delete it directly. Instead we create one more view

```
CREATE VIEW shoelace_candelete AS
    SELECT * FROM shoelace_obsolete WHERE sl_avail = 0;
```

and do it this way:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_candelete
             WHERE sl_name = shoelace.sl_name);
```

Voila:

```
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl4       |       8|black     |    40|inch    |    101.6
sl3       |      10|black     |    35|inch    |     88.9
sl8       |      21|brown     |    40|inch    |    101.6
sl10      |    1000|magenta   |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |      20|brown     |   0.9|m       |       90
(9 rows)
```

A DELETE on a view, with a subselect qualification that in total uses 4 nesting/joined views, where one of them itself has a subselect qualification containing a view and where calculated view columns are used, gets rewritten into one single parsetree that deletes the requested data from a real table.

I think there are only a few situations out in the real world, where such a construct is necessary. But it makes me feel comfortable that it works.

The truth is

Doing this I found one more bug while writing this document. But after fixing that I was a little amazed that it works at all.

43.4. Rules and Permissions

Due to rewriting of queries by the Postgres rule system, other tables/views than those used in the original query get accessed. Using update rules, this can include write access to tables.

Rewrite rules don't have a separate owner. The owner of a relation (table or view) is automatically the owner of the rewrite rules that are defined for it. The Postgres rule system changes the behaviour of the default access control system. Relations that are used due to rules get checked during the rewrite against the permissions of the relation owner, the rule is defined on. This means, that a user does only need the required permissions for the tables/views he names in his queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the secretary of the office. He can construct the following:

```
CREATE TABLE phone_data (person text, phone text, private bool);
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE NOT private;
GRANT SELECT ON phone_number TO secretary;
```

Nobody except him (and the database superusers) can access the phone_data table. But due to the GRANT, the secretary can SELECT from the phone_number view. The rule system will rewrite the SELECT from phone_number into a SELECT from phone_data and add the qualification that only entries where private is false are wanted. Since the user is the owner of phone_number, the read access to phone_data is now checked against his permissions and the query is considered granted. The check for accessing phone_number is still performed, so nobody than the secretary can use it.

The permissions are checked rule by rule. So the secretary is for now the only one who can see the public phone numbers. But the secretary can setup another view and grant access to that to public. Then, anyone can see the phone_number data through the secretaries view. What the secretary cannot do is to create a view that directly accesses phone_data (actually he can, but it will not work since every access aborts the transaction during the permission checks). And as soon as the user will notice, that the secretary opened his phone_number view, he can REVOKE his access. Immediately any access to the secretaries view will fail.

Someone might think that this rule by rule checking is a security hole, but in fact it isn't. If this would not work, the secretary could setup a table with the same columns as phone_number and copy the data to there once per day. Then it's his own data and he can grant access to everyone he wants. A GRANT means "I trust you". If someone you trust does the thing above, it's time to think it over and then REVOKE.

This mechanism does also work for update rules. In the examples of the previous section, the owner of the tables in Al's database could GRANT SELECT, INSERT, UPDATE and DELETE on the shoelace view to al. But only SELECT on shoelace_log. The rule action to write log entries will still be executed successfull. And Al could see the log entries. But he cannot create fake entries, nor could he manipulate or remove existing ones.

Warning

GRANT ALL currently includes RULE permission. This means the granted user could drop the rule, do the changes and reinstall it. I think this should get changed quickly.

43.5. Rules versus Triggers

Many things that can be done using triggers can also be implemented using the Postgres rule system. What currently cannot be implemented by rules are some kinds of constraints. It is possible, to place a qualified

rule that rewrites a query to NOTHING if the value of a column does not appear in another table. But then the data is silently thrown away and that's not a good idea. If checks for valid values are required, and in the case of an invalid value an error message should be generated, it must be done by a trigger for now.

On the other hand a trigger that is fired on INSERT on a view can do the same as a rule, put the data somewhere else and suppress the insert in the view. But it cannot do the same thing on UPDATE or DELETE, because there is no real data in the view relation that could be scanned and thus the trigger would never get called. Only a rule will help.

For the things that can be implemented by both, it depends on the usage of the database, which is the best. A trigger is fired for any row affected once. A rule manipulates the parsetree or generates an additional one. So if many rows are affected in one statement, a rule issuing one extra query would usually do a better job than a trigger that is called for any single row and must execute his operations this many times.

For example: There are two tables

```
    CREATE TABLE computer (
        hostname        text     -- indexed
 manufacturer    text     -- indexed
    );

    CREATE TABLE software (
        software        text,    -- indexed
        hostname        text     -- indexed
    );
```

Both tables have many thousands of rows and the index on hostname is unique. The hostname column contains the full qualified domain name of the computer. The rule/trigger should constraint delete rows from software that reference the deleted host. Since the trigger is called for each individual row deleted from computer, it can use the statement

```
    DELETE FROM software WHERE hostname = $1;
```

in a prepared and saved plan and pass the hostname in the parameter. The rule would be written as

```
    CREATE RULE computer_del AS ON DELETE TO computer
        DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Now we look at different types of deletes. In the case of a

```
    DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

the table computer is scanned by index (fast) and the query issued by the trigger would also be an index scan (fast too). The extra query from the rule would be a

```
    DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                           AND software.hostname = computer.hostname;
```

Since there are appropriate indices setup, the optimizer will create a plan of

```
    Nestloop
      ->  Index Scan using comp_hostidx on computer
      ->  Index Scan using soft_hostidx on software
```

So there would be not that much difference in speed between the trigger and the rule implementation. With the next delete we want to get rid of all the 2000 computers where the hostname starts with 'old'. There are two possible queries to do that. One is

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

Where the plan for the rule query will be a

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
 ->  Index Scan using comp_hostidx on computer
```

The other possible query is a

```
DELETE FROM computer WHERE hostname ~ '^old';
```

with the execution plan

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

This shows, that the optimizer does not realize that the qualification for the hostname on computer could also be used for an index scan on software when there are multiple qualification expressions combined with AND, what he does in the regexp version of the query. The trigger will get invoked once for any of the 2000 old computers that have to be deleted and that will result in one index scan over computer and 2000 index scans for the software. The rule implementation will do it with two queries over indices. And it depends on the overall size of the software table if the rule will still be faster in the seqscan situation. 2000 query executions over the SPI manager take some time, even if all the index blocks to look them up will soon appear in the cache.

The last query we look at is a

```
DELETE FROM computer WHERE manufacurer = 'bim';
```

Again this could result in many rows to be deleted from computer. So the trigger will again fire many queries into the executor. But the rule plan will again be the Nestloop over two IndexScan's. Only using another index on computer:

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

resulting from the rules query

```
DELETE FROM software WHERE computer.manufacurer = 'bim'
                       AND software.hostname = computer.hostname;
```

In any of these cases, the extra queries from the rule system will be more or less independent from the number of affected rows in a query.

Another situation are cases on UPDATE where it depends on the change of an attribute if an action should be performed or not. In Postgres version 6.4, the attribute specification for rule events is disabled (it will have it's comeback latest in 6.5, maybe earlier - stay tuned). So for now the only way to create a rule as in the shoelace_log example is to do it with a rule qualification. That results in an extra query that is performed allways, even if the attribute of interest cannot change at all because it does not appear in the targetlist of the initial query. When this is enabled again, it will be one more advantage of rules over triggers. Optimization of a trigger must fail by definition in this case, because the fact that it's actions will

only be done when a specific attribute is updated is hidden in it's functionality. The definition of a trigger only allows to specify it on row level, so whenever a row is touched, the trigger must be called to make it's decision. The rule system will know it by looking up the targetlist and will suppress the additional query completely if the attribute isn't touched. So the rule, qualified or not, will only do it's scan's if there ever could be something to do.

Rules will only be significant slower than triggers if their actions result in large and bad qualified joins, a situation where the optimizer fails. They are a big hammer. Using a big hammer without caution can cause big damage. But used with the right touch, they can hit any nail on the head.

Chapter 44. Interfacing Extensions To Indices

The procedures described thus far let you define a new type, new functions and new operators. However, we cannot yet define a secondary index (such as a B-tree, R-tree or hash access method) over a new type or its operators.

Look back at

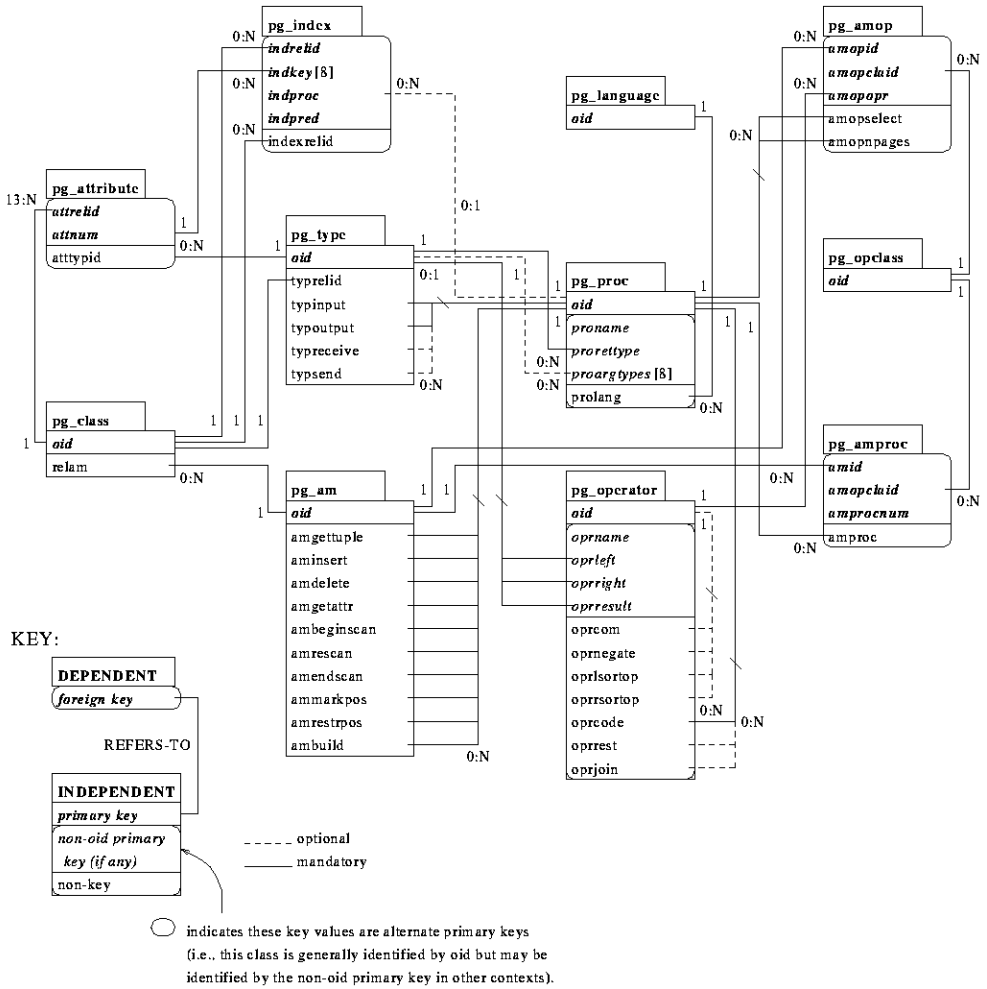


Figure 3. The major POSTGRES system catalogs.

. The right half shows the catalogs that we must modify in order to tell Postgres how to use a user-defined type and/or user-defined operators with an index (i.e., `pg_am`, `pg_amop`, `pg_amproc` and `pg_opclass`). Unfortunately, there is no simple command to do this. We will demonstrate how to modify these catalogs through a running example: a new operator class for the B-tree access method that sorts integers in ascending absolute value order.

The `pg_am` class contains one instance for every user defined access method. Support for the heap access method is built into Postgres, but every other access method is described here. The schema is

Table 44.1. Index Schema

Attribute	Description
amname	name of the access method
amowner	object id of the owner's instance in pg_user
amkind	not used at present, but set to 'o' as a place holder
amstrategies	number of strategies for this access method (see below)
amsupport	number of support routines for this access method (see below)
amgettuple aminsert ...	procedure identifiers for interface routines to the access method. For example, regproc ids for opening, closing, and getting instances from the access method appear here.

The object ID of the instance in `pg_am` is used as a foreign key in lots of other classes. You don't need to add a new instance to this class; all you're interested in is the object ID of the access method instance you want to extend:

```
SELECT oid FROM pg_am WHERE amname = 'btree';

          +----+
          |oid |
          +----+
          |403 |
          +----+
```

The `amstrategies` attribute exists to standardize comparisons across data types. For example, B-trees impose a strict ordering on keys, lesser to greater. Since Postgres allows the user to define operators, Postgres cannot look at the name of an operator (eg, ">" or "<") and tell what kind of comparison it is. In fact, some access methods don't impose any ordering at all. For example, R-trees express a rectangle-containment relationship, whereas a hashed data structure expresses only bitwise similarity based on the value of a hash function. Postgres needs some consistent way of taking a qualification in your query, looking at the operator and then deciding if a usable index exists. This implies that Postgres needs to know, for example, that the "<=" and ">" operators partition a B-tree. Postgres uses strategies to express these relationships between operators and the way they can be used to scan indices.

Defining a new set of strategies is beyond the scope of this discussion, but we'll explain how B-tree strategies work because you'll need to know that to add a new operator class. In the `pg_am` class, the amstrategies attribute is the number of strategies defined for this access method. For B-trees, this number is 5. These strategies correspond to

Table 44.2. B-tree Strategies

Operation	Index
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

The idea is that you'll need to add procedures corresponding to the comparisons above to the pg_amop relation (see below). The access method code can use these strategy numbers, regardless of data type, to figure out how to partition the B-tree, compute selectivity, and so on. Don't worry about the details of adding procedures yet; just understand that there must be a set of these procedures for int2, int4, oid, and every other data type on which a B-tree can operate. Sometimes, strategies aren't enough information for the system to figure out how to use an index. Some access methods require other support routines in order to work. For example, the B-tree access method must be able to compare two keys and determine whether one is greater than, equal to, or less than the other. Similarly, the R-tree access method must be able to compute intersections, unions, and sizes of rectangles. These operations do not correspond to user qualifications in SQL queries; they are administrative routines used by the access methods, internally.

In order to manage diverse support routines consistently across all Postgres access methods, pg_am includes an attribute called amsupport. This attribute records the number of support routines used by an access method. For B-trees, this number is one -- the routine to take two keys and return -1, 0, or +1, depending on whether the first key is less than, equal to, or greater than the second.

Note

Strictly speaking, this routine can return a negative number (< 0), 0, or a non-zero positive number (> 0).

The amstrategies entry in pg_am is just the number of strategies defined for the access method in question. The procedures for less than, less equal, and so on don't appear in pg_am. Similarly, amsupport is just the number of support routines required by the access method. The actual routines are listed elsewhere.

The next class of interest is pg_opclass. This class exists only to associate a name with an oid. In pg_amop, every B-tree operator class has a set of procedures, one through five, above. Some existing opclasses are int2_ops, int4_ops, and oid_ops. You need to add an instance with your opclass name (for example, complex_abs_ops) to pg_opclass. The oid of this instance is a foreign key in other classes.

```
INSERT INTO pg_opclass (opcname) VALUES ('complex_abs_ops');

SELECT oid, opcname
    FROM pg_opclass
    WHERE opcname = 'complex_abs_ops';

         +------+--------------+
         |oid   | opcname      |
         +------+--------------+
         |17314 | int4_abs_ops |
         +------+--------------+
```

Note that the oid for your pg_opclass instance will be different! You should substitute your value for 17314 wherever it appears in this discussion.

So now we have an access method and an operator class. We still need a set of operators; the procedure for defining operators was discussed earlier in this manual. For the complex_abs_ops operator class on Btrees, the operators we require are:

```
absolute value less-than
absolute value less-than-or-equal
absolute value equal
absolute value greater-than-or-equal
```

```
        absolute value greater-than
```

Suppose the code that implements the functions defined is stored in the file `PGROOT/src/tutorial/complex.c`

Part of the code look like this: (note that we will only show the equality operator for the rest of the examples. The other four operators are very similar. Refer to `complex.c` or `complex.sql` for the details.)

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

         bool
         complex_abs_eq(Complex *a, Complex *b)
         {
             double amag = Mag(a), bmag = Mag(b);
             return (amag==bmag);
         }
```

There are a couple of important things that are happening below.

First, note that operators for less-than, less-than-or equal, equal, greater-than-or-equal, and greater-than for `int4` are being defined. All of these operators are already defined for `int4` under the names <, <=, =, >=, and >. The new operators behave differently, of course. In order to guarantee that Postgres uses these new operators rather than the old ones, they need to be named differently from the old ones. This is a key point: you can overload operators in Postgres, but only if the operator isn't already defined for the argument types. That is, if you have < defined for (int4, int4), you can't define it again. Postgres does not check this when you define your operator, so be careful. To avoid this problem, odd names will be used for the operators. If you get this wrong, the access methods are likely to crash when you try to do scans.

The other important point is that all the operator functions return Boolean values. The access methods rely on this fact. (On the other hand, the support function returns whatever the particular access method expects -- in this case, a signed integer.) The final routine in the file is the "support routine" mentioned when we discussed the amsupport attribute of the `pg_am` class. We will use this later on. For now, ignore it.

```
CREATE FUNCTION complex_abs_eq(complex, complex)
              RETURNS bool
              AS 'PGROOT/tutorial/obj/complex.so'
              LANGUAGE 'c';
```

Now define the operators that use them. As noted, the operator names must be unique among all operators that take two `int4` operands. In order to see if the operator names listed below are taken, we can do a query on `pg_operator`:

```
    /*
     * this query uses the regular expression operator (~)
     * to find three-character operator names that end in
     * the character &
     */
    SELECT *
     FROM pg_operator
     WHERE oprname ~ '^..&$'::text;
```

to see if your name is taken for the types you want. The important things here are the procedure (which are the C functions defined above) and the restriction and join selectivity functions. You should just use the ones used below--note that there are different such functions for the less-than, equal, and greater-than cases. These must be supplied, or the access method will crash when it tries to use the operator. You should copy the names for restrict and join, but use the procedure names you defined in the last step.

```
CREATE OPERATOR = (
     leftarg = complex, rightarg = complex,
     procedure = complex_abs_eq,
     restrict = eqsel, join = eqjoinsel
         )
```

Notice that five operators corresponding to less, less equal, equal, greater, and greater equal are defined.

We're just about finished. the last thing we need to do is to update the `pg_amop` relation. To do this, we need the following attributes:

Table 44.3. `pg_amproc` Schema

Attribute	Description
amopid	the `oid` of the `pg_am` instance for B-tree (== 403, see above)
amopclaid	the `oid` of the `pg_opclass` instance for `int4_abs_ops` (== whatever you got instead of `17314`, see above)
amopopr	the `oids` of the operators for the opclass (which we'll get in just a minute)
amopselect, amopnpages	cost functions

The cost functions are used by the query optimizer to decide whether or not to use a given index in a scan. Fortunately, these already exist. The two functions we'll use are `btreesel`, which estimates the selectivity of the B-tree, and `btreenpage`, which estimates the number of pages a search will touch in the tree.

So we need the `oids` of the operators we just defined. We'll look up the names of all the operators that take two `int4`s, and pick ours out:

```
SELECT o.oid AS opoid, o.oprname
 INTO TABLE complex_ops_tmp
 FROM pg_operator o, pg_type t
 WHERE o.oprleft = t.oid and o.oprright = t.oid
  and t.typname = 'complex';
```

```
    +------+---------+
    |oid   | oprname |
    +------+---------+
    |17321 | <       |
    +------+---------+
    |17322 | <=      |
    +------+---------+
    |17323 |  =      |
    +------+---------+
    |17324 | >=      |
    +------+---------+
    |17325 | >       |
    +------+---------+
```

(Again, some of your `oid` numbers will almost certainly be different.) The operators we are interested in are those with `oids` 17321 through 17325. The values you get will probably be different, and you should substitute them for the values below. We can look at the operator names and pick out the ones we just added.

Now we're ready to update pg_amop with our new operator class. The most important thing in this entire discussion is that the operators are ordered, from less equal through greater equal, in pg_amop. We add the instances we need:

```
INSERT INTO pg_amop (amopid, amopclaid,
 amopopr, amopstrategy,
 amopselect, amopnpages)
SELECT am.oid, opcl.oid, c.opoid, 3,
 'btreesel'::regproc, 'btreenpage'::regproc
FROM pg_am am, pg_opclass opcl, complex_ops_tmp c
WHERE amname = 'btree'
 and opcname = 'complex_abs_ops'
 and c.oprname = '=';
```

Note the order: "less than" is 1, "less than or equal" is 2, "equal" is 3, "greater than or equal" is 4, and "greater than" is 5.

The last step (finally!) is registration of the "support routine" previously described in our discussion of pg_am. The oid of this support routine is stored in the pg_amproc class, keyed by the access method oid and the operator class oid. First, we need to register the function in Postgres (recall that we put the C code that implements this routine in the bottom of the file in which we implemented the operator routines):

```
CREATE FUNCTION int4_abs_cmp(int4, int4)
 RETURNS int4
 AS 'PGROOT/tutorial/obj/complex.so'
 LANGUAGE 'c';

SELECT oid, proname FROM pg_proc
 WHERE prname = 'int4_abs_cmp';

     +------+--------------+
     |oid   | proname      |
     +------+--------------+
     |17328 | int4_abs_cmp |
     +------+--------------+
```

(Again, your oid number will probably be different and you should substitute the value you see for the value below.) Recalling that the B-tree instance's oid is 403 and that of int4_abs_ops is 17314, we can add the new instance as follows:

```
INSERT INTO pg_amproc (amid, amopclaid, amproc, amprocnum)
 VALUES ('403'::oid, -- btree oid
 '17314'::oid,       --  pg_opclass tuple
 '17328'::oid,       -- new pg_proc oid
 '1'::int2);
```

Chapter 45. GiST Indices

Gene Selkov

The information about GIST is at http://GiST.CS.Berkeley.EDU:8000/gist/ with more on different indexing and sorting schemes at http://s2k-ftp.CS.Berkeley.EDU:8000/personal/jmh/ And there is more interesting reading at the Berkely database site at http://epoch.cs.berkeley.edu:8000/.

Author

This extraction from an e-mail sent by Eugene Selkov Jr.[1] contains good information on GiST. Hopefully we will learn more in the future and update this information. - thomas 1998-03-01

Well, I can't say I quite understand what's going on, but at least I (almost) succeeded in porting GiST examples to linux. The GiST access method is already in the postgres tree (`src/backend/access/gist`).

Examples at Berkeley[2] come with an overview of the methods and demonstrate spatial index mechanisms for 2D boxes, polygons, integer intervals and text (see also GiST at Berkeley[3]). In the box example, we are supposed to see a performance gain when using the GiST index; it did work for me but I do not have a reasonably large collection of boxes to check that. Other examples also worked, except polygons: I got an error doing

```
test=> create index pix on polytmp
test-> using gist (p:box gist_poly_ops) with (islossy);
ERROR:  cannot open pix

(PostgreSQL 6.3              Sun Feb  1 14:57:30 EST 1998)
```

I could not get sense of this error message; it appears to be something we'd rather ask the developers about (see also Note 4 below). What I would suggest here is that someone of you linux guys (linux==gcc?) fetch the original sources quoted above and apply my patch (see attachment) and tell us what you feel about it. Looks cool to me, but I would not like to hold it up while there are so many competent people around.

A few notes on the sources:

1. I failed to make use of the original (HPUX) Makefile and rearranged the Makefile from the ancient postgres95 tutorial to do the job. I tried to keep it generic, but I am a very poor makefile writer -- just did some monkey work. Sorry about that, but I guess it is now a little more portable that the original makefile.

2. I built the example sources right under pgsql/src (just extracted the tar file there). The aforementioned Makefile assumes it is one level below pgsql/src (in our case, in pgsql/src/pggist).

3. The changes I made to the *.c files were all about #include's, function prototypes and typecasting. Other than that, I just threw away a bunch of unused vars and added a couple parentheses to please gcc. I hope I did not screw up too much :)

4. There is a comment in polyproc.sql:

```
-- -- there's a memory leak in rtree poly_ops!!
-- -- create index pix2 on polytmp using rtree (p poly_ops);
```

[1] mailto:selkovjr@mcs.anl.gov
[2] ftp://s2k-ftp.cs.berkeley.edu/pub/gist/pggist/pggist.tgz
[3] http://gist.cs.berkeley.edu:8000/gist/

Roger that!! I thought it could be related to a number of Postgres versions back and tried the query. My system went nuts and I had to shoot down the postmaster in about ten minutes.

I will continue to look into GiST for a while, but I would also appreciate more examples of R-tree usage.

Chapter 46. Linking Dynamically-Loaded Functions

After you have created and registered a user-defined function, your work is essentially done. Postgres, however, must load the object code (e.g., a `.o` file, or a shared library) that implements your function. As previously mentioned, Postgres loads your code at runtime, as required. In order to allow your code to be dynamically loaded, you may have to compile and link-edit it in a special way. This section briefly describes how to perform the compilation and link-editing required before you can load your user-defined functions into a running Postgres server. Note that this process has changed as of Version 4.2.

Tip

The old Postgres dynamic loading mechanism required in-depth knowledge in terms of executable format, placement and alignment of executable instructions within memory, etc. on the part of the person writing the dynamic loader. Such loaders tended to be slow and buggy. As of Version 4.2, the Postgres dynamic loading mechanism has been rewritten to use the dynamic loading mechanism provided by the operating system. This approach is generally faster, more reliable and more portable than our previous dynamic loading mechanism. The reason for this is that nearly all modern versions of UNIX use a dynamic loading mechanism to implement shared libraries and must therefore provide a fast and reliable mechanism. On the other hand, the object file must be postprocessed a bit before it can be loaded into Postgres. We hope that the large increase in speed and reliability will make up for the slight decrease in convenience.

You should expect to read (and reread, and re-reread) the manual pages for the C compiler, cc(1), and the link editor, ld(1), if you have specific questions. In addition, the regression test suites in the directory `PGROOT/src/regress` contain several working examples of this process. If you copy what these tests do, you should not have any problems. The following terminology will be used below:

- *Dynamic loading* is what Postgres does to an object file. The object file is copied into the running Postgres server and the functions and variables within the file are made available to the functions within the Postgres process. Postgres does this using the dynamic loading mechanism provided by the operating system.
- *Loading and link editing* is what you do to an object file in order to produce another kind of object file (e.g., an executable program or a shared library). You perform this using the link editing program, ld(1).

The following general restrictions and notes also apply to the discussion below:

- Paths given to the create function command must be absolute paths (i.e., start with "/") that refer to directories visible on the machine on which the Postgres server is running.

  ### Tip

  Relative paths do in fact work, but are relative to the directory where the database resides (which is generally invisible to the frontend application). Obviously, it makes no sense to make the path relative to the directory in which the user started the frontend application, since the server could be running on a completely different machine!

- The Postgres user must be able to traverse the path given to the create function command and be able to read the object file. This is because the Postgres server runs as the Postgres user, not as the user who starts up the frontend process. (Making the file or a higher-level directory unreadable and/or unexecutable by the "postgres" user is an extremely common mistake.)

- Symbol names defined within object files must not conflict with each other or with symbols defined in Postgres.
- The GNU C compiler usually does not provide the special options that are required to use the operating system's dynamic loader interface. In such cases, the C compiler that comes with the operating system must be used.

46.1. ULTRIX

It is very easy to build dynamically-loaded object files under ULTRIX. ULTRIX does not have any shared library mechanism and hence does not place any restrictions on the dynamic loader interface. On the other hand, we had to (re)write a non-portable dynamic loader ourselves and could not use true shared libraries. Under ULTRIX, the only restriction is that you must produce each object file with the option -G 0. (Notice that that's the numeral ``0'' and not the letter ``O''). For example,

```
# simple ULTRIX example
% cc -G 0 -c foo.c
```

produces an object file called foo.o that can then be dynamically loaded into Postgres. No additional loading or link-editing must be performed.

46.2. DEC OSF/1

Under DEC OSF/1, you can take any simple object file and produce a shared object file by running the ld command over it with the correct options. The commands to do this look like:

```
# simple DEC OSF/1 example
% cc -c foo.c
% ld -shared -expect_unresolved '*' -o foo.so foo.o
```

The resulting shared object file can then be loaded into Postgres. When specifying the object file name to the create function command, one must give it the name of the shared object file (ending in .so) rather than the simple object file.

Tip

Actually, Postgres does not care what you name the file as long as it is a shared object file. If you prefer to name your shared object files with the extension .o, this is fine with Postgres so long as you make sure that the correct file name is given to the create function command. In other words, you must simply be consistent. However, from a pragmatic point of view, we discourage this practice because you will undoubtedly confuse yourself with regards to which files have been made into shared object files and which have not. For example, it's very hard to write Makefiles to do the link-editing automatically if both the object file and the shared object file end in .o!

If the file you specify is not a shared object, the backend will hang!

46.3. SunOS 4.x, Solaris 2.x and HP-UX

Under SunOS 4.x, Solaris 2.x and HP-UX, the simple object file must be created by compiling the source file with special compiler flags and a shared library must be produced. The necessary steps with HP-UX are as follows. The +z flag to the HP-UX C compiler produces so-called "Position Independent Code" (PIC) and the +u flag removes some alignment restrictions that the PA-RISC architecture normally enforces. The object file must be turned into a shared library using the HP-UX link editor with the -b option. This sounds complicated but is actually very simple, since the commands to do it are just:

```
# simple HP-UX example
% cc +z +u -c foo.c
% ld -b -o foo.sl foo.o
```

As with the .so files mentioned in the last subsection, the create function command must be told which file is the correct file to load (i.e., you must give it the location of the shared library, or .sl file). Under SunOS 4.x, the commands look like:

```
# simple SunOS 4.x example
% cc -PIC -c foo.c
% ld -dc -dp -Bdynamic -o foo.so foo.o
```

and the equivalent lines under Solaris 2.x are:

```
# simple Solaris 2.x example
% cc -K PIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

or

```
# simple Solaris 2.x example
% gcc -fPIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

When linking shared libraries, you may have to specify some additional shared libraries (typically system libraries, such as the C and math libraries) on your ld command line.

Chapter 47. Triggers

Postgres has various client interfaces such as Perl, Tcl, Python and C, as well as two *Procedural Languages* (PL). It is also possible to call C functions as trigger actions. Note that STATEMENT-level trigger events are not supported in the current version. You can currently specify BEFORE or AFTER on INSERT, DELETE or UPDATE of a tuple as a trigger event.

47.1. Trigger Creation

If a trigger event occurs, the trigger manager (called by the Executor) initializes the global structure TriggerData *CurrentTriggerData (described below) and calls the trigger function to handle the event.

The trigger function must be created before the trigger is created as a function taking no arguments and returns opaque.

The syntax for creating triggers is as follows:

```
CREATE TRIGGER <trigger name> <BEFORE|AFTER> <INSERT|DELETE|UPDATE>
    ON <relation name> FOR EACH <ROW|STATEMENT>
    EXECUTE PROCEDURE <procedure name> (<function args>);
```

The name of the trigger is used if you ever have to delete the trigger. It is used as an argument to the DROP TRIGGER command.

The next word determines whether the function is called before or after the event.

The next element of the command determines on what event(s) will trigger the function. Multiple events can be specified separated by OR.

The relation name determines which table the event applies to.

The FOR EACH statement determines whether the trigger is fired for each affected row or before (or after) the entire statement has completed.

The procedure name is the C function called.

The args are passed to the function in the CurrentTriggerData structure. The purpose of passing arguments to the function is to allow different triggers with similar requirements to call the same function.

Also, function may be used for triggering different relations (these functions are named as "general trigger functions").

As example of using both features above, there could be a general function that takes as its arguments two field names and puts the current user in one and the current timestamp in the other. This allows triggers to be written on INSERT events to automatically track creation of records in a transaction table for example. It could also be used as a "last updated" function if used in an UPDATE event.

Trigger functions return HeapTuple to the calling Executor. This is ignored for triggers fired after an INSERT, DELETE or UPDATE operation but it allows BEFORE triggers to: - return NULL to skip the operation for the current tuple (and so the tuple will not be inserted/updated/deleted); - return a pointer to another tuple (INSERT and UPDATE only) which will be inserted (as the new version of the updated tuple if UPDATE) instead of original tuple.

Note, that there is no initialization performed by the CREATE TRIGGER handler. This will be changed in the future. Also, if more than one trigger is defined for the same event on the same relation, the order of trigger firing is unpredictable. This may be changed in the future.

If a trigger function executes SQL-queries (using SPI) then these queries may fire triggers again. This is known as cascading triggers. There is no explicit limitation on the number of cascade levels.

If a trigger is fired by INSERT and inserts a new tuple in the same relation then this trigger will be fired again. Currently, there is nothing provided for synchronization (etc) of these cases but this may change. At the moment, there is function funny_dup17() in the regress tests which uses some techniques to stop recursion (cascading) on itself...

47.2. Interaction with the Trigger Manager

As mentioned above, when function is called by the trigger manager, structure TriggerData *CurrentTriggerData is NOT NULL and initialized. So it is better to check CurrentTriggerData against being NULL at the start and set it to NULL just after fetching the information to prevent calls to a trigger function not from the trigger manager.

struct TriggerData is defined in src/include/commands/trigger.h:

```
typedef struct TriggerData
{
 TriggerEvent tg_event;
 Relation tg_relation;
 HeapTuple tg_trigtuple;
 HeapTuple tg_newtuple;
 Trigger  *tg_trigger;
} TriggerData;

tg_event
   describes event for which the function is called. You may use the
   following macros to examine tg_event:

   TRIGGER_FIRED_BEFORE(event) returns TRUE if trigger fired BEFORE;
   TRIGGER_FIRED_AFTER(event) returns TRUE if trigger fired AFTER;
   TRIGGER_FIRED_FOR_ROW(event) returns TRUE if trigger fired for
                                ROW-level event;
   TRIGGER_FIRED_FOR_STATEMENT(event) returns TRUE if trigger fired
 for
                                STATEMENT-level event;
   TRIGGER_FIRED_BY_INSERT(event) returns TRUE if trigger fired by
 INSERT;
   TRIGGER_FIRED_BY_DELETE(event) returns TRUE if trigger fired by
 DELETE;
   TRIGGER_FIRED_BY_UPDATE(event) returns TRUE if trigger fired by
 UPDATE.

tg_relation
   is pointer to structure describing the triggered relation. Look at
   src/include/utils/rel.h for details about this structure.  The most
   interest things are tg_relation->rd_att (descriptor of the relation
   tuples) and tg_relation->rd_rel->relname (relation's name. This is
 not
   char*, but NameData.  Use SPI_getrelname(tg_relation) to get char*
 if
   you need a copy of name).
```

```
tg_trigtuple
   is a pointer to the tuple for which the trigger is fired. This is
 the tuple
   being inserted (if INSERT), deleted (if DELETE) or updated (if
 UPDATE).
   If INSERT/DELETE then this is what you are to return to Executor
 if
   you don't want to replace tuple with another one (INSERT) or skip
 the
   operation.

tg_newtuple
   is a pointer to the new version of tuple if UPDATE and NULL if this
 is
   for an INSERT or a DELETE. This is what you are to return to
 Executor if
   UPDATE and you don't want to replace this tuple with another one or
 skip
   the operation.

tg_trigger
   is pointer to structure Trigger defined in src/include/utils/rel.h:

typedef struct Trigger
{
 char  *tgname;
 Oid  tgfoid;
 func_ptr tgfunc;
 int16  tgtype;
 int16  tgnargs;
 int16  tgattr[8];
 char  **tgargs;
} Trigger;

   tgname is the trigger's name, tgnargs is number of arguments in
 tgargs,
   tgargs is an array of pointers to the arguments specified in the
 CREATE
   TRIGGER statement. Other members are for internal use only.
```

47.3. Visibility of Data Changes

Postgres data changes visibility rule: during a query execution, data changes made by the query itself (via SQL-function, SPI-function, triggers) are invisible to the query scan. For example, in query

```
   INSERT INTO a SELECT * FROM a
```

tuples inserted are invisible for SELECT' scan. In effect, this duplicates the database table within itself (subject to unique index rules, of course) without recursing.

But keep in mind this notice about visibility in the SPI documentation:

```
   Changes made by query Q are visible by queries which are started
 after
```

```
   query Q, no matter whether they are started inside Q (during the
   execution of Q) or after Q is done.
```

This is true for triggers as well so, though a tuple being inserted (tg_trigtuple) is not visible to queries in a BEFORE trigger, this tuple (just inserted) is visible to queries in an AFTER trigger, and to queries in BEFORE/AFTER triggers fired after this!

47.4. Examples

There are more complex examples in in src/test/regress/regress.c and in contrib/spi.

Here is a very simple example of trigger usage. Function trigf reports the number of tuples in the triggered relation ttest and skips the operation if the query attempts to insert NULL into x (i.e - it acts as a NOT NULL constraint but doesn't abort the transaction).

```
#include "executor/spi.h" /* this is what you need to work with SPI */
#include "commands/trigger.h" /* -"- and triggers */

HeapTuple  trigf(void);

HeapTuple
trigf()
{
 TupleDesc tupdesc;
 HeapTuple rettuple;
 char  *when;
 bool  checknull = false;
 bool  isnull;
 int  ret, i;

 if (!CurrentTriggerData)
  elog(WARN, "trigf: triggers are not initialized");

 /* tuple to return to Executor */
 if (TRIGGER_FIRED_BY_UPDATE(CurrentTriggerData->tg_event))
  rettuple = CurrentTriggerData->tg_newtuple;
 else
  rettuple = CurrentTriggerData->tg_trigtuple;

 /* check for NULLs ? */
 if (!TRIGGER_FIRED_BY_DELETE(CurrentTriggerData->tg_event) &&
  TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
  checknull = true;

 if (TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
  when = "before";
 else
  when = "after ";

 tupdesc = CurrentTriggerData->tg_relation->rd_att;
 CurrentTriggerData = NULL;

 /* Connect to SPI manager */
 if ((ret = SPI_connect()) < 0)
```

```
  elog(WARN, "trigf (fired %s): SPI_connect returned %d", when, ret);

 /* Get number of tuples in relation */
 ret = SPI_exec("select count(*) from ttest", 0);

 if (ret < 0)
  elog(WARN, "trigf (fired %s): SPI_exec returned %d", when, ret);

 i = SPI_getbinval(SPI_tuptable->vals[0], SPI_tuptable->tupdesc, 1,
 &isnull);

 elog (NOTICE, "trigf (fired %s): there are %d tuples in ttest", when,
 i);

 SPI_finish();

 if (checknull)
 {
  i = SPI_getbinval(rettuple, tupdesc, 1, &isnull);
  if (isnull)
   rettuple = NULL;
 }

 return (rettuple);
}
```

Now, compile and create table ttest (x int4); create function trigf () returns opaque as '...path_to_so' language 'c';

```
vac=> create trigger tbefore before insert or update or delete on
 ttest
for each row execute procedure trigf();
CREATE
vac=> create trigger tafter after insert or update or delete on ttest
for each row execute procedure trigf();
CREATE
vac=> insert into ttest values (null);
NOTICE:trigf (fired before): there are 0 tuples in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

vac=> select * from ttest;
x
-
(0 rows)

vac=> insert into ttest values (1);
NOTICE:trigf (fired before): there are 0 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
                                       ^^^^^^^^
                              remember what we said about visibility.
INSERT 167793 1
vac=> select * from ttest;
```

```
x
-
1
(1 row)

vac=> insert into ttest select x * 2 from ttest;
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167794 1
vac=> select * from ttest;
x
-
1
2
(2 rows)

vac=> update ttest set x = null where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
UPDATE 0
vac=> update ttest set x = 4 where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
UPDATE 1
vac=> select * from ttest;
x
-
1
4
(2 rows)

vac=> delete from ttest;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 0 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
DELETE 2
vac=> select * from ttest;
x
-
(0 rows)
```

Chapter 48. Server Programming Interface

Vadim Mikheev

The *Server Programming Interface* (SPI) gives users the ability to run SQL queries inside user-defined C functions. The available Procedural Languages (PL) give an alternate means to access these capabilities.

In fact, SPI is just a set of native interface functions to simplify access to the Parser, Planner, Optimizer and Executor. SPI also does some memory management.

To avoid misunderstanding we'll use *function* to mean SPI interface functions and *procedure* for user-defined C-functions using SPI.

SPI procedures are always called by some (upper) Executor and the SPI manager uses the Executor to run your queries. Other procedures may be called by the Executor running queries from your procedure.

Note, that if during execution of a query from a procedure the transaction is aborted then control will not be returned to your procedure. Rather, all work will be rolled back and the server will wait for the next command from the client. This will be changed in future versions.

Other restrictions are the inability to execute BEGIN, END and ABORT (transaction control statements) and cursor operations. This will also be changed in the future.

If successful, SPI functions return a non-negative result (either via a returned integer value or in SPI_result global variable, as described below). On error, a negative or NULL result will be returned.

48.1. Interface Functions

Name

SPI_connect — Connects your procedure to the SPI manager.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
int SPI_connect(void)
```

Inputs

None

Outputs

int

Return status

SPI_OK_CONNECT

if connected

SPI_ERROR_CONNECT

if not connected

Description

`SPI_connect` opens a connection to the Postgres backend. You should call this function if you will need to execute queries. Some utility SPI functions may be called from un-connected procedures.

You may get SPI_ERROR_CONNECT error if `SPI_connect` is called from an already connected procedure - e.g. if you directly call one procedure from another connected one. Actually, while the child procedure will be able to use SPI, your parent procedure will not be able to continue to use SPI after the child returns (if `SPI_finish` is called by the child). It's bad practice.

Usage

XXX thomas 1997-12-24

Algorithm

`SPI_connect` performs the following:

- Initializes the SPI internal structures for query execution and memory management.

Name

SPI_finish — Disconnects your procedure from the SPI manager.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_finish(void)
```

Inputs

None

Outputs

int

SPI_OK_FINISH if properly disconnected
SPI_ERROR_UNCONNECTED if called from an un-connected procedure

Description

`SPI_finish` closes an existing connection to the Postgres backend. You should call this function after completing operations through the SPI manager.

You may get the error return SPI_ERROR_UNCONNECTED if `SPI_finish` is called without having a current valid connection. There is no fundamental problem with this; it means that nothing was done by the SPI manager.

Usage

`SPI_finish` *must* be called as a final step by a connected procedure or you may get unpredictable results! Note that you can safely skip the call to `SPI_finish` if you abort the transaction (via elog(ERROR)).

Algorithm

`SPI_finish` performs the following:

- Disconnects your procedure from the SPI manager and frees all memory allocations made by your procedure via `palloc` since the `SPI_connect`. These allocations can't be used any more! See Memory management.

Name

SPI_exec — Creates an execution plan (parser+planner+optimizer) and executes a query.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_exec(query, tcount)
```

Inputs

char *query

String containing query plan

int tcount

Maximum number of tuples to return

Outputs

int

SPI_OK_EXEC if properly disconnected
SPI_ERROR_UNCONNECTED if called from an un-connected procedure
SPI_ERROR_ARGUMENT if query is NULL or tcount < 0.
SPI_ERROR_UNCONNECTED if procedure is unconnected.
SPI_ERROR_COPY if COPY TO/FROM stdin.
SPI_ERROR_CURSOR if DECLARE/CLOSE CURSOR, FETCH.
SPI_ERROR_TRANSACTION if BEGIN/ABORT/END.
SPI_ERROR_OPUNKNOWN if type of query is unknown (this shouldn't occur).

If execution of your query was successful then one of the following (non-negative) values will be returned:

SPI_OK_UTILITY if some utility (e.g. CREATE TABLE ...) was executed
SPI_OK_SELECT if SELECT (but not SELECT ... INTO!) was executed
SPI_OK_SELINTO if SELECT ... INTO was executed
SPI_OK_INSERT if INSERT (or INSERT ... SELECT) was executed
SPI_OK_DELETE if DELETE was executed
SPI_OK_UPDATE if UPDATE was executed

Description

SPI_exec creates an execution plan (parser+planner+optimizer) and executes the query for tcount tuples.

Usage

This should only be called from a connected procedure. If tcount is zero then it executes the query for all tuples returned by the query scan. Using tcount > 0 you may restrict the number of tuples for which the query will be executed. For example,

```
SPI_exec ("insert into table select * from table", 5);
```

will allow at most 5 tuples to be inserted into table. If execution of your query was successful then a non-negative value will be returned.

Note

You may pass many queries in one string or query string may be re-written by RULEs. `SPI_exec` returns the result for the last query executed.

The actual number of tuples for which the (last) query was executed is returned in the global variable SPI_processed (if not SPI_OK_UTILITY). If SPI_OK_SELECT returned and SPI_processed > 0 then you may use global pointer SPITupleTable *SPI_tuptable to access the selected tuples: Also NOTE, that `SPI_finish` frees and makes all SPITupleTables unusable! (See Memory management).

`SPI_exec` may return one of the following (negative) values:

SPI_ERROR_ARGUMENT if query is NULL or *`tcount`* < 0.
SPI_ERROR_UNCONNECTED if procedure is unconnected.
SPI_ERROR_COPY if COPY TO/FROM stdin.
SPI_ERROR_CURSOR if DECLARE/CLOSE CURSOR, FETCH.
SPI_ERROR_TRANSACTION if BEGIN/ABORT/END.
SPI_ERROR_OPUNKNOWN if type of query is unknown (this shouldn't occur).

Algorithm

`SPI_exec` performs the following:

- Disconnects your procedure from the SPI manager and frees all memory allocations made by your procedure via `palloc` since the `SPI_connect`. These allocations can't be used any more! See Memory management.

Name

SPI_prepare — Connects your procedure to the SPI manager.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_prepare(query, nargs, argtypes)
```

Inputs

query

Query string

nargs

Number of input parameters ($1 ... $nargs - as in SQL-functions)

argtypes

Pointer list of type OIDs to input arguments

Outputs

void *

Pointer to an execution plan (parser+planner+optimizer)

Description

`SPI_prepare` creates and returns an execution plan (parser+planner+optimizer) but doesn't execute the query. Should only be called from a connected procedure.

Usage

nargs is number of parameters ($1 ... $nargs - as in SQL-functions), and nargs may be 0 only if there is not any $1 in query.

Execution of prepared execution plans is sometimes much faster so this feature may be useful if the same query will be executed many times.

The plan returned by `SPI_prepare` may be used only in current invocation of the procedure since `SPI_finish` frees memory allocated for a plan. See `SPI_saveplan`.

If successful, a non-null pointer will be returned. Otherwise, you'll get a NULL plan. In both cases SPI_result will be set like the value returned by SPI_exec, except that it is set to SPI_ERROR_ARGUMENT if query is NULL or nargs < 0 or nargs > 0 && argtypes is NULL.

Name

SPI_saveplan — Saves a passed plan

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_saveplan(plan)
```

Inputs

void *query

Passed plan

Outputs

void *

Execution plan location. NULL if unsuccessful.

SPI_result

SPI_ERROR_ARGUMENT if plan is NULL
SPI_ERROR_UNCONNECTED if procedure is un-connected

Description

`SPI_saveplan` stores a plan prepared by `SPI_prepare` in safe memory protected from freeing by `SPI_finish` or the transaction manager.

In the current version of Postgres there is no ability to store prepared plans in the system catalog and fetch them from there for execution. This will be implemented in future versions. As an alternative, there is the ability to reuse prepared plans in the consequent invocations of your procedure in the current session. Use `SPI_execp` to execute this saved plan.

Usage

`SPI_saveplan` saves a passed plan (prepared by `SPI_prepare`) in memory protected from freeing by `SPI_finish` and by the transaction manager and returns a pointer to the saved plan. You may save the pointer returned in a local variable. Always check if this pointer is NULL or not either when preparing a plan or using an already prepared plan in SPI_execp (see below).

Note

If one of the objects (a relation, function, etc.) referenced by the prepared plan is dropped during your session (by your backend or another process) then the results of `SPI_execp` for this plan will be unpredictable.

Name

SPI_execp — Executes a plan from `SPI_saveplan`

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_execp(plan,
values,
nulls,
tcount)
```

Inputs

void **plan*

Execution plan

Datum **values*

Actual parameter values

char **nulls*

Array describing what parameters get NULLs

'n' indicates NULL allowed
' ' indicates NULL not allowed

int *tcount*

Number of tuples for which plan is to be executed

Outputs

int

Returns the same value as `SPI_exec` as well as

SPI_ERROR_ARGUMENT if *plan* is NULL or *tcount* < 0
SPI_ERROR_PARAM if *values* is NULL and *plan* was prepared with some parameters.

SPI_tuptable

initialized as in `SPI_exec` if successful

SPI_processed

initialized as in `SPI_exec` if successful

Description

`SPI_execp` stores a plan prepared by `SPI_prepare` in safe memory protected from freeing by `SPI_finish` or the transaction manager.

In the current version of Postgres there is no ability to store prepared plans in the system catalog and fetch them from there for execution. This will be implemented in future versions. As a work arround, there is

the ability to reuse prepared plans in the consequent invocations of your procedure in the current session. Use `SPI_execp` to execute this saved plan.

Usage

If *`nulls`* is NULL then `SPI_execp` assumes that all values (if any) are NOT NULL.

Note

If one of the objects (a relation, function, etc.) referenced by the prepared plan is dropped during your session (by your backend or another process) then the results of `SPI_execp` for this plan will be unpredictable.

48.2. Interface Support Functions

All functions described below may be used by connected and unconnected procedures.

Name

SPI_copytuple — Makes copy of tuple in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_copytuple(tuple)
```

Inputs

HeapTuple *tuple*

Input tuple to be copied

Outputs

HeapTuple

Copied tuple

non-NULL if *tuple* is not NULL and the copy was successful
NULL only if *tuple* is NULL

Description

`SPI_copytuple` makes a copy of tuple in upper Executor context. See the section on Memory Management.

Usage

TBD

Name

SPI_modifytuple — Modifies tuple of relation

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_modifytuple(rel, tuple , nattrs
, attnum , Values , Nulls)
```

Inputs

Relation *rel*

HeapTuple *tuple*

Input tuple to be modified

int *nattrs*

Number of attribute numbers in attnum

int * *attnum*

Array of numbers of the attributes which are to be changed

Datum * *Values*

New values for the attributes specified

char * *Nulls*

Which attributes are NULL, if any

Outputs

HeapTuple

New tuple with modifications

non-NULL if *tuple* is not NULL and the modify was successful
NULL only if *tuple* is NULL

SPI_result

SPI_ERROR_ARGUMENT if rel is NULL or tuple is NULL or natts ≤ 0 or attnum is NULL or Values is NULL.
SPI_ERROR_NOATTRIBUTE if there is an invalid attribute number in attnum (attnum ≤ 0 or > number of attributes in tuple)

Description

`SPI_modifytuple` Modifies a tuple in upper Executor context. See the section on Memory Management.

Usage

If successful, a pointer to the new tuple is returned. The new tuple is allocated in upper Executor context (see Memory management). Passed tuple is not changed.

Name

SPI_fnumber — Finds the attribute number for specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_fnumber(tupdesc, fname)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

char * *fname*

Field name

Outputs

int

Attribute number

Valid one-based index number of attribute
SPI_ERROR_NOATTRIBUTE if the named attribute is not found

Description

`SPI_fnumber` returns the attribute number for the attribute with name in fname.

Usage

Attribute numbers are 1 based.

Name

SPI_fname — Finds the attribute name for the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_fname(tupdesc, fname)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

char * *fnumber*

Attribute number

Outputs

char *

Attribute name

NULL if fnumber is out of range
SPI_result set to SPI_ERROR_NOATTRIBUTE on error

Description

`SPI_fname` returns the attribute name for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Returns a newly-allocated copy of the attribute name.

Name

SPI_getvalue — Returns the string value of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getvalue(tuple, tupdesc, fnumber)
```

Inputs

HeapTuple *tuple*

Input tuple to be examined

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

Attribute value or NULL if

attribute is NULL
fnumber is out of range (SPI_result set to SPI_ERROR_NOATTRIBUTE)
no output function available (SPI_result set to SPI_ERROR_NOOUTFUNC)

Description

`SPI_getvalue` returns an external (string) representation of the value of the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Allocates memory as required by the value.

Name

SPI_getbinval — Returns the binary value of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getbinval(tuple, tupdesc, fnumber, isnull)
```

Inputs

HeapTuple *tuple*

Input tuple to be examined

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

Datum

Attribute binary value

bool * *isnull*

flag for null value in attribute

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_getbinval` returns the binary value of the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Does not allocate new space for the binary value.

Name

SPI_gettype — Returns the type name of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_gettype(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

The type name for the specified attribute number

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_gettype` returns a copy of the type name for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Does not allocate new space for the binary value.

Name

SPI_gettypeid — Returns the type OID of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_gettypeid(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

OID

The type OID for the specified attribute number

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_gettypeid` returns the type OID for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

TBD

Name

SPI_getrelname — Returns the name of the specified relation

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getrelname(rel)
```

Inputs

Relation *rel*

Input relation

Outputs

char *

The name of the specified relation

Description

`SPI_getrelname` returns the name of the specified relation.

Usage

TBD

Algorithm

Copies the relation name into new storage.

Name

SPI_palloc — Allocates memory in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_palloc(size)
```

Inputs

Size *size*

Octet size of storage to allocate

Outputs

void *

New storage space of specified size

Description

`SPI_palloc` allocates memory in upper Executor context. See section on memory management.

Usage

TBD

Name

SPI_repalloc — Re-allocates memory in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_repalloc(pointer, size)
```

Inputs

void * *pointer*

Pointer to existing storage

Size *size*

Octet size of storage to allocate

Outputs

void *

New storage space of specified size with contents copied from existing area

Description

`SPI_repalloc` re-allocates memory in upper Executor context. See section on memory management.

Usage

TBD

Name

SPI_pfree — Frees memory from upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_pfree(pointer)
```

Inputs

void * *pointer*

Pointer to existing storage

Outputs

None

Description

`SPI_pfree` frees memory in upper Executor context. See section on memory management.

Usage

TBD

48.3. Memory Management

Server allocates memory in memory contexts in such way that allocations made in one context may be freed by context destruction without affecting allocations made in other contexts. All allocations (via `palloc`, etc) are made in the context which are chosen as current one. You'll get unpredictable results if you'll try to free (or reallocate) memory allocated not in current context.

Creation and switching between memory contexts are subject of SPI manager memory management.

SPI procedures deal with two memory contexts: upper Executor memory context and procedure memory context (if connected).

Before a procedure is connected to the SPI manager, current memory context is upper Executor context so all allocation made by the procedure itself via `palloc`/`repalloc` or by SPI utility functions before connecting to SPI are made in this context.

After `SPI_connect` is called current context is the procedure's one. All allocations made via `palloc`/`repalloc` or by SPI utility functions (except for `SPI_copytuple`, `SPI_modifytuple`, `SPI_palloc` and `SPI_repalloc`) are made in this context.

When a procedure disconnects from the SPI manager (via `SPI_finish`) the current context is restored to the upper Executor context and all allocations made in the procedure memory context are freed and can't be used any more!

If you want to return something to the upper Executor then you have to allocate memory for this in the upper context!

SPI has no ability to automatically free allocations in the upper Executor context!

SPI automatically frees memory allocated during execution of a query when this query is done!

48.4. Visibility of Data Changes

Postgres data changes visibility rule: during a query execution, data changes made by the query itself (via SQL-function, SPI-function, triggers) are invisible to the query scan. For example, in query INSERT INTO a SELECT * FROM a tuples inserted are invisible for SELECT' scan. In effect, this duplicates the database table within itself (subject to unique index rules, of course) without recursing.

Changes made by query Q are visible by queries which are started after query Q, no matter whether they are started inside Q (during the execution of Q) or after Q is done.

48.5. Examples

This example of SPI usage demonstrates the visibility rule. There are more complex examples in in src/test/regress/regress.c and in contrib/spi.

This is a very simple example of SPI usage. The procedure execq accepts an SQL-query in its first argument and tcount in its second, executes the query using SPI_exec and returns the number of tuples for which the query executed:

```
#include "executor/spi.h" /* this is what you need to work with SPI */

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
 int ret;
 int proc = 0;

 SPI_connect();

 ret = SPI_exec(textout(sql), cnt);

 proc = SPI_processed;
 /*
  * If this is SELECT and some tuple(s) fetched -
  * returns tuples to the caller via elog (NOTICE).
  */
 if ( ret == SPI_OK_SELECT && SPI_processed > 0 )
 {
  TupleDesc tupdesc = SPI_tuptable->tupdesc;
  SPITupleTable *tuptable = SPI_tuptable;
  char buf[8192];
  int i;

  for (ret = 0; ret < proc; ret++)
  {
   HeapTuple tuple = tuptable->vals[ret];
```

```
   for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
    sprintf(buf + strlen (buf), " %s%s",
     SPI_getvalue(tuple, tupdesc, i),
     (i == tupdesc->natts) ? " " : " |");
   elog (NOTICE, "EXECQ: %s", buf);
  }
 }

 SPI_finish();

 return (proc);
}
```

Now, compile and create the function:

```
create function execq (text, int4) returns int4 as '...path_to_so'
 language 'c';

vac=> select execq('create table a (x int4)', 0);
execq
-----
    0
(1 row)

vac=> insert into a values (execq('insert into a values (0)',0));
INSERT 167631 1
vac=> select execq('select * from a',0);
NOTICE:EXECQ:  0 <<< inserted by execq

NOTICE:EXECQ:  1 <<< value returned by execq and inserted by upper
 INSERT

execq
-----
    2
(1 row)

vac=> select execq('insert into a select x + 2 from a',1);
execq
-----
    1
(1 row)

vac=> select execq('select * from a', 10);
NOTICE:EXECQ:  0

NOTICE:EXECQ:  1

NOTICE:EXECQ:  2 <<< 0 + 2, only one tuple inserted - as specified

execq
-----
    3            <<< 10 is max value only, 3 is real # of tuples
(1 row)
```

```
vac=> delete from a;
DELETE 3
vac=> insert into a values (execq('select * from a', 0) + 1);
INSERT 167712 1
vac=> select * from a;
x
-
1               <<< no tuples in a (0) + 1
(1 row)

vac=> insert into a values (execq('select * from a', 0) + 1);
NOTICE:EXECQ:  0
INSERT 167713 1
vac=> select * from a;
x
-
1
2               <<< there was single tuple in a + 1
(2 rows)

--   This demonstrates data changes visibility rule:

vac=> insert into a select execq('select * from a', 0) * x from a;
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  2
INSERT 0 2
vac=> select * from a;
x
-
1
2
2               <<< 2 tuples * 1 (x in first tuple)
6               <<< 3 tuples (2 + 1 just inserted) * 2 (x in second
 tuple)
(4 rows)            ^^^^^^^^
                    tuples visible to execq() in different
 invocations
```

Chapter 49. Procedural Languages

Beginning with the release of version 6.3, Postgres supports the definition of procedural languages. In the case of a function or trigger procedure defined in a procedural language, the database has no builtin knowlege how to interpret the functions source text. Instead, the calls are passed into a handler that knows the details of the language. The handler itself is a special programming language function compiled into a shared object and loaded on demand.

49.1. Installing Procedural Languages

Procedural Language Installation

A procedural language is installed in the database in three steps.

1. The shared object for the language handler must be compiled and installed. By default the handler for PL/pgSQL is built and installed into the database library directory. If Tcl/Tk support is configured in, the handler for PL/Tcl is also built and installed in the same location.

   Writing a handler for a new procedural language (PL) is outside the scope of this manual.

2. The handler must be declared with the command

   ```
   CREATE FUNCTION handler_function_name () RETURNS OPAQUE AS
       'path-to-shared-object' LANGUAGE 'C';
   ```

   The special return type of OPAQUE tells the database, that this function does not return one of the defined base- or composite types and is not directly usable in SQL statements.

3. The PL must be declared with the command

   ```
   CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'language-name'
       HANDLER handler_function_name
LANCOMPILER 'description';
   ```

   The optional keyword TRUSTED tells if ordinary database users that have no superuser privileges can use this language to create functions and trigger procedures. Since PL functions are executed inside the database backend it should only be used for languages that don't gain access to database backends internals or the filesystem. The languages PL/pgSQL and PL/Tcl are known to be trusted.

Example

1. The following command tells the database where to find the shared object for the PL/pgSQL languages call handler function.

   ```
   CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
       '/usr/local/pgsql/lib/plpgsql.so' LANGUAGE 'C';
   ```

2. The command

   ```
   CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
   ```

```
HANDLER plpgsql_call_handler
LANCOMPILER 'PL/pgSQL';
```

then defines that the previously declared call handler function should be invoked for functions and trigger procedures where the language attribute is 'plpgsql'.

PL handler functions have a special call interface that is different from regular C language functions. One of the arguments given to the handler is the object ID in the `pg_proc` tables entry for the function that should be executed. The handler examines various system catalogs to analyze the functions call arguments and it's return data type. The source text of the functions body is found in the prosrc attribute of `pg_proc`. Due to this, in contrast to C language functions, PL functions can be overloaded like SQL language functions. There can be multiple different PL functions having the same function name, as long as the call arguments differ.

Procedural languages defined in the `template1` database are automatically defined in all subsequently created databases. So the database administrator can decide which languages are available by default.

49.2. PL/pgSQL

PL/pgSQL is a loadable procedural language for the Postgres database system.

This package was originally written by Jan Wieck.

49.2.1. Overview

The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user defined types, functions and operators,
- can be defined to be trusted by the server,
- is easy to use.

The PL/pgSQL call handler parses the functions source text and produces an internal binary instruction tree on the first time, the function is called by a backend. The produced bytecode is identified in the call handler by the object ID of the function. This ensures, that changing a function by a DROP/CREATE sequence will take effect without establishing a new database connection.

For all expressions and SQL statements used in the function, the PL/pgSQL bytecode interpreter creates a prepared execution plan using the SPI managers SPI_prepare() and SPI_saveplan() functions. This is done the first time, the individual statement is processed in the PL/pgSQL function. Thus, a function with conditional code that contains many statements for which execution plans would be required, will only prepare and save those plans that are really used during the entire lifetime of the database connection.

Except for input-/output-conversion and calculation functions for user defined types, anything that can be defined in C language functions can also be done with PL/pgSQL. It is possible to create complex conditional computation functions and later use them to define operators or use them in functional indices.

49.2.2. Description

49.2.2.1. Structure of PL/pgSQL

The PL/pgSQL language is case insensitive. All keywords and identifiers can be used in mixed upper- and lowercase.

PL/pgSQL is a block oriented language. A block is defined as

```
[<<label>>]
[DECLARE
    declarations]
BEGIN
    statements
END;
```

There can be any number of subblocks in the statement section of a block. Subblocks can be used to hide variables from outside a block of statements. The variables declared in the declarations section preceding a block are initialized to their default values every time the block is entered, not only once per function call.

It is important not to misunderstand the meaning of BEGIN/END for grouping statements in PL/pgSQL and the database commands for transaction control. Functions and trigger procedures cannot start or commit transactions and Postgres does not have nested transactions.

49.2.2.2. Comments

There are two types of comments in PL/pgSQL. A double dash '--' starts a comment that extends to the end of the line. A '/*' starts a block comment that extends to the next occurence of '*/'. Block comments cannot be nested, but double dash comments can be enclosed into a block comment and a double dash can hide the block comment delimiters '/*' and '*/'.

49.2.2.3. Declarations

All variables, rows and records used in a block or it's subblocks must be declared in the declarations section of a block except for the loop variable of a FOR loop iterating over a range of integer values. Parameters given to a PL/pgSQL function are automatically declared with the usual identifiers $n. The declarations have the following syntax:

name [CONSTANT] *type* [NOT NULL] [DEFAULT | := *value*];

> Declares a variable of the specified base type. If the variable is declared as CONSTANT, the value cannot be changed. If NOT NULL is specified, an assignment of a NULL value results in a runtime error. Since the default value of all variables is the SQL NULL value, all variables declared as NOT NULL must also have a default value specified.
>
> The default value is evaluated ever time the function is called. So assigning '*now*' to a variable of type *datetime* causes the variable to have the time of the actual function call, not when the function was precompiled into it's bytecode.

name class%ROWTYPE;

> Declares a row with the structure of the given class. Class must be an existing table- or viewname of the database. The fields of the row are accessed in the dot notation. Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a rowtype,

but it must be aliased using the ALIAS command described below. Only the user attributes of a table row are accessible in the row, no Oid or other system attributes (hence the row could be from a view and view rows don't have useful system attributes).

The fields of the rowtype inherit the tables fieldsizes or precision for char() etc. data types.

name RECORD;

Records are similar to rowtypes, but they have no predefined structure. They are used in selections and FOR loops to hold one actual database row from a SELECT operation. One and the same record can be used in different selections. Accessing a record or an attempt to assign a value to a record field when there is no actual row in it results in a runtime error.

The NEW and OLD rows in a trigger are given to the procedure as records. This is necessary because in Postgres one and the same trigger procedure can handle trigger events for different tables.

name ALIAS FOR $n;

For better readability of the code it is possible to define an alias for a positional parameter to a function.

This aliasing is required for composite types given as arguments to a function. The dot notation $1.salary as in SQL functions is not allowed in PL/pgSQL.

RENAME *oldname* TO *newname*;

Change the name of a variable, record or row. This is useful if NEW or OLD should be referenced by another name inside a trigger procedure.

49.2.2.4. Data Types

The type of a varible can be any of the existing basetypes of the database. *type* in the declarations section above is defined as:

- Postgres-basetype
- *variable*%TYPE
- *class.field*%TYPE

variable is the name of a variable, previously declared in the same function, that is visible at this point.

class is the name of an existing table or view where *field* is the name of an attribute.

Using the *class.field*%TYPE causes PL/pgSQL to lookup the attributes definitions at the first call to the funciton during the lifetime of a backend. Have a table with a char(20) attribute and some PL/pgSQL functions that deal with it's content in local variables. Now someone decides that char(20) isn't enough, dumps the table, drops it, recreates it now with the attribute in question defined as char(40) and restores the data. Ha - he forgot about the funcitons. The computations inside them will truncate the values to 20 characters. But if they are defined using the *class.field*%TYPE declarations, they will automagically handle the size change or if the new table schema defines the attribute as text type.

49.2.2.5. Expressions

All expressions used in PL/pgSQL statements are processed using the backends executor. Expressions which appear to contain constants may in fact require run-time evaluation (e.g. 'now' for the datetime type)

so it is impossible for the PL/pgSQL parser to identify real constant values other than the NULL keyword. All expressions are evaluated internally by executing a query

```
SELECT expression
```

using the SPI manager. In the expression, occurences of variable identifiers are substituted by parameters and the actual values from the variables are passed to the executor in the parameter array. All expressions used in a PL/pgSQL function are only prepared and saved once.

The type checking done by the Postgres main parser has some side effects to the interpretation of constant values. In detail there is a difference between what the two functions

```
CREATE FUNCTION logfunc1 (text) RETURNS datetime AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, ''now'');
        RETURN ''now'';
    END;
' LANGUAGE 'plpgsql';
```

and

```
CREATE FUNCTION logfunc2 (text) RETURNS datetime AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime datetime;
    BEGIN
        curtime := ''now'';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
' LANGUAGE 'plpgsql';
```

do. In the case of logfunc1(), the Postgres main parser knows when preparing the plan for the INSERT, that the string 'now' should be interpreted as datetime because the target field of logtable is of that type. Thus, it will make a constant from it at this time and this constant value is then used in all invocations of logfunc1() during the lifetime of the backend. Needless to say that this isn't what the programmer wanted.

In the case of logfunc2(), the Postgres main parser does not know what type 'now' should become and therefor it returns a datatype of text containing the string 'now'. During the assignment to the local variable curtime, the PL/pgSQL interpreter casts this string to the datetime type by calling the text_out() and datetime_in() functions for the conversion.

This type checking done by the Postgres main parser got implemented after PL/pgSQL was nearly done. It is a difference between 6.3 and 6.4 and affects all functions using the prepared plan feature of the SPI manager. Using a local variable in the above manner is currently the only way in PL/pgSQL to get those values interpreted correctly.

If record fields are used in expressions or statements, the data types of fields should not change between calls of one and the same expression. Keep this in mind when writing trigger procedures that handle events for more than one table.

49.2.2.6. Statements

Anything not understood by the PL/pgSQL parser as specified below will be put into a query and sent down to the database engine to execute. The resulting query should not return any data.

Assignment

An assignment of a value to a variable or row/record field is written as

```
identifier := expression;
```

If the expressions result data type doesn't match the variables data type, or the variable has a size/precision that is known (as for char(20)), the result value will be implicitly casted by the PL/pgSQL bytecode interpreter using the result types output- and the variables type input-functions. Note that this could potentially result in runtime errors generated by the types input functions.

An assignment of a complete selection into a record or row can be done by

```
SELECT expressions INTO target FROM ...;
```

target can be a record, a row variable or a comma separated list of variables and record-/row-fields.

if a row or a variable list is used as target, the selected values must exactly match the structure of the target(s) or a runtime error occurs. The FROM keyword can be followed by any valid qualification, grouping, sorting etc. that can be given for a SELECT statement.

There is a special variable named FOUND of type bool that can be used immediately after a SELECT INTO to check if an assignment had success.

```
SELECT * INTO myrec FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

If the selection returns multiple rows, only the first is moved into the target fields. All others are silently discarded.

Calling another function

All functions defined in a Prostgres database return a value. Thus, the normal way to call a function is to execute a SELECT query or doing an assignment (resulting in a PL/pgSQL internal SELECT). But there are cases where someone isn't interested int the functions result.

```
PERFORM query
```

executes a 'SELECT *query*' over the SPI manager and discards the result. Identifiers like local variables are still substituted into parameters.

Returning from the function

```
RETURN expression
```

The function terminates and the value of *expression* will be returned to the upper executor. The return value of a function cannot be undefined. If control reaches the end of the toplevel block of the function without hitting a RETURN statement, a runtime error will occur.

The expressions result will be automatically casted into the functions return type as described for assignments.

Aborting and messages

As indicated in the above examples there is a RAISE statement that can throw messages into the Postgres elog mechanism.

```
RAISE level ''format'' [, identifier [...]];
```

Inside the format, "%" is used as a placeholder for the subsequent comma-separated identifiers. Possible levels are DEBUG (silently suppressed in production running databases), NOTICE (written into the database log and forwarded to the client application) and EXCEPTION (written into the database log and aborting the transaction).

Conditionals

```
IF expression THEN
    statements
[ELSE
    statements]
END IF;
```

The *expression* must return a value that at least can be casted into a boolean type.

Loops

There are multiple types of loops.

```
[<<label>>]
LOOP
    statements
END LOOP;
```

An unconditional loop that must be terminated explicitly by an EXIT statement. The optional label can be used by EXIT statements of nested loops to specify which level of nesting should be terminated.

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

A conditional loop that is executed as long as the evaluation of *expression* is true.

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
```

```
END LOOP;
```

A loop that iterates over a range of integer values. The variable *name* is automatically created as type integer and exists only inside the loop. The two expressions giving the lower and upper bound of the range are evaluated only when entering the loop. The iteration step is always 1.

```
[<<label>>]
FOR record | row IN select_clause LOOP
    statements
END LOOP;
```

The record or row is assigned all the rows resulting from the select clause and the statements executed for each. If the loop is terminated with an EXIT statement, the last assigned row is still accessible after the loop.

```
EXIT [ label ] [ WHEN expression ];
```

If no *label* given, the innermost loop is terminated and the statement following END LOOP is executed next. If *label* is given, it must be the label of the current or an upper level of nested loop blocks. Then the named loop or block is terminated and control continues with the statement after the loops/blocks corresponding END.

49.2.2.7. Trigger Procedures

PL/pgSQL can be used to define trigger procedures. They are created with the usual CREATE FUNCTION command as a function with no arguments and a return type of OPAQUE.

There are some Postgres specific details in functions used as trigger procedures.

First they have some special variables created automatically in the toplevel blocks declaration section. They are

NEW

Datatype RECORD; variable holding the new database row on INSERT/UPDATE operations on ROW level triggers.

OLD

Datatype RECORD; variable holding the old database row on UPDATE/DELETE operations on ROW level triggers.

TG_NAME

Datatype name; variable that contains the name of the trigger actually fired.

TG_WHEN

Datatype text; a string of either 'BEFORE' or 'AFTER' depending on the triggers definition.

TG_LEVEL

Datatype text; a string of either 'ROW' or 'STATEMENT' depending on the triggers definition.

TG_OP

Datatype text; a string of 'INSERT', 'UPDATE' or 'DELETE' telling for which operation the trigger is actually fired.

TG_RELID

Datatype oid; the object ID of the table that caused the trigger invocation.

TG_RELNAME

Datatype name; the name of the table that caused the trigger invocation.

TG_NARGS

Datatype integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.

TG_ARGV[]

Datatype array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0 and can be given as an expression. Invalid indices (< 0 or >= tg_nargs) result in a NULL value.

Second they must return either NULL or a record/row containing exactly the structure of the table the trigger was fired for. Triggers fired AFTER might always return a NULL value with no effect. Triggers fired BEFORE signal the trigger manager to skip the operation for this actual row when returning NULL. Otherwise, the returned record/row replaces the inserted/updated row in the operation. It is possible to replace single values directly in NEW and return that or to build a complete new record/row to return.

49.2.2.8. Exceptions

Postgres does not have a very smart exception handling model. Whenever the parser, planner/optimizer or executor decide that a statement cannot be processed any longer, the whole transaction gets aborted and the system jumps back into the mainloop to get the next query from the client application.

It is possible to hook into the error mechanism to notice that this happens. But currently it's impossible to tell what really caused the abort (input/output conversion error, floating point error, parse error). And it is possible that the database backend is in an inconsistent state at this point so returning to the upper executor or issuing more commands might corrupt the whole database. And even if, at this point the information, that the transaction is aborted, is already sent to the client application, so resuming operation does not make any sense.

Thus, the only thing PL/pgSQL currently does when it encounters an abort during execution of a function or trigger procedure is to write some additional DEBUG level log messages telling in which function and where (line number and type of statement) this happened.

49.2.3. Examples

Here are only a few functions to demonstrate how easy PL/pgSQL functions can be written. For more complex examples the programmer might look at the regression test for PL/pgSQL.

One painful detail of writing functions in PL/pgSQL is the handling of single quotes. The functions source text on CREATE FUNCTION must be a literal string. Single quotes inside of literal strings must be either doubled or quoted with a backslash. We are still looking for an elegant alternative. In the meantime, doubling the single qoutes as in the examples below should be used. Any solution for this in future versions of Postgres will be upward compatible.

49.2.3.1. Some Simple PL/pgSQL Functions

The following two PL/pgSQL functions are identical to their counterparts from the C language function discussion.

```
CREATE FUNCTION add_one (int4) RETURNS int4 AS '
    BEGIN
        RETURN $1 + 1;
    END;
' LANGUAGE 'plpgsql';


CREATE FUNCTION concat_text (text, text) RETURNS text AS '
    BEGIN
        RETURN $1 || $2;
    END;
' LANGUAGE 'plpgsql';
```

49.2.3.2. PL/pgSQL Function on Composite Type

Again it is the PL/pgSQL equivalent to the example from The C functions.

```
CREATE FUNCTION c_overpaid (EMP, int4) RETURNS bool AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

49.2.3.3. PL/pgSQL Trigger Procedure

This trigger ensures, that any time a row is inserted or updated in the table, the current username and time are stamped into the row. And it ensures that an employees name is given and that the salary is a positive value.

```
CREATE TABLE emp (
    empname text,
    salary int4,
    last_date datetime,
    last_user name);

CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname ISNULL THEN
            RAISE EXCEPTION ''empname cannot be NULL value'';
        END IF;
```

```
            IF NEW.salary ISNULL THEN
                RAISE EXCEPTION ''% cannot have NULL salary'',
NEW.empname;
            END IF;

            -- Who works for us when she must pay for?
            IF NEW.salary < 0 THEN
                RAISE EXCEPTION ''% cannot have a negative salary'',
NEW.empname;
            END IF;

            -- Remember who changed the payroll when
            NEW.last_date := ''now'';
            NEW.last_user := getpgusername();
            RETURN NEW;
        END;
    ' LANGUAGE 'plpgsql';

    CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
        FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

49.3. PL/Tcl

PL/Tcl is a loadable procedural language for the Postgres database system that enables the Tcl language to be used to create functions and trigger-procedures.

This package was originally written by Jan Wieck.

49.3.1. Overview

PL/Tcl offers most of the capabilities a function writer has in the C language, except for some restrictions.

The good restriction is, that everything is executed in a safe Tcl-interpreter. In addition to the limited command set of safe Tcl, only a few commands are available to access the database over SPI and to raise messages via elog(). There is no way to access internals of the database backend or gaining OS-level access under the permissions of the Postgres user ID like in C. Thus, any unprivileged database user may be permitted to use this language.

The other, internal given, restriction is, that Tcl procedures cannot be used to create input-/output-functions for new data types.

The shared object for the PL/Tcl call handler is automatically built and installed in the Postgres library directory if the Tcl/Tk support is specified in the configuration step of the installation procedure.

49.3.2. Description

49.3.2.1. Postgres Functions and Tcl Procedure Names

In Postgres, one and the same function name can be used for different functions as long as the number of arguments or their types differ. This would collide with Tcl procedure names. To offer the same flexibility in PL/Tcl, the internal Tcl procedure names contain the object ID of the procedures pg_proc row as part of their name. Thus, different argtype versions of the same Postgres function are different for Tcl too.

49.3.2.2. Defining Functions in PL/Tcl

To create a function in the PL/Tcl language, use the known syntax

```
CREATE FUNCTION funcname (argument-types) RETURNS returntype AS '
    # PL/Tcl function body
' LANGUAGE 'pltcl';
```

When calling this function in a query, the arguments are given as variables $1 ... $n to the Tcl procedure body. So a little max function returning the higher of two int4 values would be created as:

```
CREATE FUNCTION tcl_max (int4, int4) RETURNS int4 AS '
    if {$1 > $2} {return $1}
return $2
' LANGUAGE 'pltcl';
```

Composite type arguments are given to the procedure as Tcl arrays. The element names in the array are the attribute names of the composite type. If an attribute in the actual row has the NULL value, it will not appear in the array! Here is an example that defines the overpaid_2 function (as found in the older Postgres documentation) in PL/Tcl

```
CREATE FUNCTION overpaid_2 (EMP) RETURNS bool AS '
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
' LANGUAGE 'pltcl';
```

49.3.2.3. Global Data in PL/Tcl

Sometimes (especially when using the SPI functions described later) it is useful to have some global status data that is held between two calls to a procedure. All PL/Tcl procedures executed in one backend share the same safe Tcl interpreter. To help protecting PL/Tcl procedures from side effects, an array is made available to each procedure via the upvar command. The global name of this variable is the procedures internal name and the local name is GD.

49.3.2.4. Trigger Procedures in PL/Tcl

Trigger procedures are defined in Postgres as functions without arguments and a return type of opaque. And so are they in the PL/Tcl language.

The informations from the trigger manager are given to the procedure body in the following variables:

$TG_name

The name of the trigger from the CREATE TRIGGER statement.

$TG_relid

The object ID of the table that caused the trigger procedure to be invoked.

$TG_relatts

A Tcl list of the tables field names prefixed with an empty list element. So looking up an element name in the list with the lsearch Tcl command returns the same positive number starting from 1 as the fields are numbered in the pg_attribute system catalog.

$TG_when

The string BEFORE or AFTER depending on the event of the trigger call.

$TG_level

The string ROW or STATEMENT depending on the event of the trigger call.

$TG_op

The string INSERT, UPDATE or DELETE depending on the event of the trigger call.

$NEW

An array containing the values of the new table row on INSERT/UPDATE actions, or empty on DELETE.

$OLD

An array containing the values of the old table row on UPDATE/DELETE actions, or empty on INSERT.

$GD

The global status data array as described above.

$args

A Tcl list of the arguments to the procedure as given in the CREATE TRIGGER statement. The arguments are also accessible as $1 ... $n in the procedure body.

The return value from a trigger procedure is one of the strings OK or SKIP, or a list as returned by the 'array get' Tcl command. If the return value is OK, the normal operation (INSERT/UPDATE/DELETE) that fired this trigger will take place. Obviously, SKIP tells the trigger manager to silently suppress the operation. The list from 'array get' tells PL/Tcl to return a modified row to the trigger manager that will be inserted instead of the one given in $NEW (INSERT/UPDATE only). Needless to say that all this is only meaningful when the trigger is BEFORE and FOR EACH ROW.

Here's a little example trigger procedure that forces an integer value in a table to keep track of the # of updates that are performed on the row. For new row's inserted, the value is initialized to 0 and then incremented on every update operation:

```
CREATE FUNCTION trigfunc_modcount() RETURNS OPAQUE AS '
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
```

```
            default {
                return OK
            }
        }
        return [array get NEW]
    ' LANGUAGE 'pltcl';

    CREATE TABLE mytab (num int4, modcnt int4, desc text);

    CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON
mytab
        FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

49.3.2.5. Database Access from PL/Tcl

The following commands are available to access the database from the body of a PL/Tcl procedure:

elog *level msg*

Fire a log message. Possible levels are NOTICE, WARN, ERROR, FATAL, DEBUG and NOIND like for the elog() C function.

quote *string*

Duplicates all occurences of single quote and backslash characters. It should be used when variables are used in the query string given to spi_exec or spi_prepare (not for the value list on spi_execp). Think about a query string like

```
"SELECT '$val' AS ret"
```

where the Tcl variable val actually contains "doesn't". This would result in the final query string

```
"SELECT 'doesn't' AS ret"
```

what would cause a parse error during spi_exec or spi_prepare. It should contain

```
"SELECT 'doesn''t' AS ret"
```

and has to be written as

```
"SELECT '[ quote $val ]' AS ret"
```

spi_exec ?-count *n*? ?-array *name*? *query* ?*loop-body*?

Call parser/planner/optimizer/executor for query. The optional -count value tells spi_exec the maximum number of rows to be processed by the query.

If the query is a SELECT statement and the optional loop-body (a body of Tcl commands like in a foreach statement) is given, it is evaluated for each row selected and behaves like expected on continue/break. The values of selected fields are put into variables named as the column names. So a

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

will set the variable $cnt to the number of rows in the pg_proc system catalog. If the option -array is given, the column values are stored in the associative array named 'name' indexed by the column name instead of individual variables.

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

will print a DEBUG log message for every row of pg_class. The return value of spi_exec is the number of rows affected by query as found in the global variable SPI_processed.

spi_prepare *query typelist*

Prepares AND SAVES a query plan for later execution. It is a bit different from the C level SPI_prepare in that the plan is automatically copied to the toplevel memory context. Thus, there is currently no way of preparing a plan without saving it.

If the query references arguments, the type names must be given as a Tcl list. The return value from spi_prepare is a query ID to be used in subsequent calls to spi_execp. See spi_execp for a sample.

spi_exec ?-count *n*? ?-array *name*? ?-nulls *str*? *query* ?*valuelist*? ?*loop-body*?

Execute a prepared plan from spi_prepare with variable substitution. The optional -count value tells spi_execp the maximum number of rows to be processed by the query.

The optional value for -nulls is a string of spaces and 'n' characters telling spi_execp which of the values are NULL's. If given, it must have exactly the length of the number of values.

The queryid is the ID returned by the spi_prepare call.

If there was a typelist given to spi_prepare, a Tcl list of values of exactly the same length must be given to spi_execp after the query. If the type list on spi_prepare was empty, this argument must be omitted.

If the query is a SELECT statement, the same as described for spi_exec happens for the loop-body and the variables for the fields selected.

Here's an example for a PL/Tcl function using a prepared plan:

```
CREATE FUNCTION t1_count(int4, int4) RETURNS int4 AS '
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \\
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \\
$1 AND num <= \\$2" \\
                int4 ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
' LANGUAGE 'pltcl';
```

Note that each backslash that Tcl should see must be doubled in the query creating the function, since the main parser processes backslashes too on CREATE FUNCTION. Inside the query string given to

spi_prepare should really be dollar signs to mark the parameter positions and to not let $1 be substituted by the value given in the first function call.

Modules and the unknown command

PL/Tcl has a special support for things often used. It recognizes two magic tables, pltcl_modules and pltcl_modfuncs. If these exist, the module 'unknown' is loaded into the interpreter right after creation. Whenever an unknown Tcl procedure is called, the unknown proc is asked to check if the procedure is defined in one of the modules. If this is true, the module is loaded on demand. To enable this behavior, the PL/Tcl call handler must be compiled with -DPLTCL_UNKNOWN_SUPPORT set.

There are support scripts to maintain these tables in the modules subdirectory of the PL/Tcl source including the source for the unknown module that must get installed initially.

Part V. Interfaces

User and programmer interfaces.

Table of Contents

Chapter 50. Functions

Reference information for user-callable functions.

Note

This section needs to be written. Volunteers?

Chapter 51. Large Objects

In Postgres, data values are stored in tuples and individual tuples cannot span data pages. Since the size of a data page is 8192 bytes, the upper limit on the size of a data value is relatively low. To support the storage of larger atomic values, Postgres provides a large object interface. This interface provides file oriented access to user data that has been declared to be a large type. This section describes the implementation and the programmatic and query language interfaces to Postgres large object data.

51.1. Historical Note

Originally, Postgres 4.2 supported three standard implementations of large objects: as files external to Postgres, as UNIX files managed by Postgres, and as data stored within the Postgres database. It causes considerable confusion among users. As a result, we only support large objects as data stored within the Postgres database in PostgreSQL. Even though is is slower to access, it provides stricter data integrity. For historical reasons, this storage scheme is referred to as Inversion large objects. (We will use Inversion and large objects interchangeably to mean the same thing in this section.)

51.2. Inversion Large Objects

The Inversion large object implementation breaks large objects up into "chunks" and stores the chunks in tuples in the database. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes.

51.3. Large Object Interfaces

The facilities Postgres provides to access large objects, both in the backend as part of user-defined functions or the front end as part of an application using the interface, are described below. (For users familiar with Postgres 4.2, PostgreSQL has a new set of functions providing a more coherent interface. The interface is the same for dynamically-loaded C functions as well as for XXX LOST TEXT? WHAT SHOULD GO HERE??. The Postgres large object interface is modeled after the UNIX file system interface, with analogues of `open(2)`, `read(2)`, `write(2)`, `lseek(2)`, etc. User functions call these routines to retrieve only the data of interest from a large object. For example, if a large object type called mugshot existed that stored photographs of faces, then a function called beard could be declared on mugshot data. Beard could look at the lower third of a photograph, and determine the color of the beard that appeared there, if any. The entire large object value need not be buffered, or even examined, by the beard function. Large objects may be accessed from dynamically-loaded C functions or database client programs that link the library. Postgres provides a set of routines that support opening, reading, writing, closing, and seeking on large objects.

51.3.1. Creating a Large Object

The routine

```
Oid lo_creat(PGconn *conn, int mode)
```

creates a new large object. The mode is a bitmask describing several different attributes of the new object. The symbolic constants listed here are defined in `PGROOT/src/backend/libpq/libpq-fs.h` The access type (read, write, or both) is controlled by OR ing together the bits INV_READ and INV_WRITE. If the large object should be archived -- that is, if historical versions of it should be moved periodically to a special archive relation -- then the INV_ARCHIVE bit should be set. The low-order sixteen bits of mask

are the storage manager number on which the large object should reside. For sites other than Berkeley, these bits should always be zero. The commands below create an (Inversion) large object:

```
inv_oid = lo_creat(INV_READ|INV_WRITE|INV_ARCHIVE);
```

51.3.2. Importing a Large Object

To import a UNIX file as a large object, call

```
Oid lo_import(PGconn *conn, text *filename)
```

The filename argument specifies the UNIX pathname of the file to be imported as a large object.

51.3.3. Exporting a Large Object

To export a large object into UNIX file, call

```
int lo_export(PGconn *conn, Oid lobjId, text *filename)
```

The lobjId argument specifies the Oid of the large object to export and the filename argument specifies the UNIX pathname of the file.

51.3.4. Opening an Existing Large Object

To open an existing large object, call

```
int lo_open(PGconn *conn, Oid lobjId, int mode, ...)
```

The lobjId argument specifies the Oid of the large object to open. The mode bits control whether the object is opened for reading INV_READ), writing or both. A large object cannot be opened before it is created. lo_open returns a large object descriptor for later use in lo_read, lo_write, lo_lseek, lo_tell, and lo_close.

51.3.5. Writing Data to a Large Object

The routine

```
int lo_write(PGconn *conn, int fd, char *buf, int len)
```

writes len bytes from buf to large object fd. The fd argument must have been returned by a previous lo_open. The number of bytes actually written is returned. In the event of an error, the return value is negative.

51.3.6. Seeking on a Large Object

To change the current read or write location on a large object, call

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence)
```

This routine moves the current location pointer for the large object described by fd to the new location specified by offset. The valid values for .i whence are SEEK_SET SEEK_CUR and SEEK_END.

51.3.7. Closing a Large Object Descriptor

A large object may be closed by calling

```
int lo_close(PGconn *conn, int fd)
```

where fd is a large object descriptor returned by lo_open. On success, lo_close returns zero. On error, the return value is negative.

51.4. Built in registered functions

There are two built-in registered functions, lo_import and lo_export which are convenient for use in SQL queries. Here is an example of their use

```
CREATE TABLE image (
    name            text,
    raster          oid
);

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

SELECT lo_export(image.raster, "/tmp/motd") from image
    WHERE name = 'beautiful image';
```

51.5. Accessing Large Objects from LIBPQ

Below is a sample program which shows how the large object interface in LIBPQ can be used. Parts of the program are commented out but are left in the source for the readers benefit. This program can be found in `../src/test/examples` Frontend applications which use the large object interface in LIBPQ should include the header file libpq/libpq-fs.h and link with the libpq library.

51.6. Sample Program

```
/*--------------------------------------------------------------
         *
         * testlo.c--
         *    test using large objects with libpq
         *
         * Copyright (c) 1994, Regents of the University of
 California
         *
         *
         * IDENTIFICATION
         *    /usr/local/devel/pglite/cvs/src/doc/manual.me,v 1.16
 1995/09/01 23:55:00 jolly Exp
         *

 *--------------------------------------------------------------
         */
        #include <stdio.h>
        #include "libpq-fe.h"
        #include "libpq/libpq-fs.h"

        #define BUFSIZE         1024

        /*
```

```
          * importFile *    import file "in_filename" into database as
large object "lobjOid"
          *
          */
         Oid importFile(PGconn *conn, char *filename)
         {
             Oid lobjId;
             int lobj_fd;
             char buf[BUFSIZE];
             int nbytes, tmp;
             int fd;

             /*
              * open the file to be read in
              */
             fd = open(filename, O_RDONLY, 0666);
             if (fd < 0)  {   /* error */
              fprintf(stderr, "can't open unix file %s\n", filename);
             }

             /*
              * create the large object
              */
             lobjId = lo_creat(conn, INV_READ|INV_WRITE);
             if (lobjId == 0) {
              fprintf(stderr, "can't create large object\n");
             }

             lobj_fd = lo_open(conn, lobjId, INV_WRITE);
             /*
              * read in from the Unix file and write to the inversion
file
              */
             while ((nbytes = read(fd, buf, BUFSIZE)) > 0) {
              tmp = lo_write(conn, lobj_fd, buf, nbytes);
              if (tmp < nbytes) {
                  fprintf(stderr, "error while reading large object
\n");
              }
             }

             (void) close(fd);
             (void) lo_close(conn, lobj_fd);

             return lobjId;
         }

         void pickout(PGconn *conn, Oid lobjId, int start, int len)
         {
             int lobj_fd;
             char* buf;
             int nbytes;
             int nread;
```

```
            lobj_fd = lo_open(conn, lobjId, INV_READ);
            if (lobj_fd < 0) {
             fprintf(stderr,"can't open large object %d\n",
                  lobjId);
            }

            lo_lseek(conn, lobj_fd, start, SEEK_SET);
            buf = malloc(len+1);

            nread = 0;
            while (len - nread > 0) {
             nbytes = lo_read(conn, lobj_fd, buf, len - nread);
             buf[nbytes] = ' ';
             fprintf(stderr,">>> %s", buf);
             nread += nbytes;
            }
            fprintf(stderr,"\n");
            lo_close(conn, lobj_fd);
        }

        void overwrite(PGconn *conn, Oid lobjId, int start, int len)
        {
            int lobj_fd;
            char* buf;
            int nbytes;
            int nwritten;
            int i;

            lobj_fd = lo_open(conn, lobjId, INV_READ);
            if (lobj_fd < 0) {
             fprintf(stderr,"can't open large object %d\n",
                  lobjId);
            }

            lo_lseek(conn, lobj_fd, start, SEEK_SET);
            buf = malloc(len+1);

            for (i=0;i<len;i++)
             buf[i] = 'X';
            buf[i] = ' ';

            nwritten = 0;
            while (len - nwritten > 0) {
             nbytes = lo_write(conn, lobj_fd, buf + nwritten, len -
nwritten);
             nwritten += nbytes;
            }
            fprintf(stderr,"\n");
            lo_close(conn, lobj_fd);
        }

        /*
         * exportFile *    export large object "lobjOid" to file
"out_filename"
```

```
 *
 */
void exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
    int fd;

    /*
     * create an inversion "object"
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
     fprintf(stderr,"can't open large object %d\n",
          lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT|O_WRONLY, 0666);
    if (fd < 0)  {   /* error */
     fprintf(stderr, "can't open unix file %s\n",
          filename);
    }

    /*
     * read in from the Unix file and write to the inversion
file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) >
0) {
     tmp = write(fd, buf, nbytes);
        if (tmp < nbytes) {
         fprintf(stderr,"error while writing %s\n",
              filename);
     }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn* conn)
{
  PQfinish(conn);
  exit(1);
}

int
```

```
        main(int argc, char **argv)
        {
            char *in_filename, *out_filename;
            char *database;
            Oid lobjOid;
            PGconn *conn;
            PGresult *res;

            if (argc != 4) {
             fprintf(stderr, "Usage: %s database_name in_filename
 out_filename\n",
                  argv[0]);
             exit(1);
            }

            database = argv[1];
            in_filename = argv[2];
            out_filename = argv[3];

            /*
             * set up the connection
             */
            conn = PQsetdb(NULL, NULL, NULL, NULL, database);

            /* check to see that the backend connection was
 successfully made */
            if (PQstatus(conn) == CONNECTION_BAD) {
             fprintf(stderr,"Connection to database '%s' failed.\n",
 database);
             fprintf(stderr,"%s",PQerrorMessage(conn));
             exit_nicely(conn);
            }

            res = PQexec(conn, "begin");
            PQclear(res);

            printf("importing file %s\n", in_filename);
        /*  lobjOid = importFile(conn, in_filename); */
            lobjOid = lo_import(conn, in_filename);
        /*
            printf("as large object %d.\n", lobjOid);

            printf("picking out bytes 1000-2000 of the large object
\n");
            pickout(conn, lobjOid, 1000, 1000);

            printf("overwriting bytes 1000-2000 of the large object
 with X's\n");
            overwrite(conn, lobjOid, 1000, 1000);
        */

            printf("exporting large object to file %s\n",
 out_filename);
        /*    exportFile(conn, lobjOid, out_filename); */
```

```
    lo_export(conn, lobjOid,out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Chapter 52. ecpg - Embedded SQL in C

Linux Tolke
Michael Meskes

This describes an embedded SQL in C package for Postgres. It is written by Linus Tolke[1] and Michael Meskes[2].

Note

Permission is granted to copy and use in the same way as you are allowed to copy and use the rest of the PostgreSQL.

52.1. Why Embedded SQL?

Embedded SQL has some small advantages over other ways to handle SQL queries. It takes care of all the tedious moving of information to and from variables in your C program. Many RDBMS packages support this embedded language.

There is an ANSI-standard describing how the embedded language should work. ecpg was designed to meet this standard as much as possible. So it is possible to port programs with embedded SQL written for other RDBMS packages to Postgres and thus promoting the spirit of free software.

52.2. The Concept

You write your program in C with some special SQL things. For declaring variables that can be used in SQL statements you need to put them in a special declare section. You use a special syntax for the SQL queries.

Before compiling you run the file through the embedded SQL C preprocessor and it converts the SQL statements you used to function calls with the variables used as arguments. Both variables that are used as input to the SQL statements and variables that will contain the result are passed.

Then you compile and at link time you link with a special library that contains the functions used. These functions (actually it is mostly one single function) fetches the information from the arguments, performs the SQL query using the ordinary interface (`libpq`) and puts back the result in the arguments dedicated for output.

Then you run your program and when the control arrives to the SQL statement the SQL statement is performed against the database and you can continue with the result.

52.3. How To Use egpc

This section describes how to use the egpc tool.

52.3.1. Preprocessor

The preprocessor is called ecpg. After installation it resides in the Postgres `bin/` directory.

[1] mailto:linus@epact.se
[2] mailto:meskes@debian.org

52.3.2. Library

The ecpg library is called `libecpg.a` or `libecpg.so`. Additionally, the library uses the `libpq` library for communication to the Postgres server so you will have to link your program with *`-lecpg -lpq`*.

The library has some methods that are "hidden" but that could prove very useful sometime.

- `ECPGdebug(int` *`on`*`, FILE *`*`stream`*`)` turns on debug logging if called with the first argument non-zero. Debug logging is done on *`stream`*. Most SQL statement logs its arguments and result.

  The most important one (`ECPGdo`) that is called on all SQL statements except `EXEC SQL COMMIT`, `EXEC SQL ROLLBACK`, `EXEC SQL CONNECT` logs both its expanded string, i.e. the string with all the input variables inserted, and the result from the Postgres server. This can be very useful when searching for errors in your SQL statements.

- `ECPGstatus()` This method returns TRUE if we are connected to a database and FALSE if not.

52.3.3. Error handling

To be able to detect errors from the Postgres server you include a line like

```
exec sql include sqlca;
```

in the include section of your file. This will define a struct and a variable with the name *`sqlca`* as following:

```
struct sqlca {
    int sqlcode;
    struct {
        int sqlerrml;
        char sqlerrmc[1000];
    } sqlerrm;
} sqlca;
```

If an error occured in the last SQL statement then *`sqlca.sqlcode`* will be non-zero. If *`sqlca.sqlcode`* is less that 0 then this is some kind of serious error, like the database definition does not match the query given. If it is bigger than 0 then this is a normal error like the table did not contain the requested row.

sqlca.sqlerrm.sqlerrmc will contain a string that describes the error. The string ends with “line 23.” where the line is the line number in the source file (actually the file generated by the preprocessor but I hope I can fix this to be the line number in the input file.)

List of errors that can occur:

-1, Unsupported type %s on line %d.

> Does not normally occur. This is a sign that the preprocessor has generated something that the library does not know about. Perhaps you are running incompatible versions of the preprocessor and the library.

-1, Too many arguments line %d.

> The preprocessor has goofed up and generated some incorrect code.

-1, Too few arguments line %d.

The preprocessor has goofed up and generated some incorrect code.

-1, Error starting transaction line %d.

Postgres signalled to us that we cannot open the connection.

-1, Postgres error: %s line %d.

Some Postgres error. The message contains the error message from the Postgres backend.

1, Data not found line %d.

This is a "normal" error that tells you that what you are quering cannot be found or we have gone through the cursor.

-1, To many matches line %d.

This means that the query has returned several lines. The `SELECT` you made probably was not unique.

-1, Not correctly formatted int type: %s line %d.

This means that the host variable is of an `int` type and the field in the Postgres database is of another type and contains a value that cannot be interpreted as an `int`. The library uses `strtol` for this conversion.

-1, Not correctly formatted unsigned type: %s line %d.

This means that the host variable is of an `unsigned int` type and the field in the Postgres database is of another type and contains a value that cannot be interpreted as an `unsigned int`. The library uses `strtoul` for this conversion.

-1, Not correctly formatted floating point type: %s line %d.

This means that the host variable is of an `float` type and the field in the Postgres database is of another type and contains a value that cannot be interpreted as an `float`. The library uses `strtod` for this conversion.

-1, Too few arguments line %d.

This means that Postgres has returned more records than we have matching variables. Perhaps you have forgotten a couple of the host variables in the `INTO :var1,:var2`-list.

-1, Too many arguments line %d.

This means that Postgres has returned fewer records than we have host variables. Perhaps you have to many host variables in the `INTO :var1,:var2`-list.

-1, Empty query line %d.

Postgres returned PGRES_EMPTY_QUERY.

-1, Error: %s line %d.

This means that Postgres returned on of the errors PGRES_NONFATAL_ERROR, PGRES_FATAL_ERROR or PGRES_BAD_RESPONSE. Which one and why is explained in the message.

-1, Postgres error line %d.

Postgres returns something that the library does not know how to handle. This is probably because the version of Postgres does not match the version of the ecpg library.

-1, Error committing line %d.

Error during `COMMIT`. `EXEC SQL COMMIT` is translated to an `end` operation in Postgres and that is the operation that could not be performed.

-1, Error rolling back line %d.

Error during `ROLLBACK`. `EXEC SQL ROLLBACK` is translated to an `abort` operation in Postgres and that is the operation that could not be performed.

-1, ECPGconnect: could not open database %s.

The connect to the database did not work.

52.4. Limitations

What will never be included and why or what cannot be done with this concept.

oracles single tasking possibility

Oracle version 7.0 on AIX 3 uses the OS-supported locks on the shared memory segments and allows the application designer to link an application in a so called single tasking way. Instead of starting one client process per application process both the database part and the application part is run in the same process. In later versions of oracle this is no longer supported.

This would require a total redesign of the Postgres access model and that effort can not justify the performance gained.

52.5. Porting From Other RDBMS Packages

To be written by someone who knows the different RDBMS packages and who actually does port something...

52.6. Installation

Since version 0.5 ecpg is distributed together with Postgres. So you should get your precompiler, libraries and header files compiled and installed by default as a part of your installation.

52.7. For the Developer

This section is for those who want to develop the ecpg interface. It describes how the things work. The ambition is to make this section contain things for those that want to have a look inside and the section on How to use it should be enough for all normal questions. So, read this before looking at the internals of the ecpg. If you are not interested in how it really works, skip this section.

52.7.1. ToDo List

This version the preprocessor has some flaws:

no restriction to strings only

The PQ interface, and most of all the PQexec function, that is used by the ecpg relies on that the request is built up as a string. In some cases, like when the data contains the null character, this will be a serious problem.

error codes

There should be different error numbers for the different errors instead of just -1 for them all.

library functions

to_date et al.

records

Records or structures have to be defined in the declare section.

missing statements

The following statements are not implemented thus far:

exec sql type

exec sql prepare

exec sql allocate

exec sql free

exec sql whenever sqlwarning

SQLSTATE

message ´no data found´

The error message for "no data" in an exec sql insert select from statement has to be 100.

sqlwanr[6]

sqlwarn[6] should be 'W' if the PRECISION or SCALE value specified in a SET DESCRIPTOR statement will be ignored.

conversion of scripts

To set up a database you need a few scripts with table definitions and other configuration parameters. If you have these scripts for an old database you would like to just apply them to get a Postgres database that works in the same way.

To set up a database you need a few scripts with table definitions and The functionality could be accomplished with some conversion scripts. Speed will never be accomplished in this way. To do this you need a bigger insight in the database construction and the use of the database than could be realised in a script.

52.7.2. The Preprocessor

First four lines are written to the output. Two comments and two include lines necessary for the interface to the library.

Then the preprocessor works in one pass only reading the input file and writing to the output as it goes along. Normally it just echoes everything to the output without looking at it further.

When it comes to an `EXEC SQL` statements it interviens and changes them depending on what iit is. The `EXEC SQL` statement can be one of these:

Declare sections

Declare sections begins with

```
exec sql begin declare section;
```

and ends with

```
exec sql end declare section;
```

In the section only variable declarations are allowed. Every variable declare within this section is also entered in a list of variables indexed on their name together with the corresponding type.

The declaration is echoed to the file to make the variable a normal C-variable also.

The special types VARCHAR and VARCHAR2 are converted into a named struct for every variable. A declaration like:

```
VARCHAR var[180];
```

is converted into

```
struct varchar_var { int len; char arr[180]; } var;
```

Include statements

An include statement looks like:

```
exec sql include filename;
```

It is converted into

```
#include <filename.h>
```

Connect statement

A connect statement looks like:

```
exec sql connect 'database';
```

That statement is converted into

```
ECPGconnect("database");
```

Open cursor statement

An open cursor statement looks like:

```
exec sql open cursor;
```

and is ignore and not copied from the output.

Commit statement

A commit statement looks like

```
exec sql commit;
```

and is translated on the output to

```
ECPGcommit(__LINE__);
```

Rollback statement

A rollback statement looks like

```
exec sql rollback;
```

and is translated on the output to

```
ECPGrollback(__LINE__);
```

Other statements

Other SQL statements are other statements that start with `exec sql` and ends with `;`. Everything inbetween is treated as an SQL statement and parsed for variable substitution.

Variable substitution occur when a symbol starts with a colon (`:`). Then a variable with that name is found among the variables that were previously declared within a declare section and depending on whether or not the SQL statements knows it to be a variable for input or output the pointers to the variables are written to the output to allow for access by the function.

For every variable that is part of the SQL request the function gets another five arguments:

The type as a special symbol
A pointer to the value
The size of the variable if it is a varchar
Number of elements in the array (for array fetches)
The offset to the next element in the array (for array fetches)

Since the array fetches are not implemented yet the two last arguments are not really important. They could perhaps have been left out.

52.7.3. A Complete Example

Here is a complete example describing the output of the preprocessor:

```
exec sql begin declare section;
int index;
int result;
exec sql end declare section;
...
    exec sql select res into :result from mytable where index
 = :index;
```

is translated into:

```
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>
#include <ecpglib.h>
/* exec sql begin declare section */

 int index;
 int result;
/* exec sql end declare section */

...
    ECPGdo(__LINE__, "select res from mytable where index = ;;",
           ECPGt_int,&index,0,0,sizeof(int),
           ECPGt_EOIT,
           ECPGt_int,&result,0,0,sizeof(int),
           ECPGt_EORT );
```

(the indentation in this manual is added for readability and not something that the preprocessor can do.)

52.7.4. The Library

The most important function in the library is the `ECPGdo` function. It takes a variable amount of arguments. Hopefully we wont run into machines with limits on the amount of variables that can be accepted by a varchar function. This could easily add up to 50 or so arguments.

The arguments are:

A line number

This is a line number for the original line used in error messages only.

A string

This is the SQL request that is to be issued. This request is modified by the input variables, i.e. the variables that where not known at compile time but are to be entered in the request. Where the variables should go the string contains “;”.

Input variables

As described in the section about the preprocessor every input variable gets five arguments.

ECPGt_EOIT

An enum telling that there are no more input variables.

Output variables

As described in the section about the preprocessor every input variable gets five arguments. These variables are filled by the function.

ECPGt_EORT

An enum telling that there are no more variables.

All the SQL statements are performed in one transaction unless you issue a commit transaction. This works so that the first transaction or the first after a commit or rollback always begins a transaction.

To be completed: entries describing the other entries.

Chapter 53. libpq

libpq is the C application programmer's interface to Postgres. libpq is a set of library routines that allow client programs to pass queries to the Postgres backend server and to receive the results of these queries. libpq is also the underlying engine for several other Postgres application interfaces, including libpq++ (C++), libpgtcl (Tcl), perl5, and ecpg. So some aspects of libpq's behavior will be important to you if you use one of those packages. Three short programs are included at the end of this section to show how to write programs that use libpq. There are several complete examples of libpq applications in the following directories:

```
../src/test/regress
../src/test/examples
../src/bin/psql
```

Frontend programs which use libpq must include the header file libpq-fe.h and must link with the libpq library.

53.1. Database Connection Functions

The following routines deal with making a connection to a Postgres backend server. The application program can have several backend connections open at one time. (One reason to do that is to access more than one database.) Each connection is represented by a PGconn object which is obtained from PQconnectdb() or PQsetdbLogin(). NOTE that these functions will always return a non-null object pointer, unless perhaps there is too little memory even to allocate the PGconn object. The PQstatus function should be called to check whether a connection was successfully made before queries are sent via the connection object.

- PQsetdbLogin Makes a new connection to a backend.

  ```
  PGconn *PQsetdbLogin(const char *pghost,
                  const char *pgport,
                  const char *pgoptions,
                  const char *pgtty,
                  const char *dbName,
                  const char *login,
                  const char *pwd)
  ```

  If any argument is NULL, then the corresponding environment variable (see "Environment Variables" section) is checked. If the environment variable is also not set, then hardwired defaults are used. The return value is a pointer to an abstract struct representing the connection to the backend.

- PQsetdb Makes a new connection to a backend.

  ```
  PGconn *PQsetdb(char *pghost,
                  char *pgport,
                  char *pgoptions,
                  char *pgtty,
                  char *dbName)
  ```

  This is a macro that calls PQsetdbLogin() with null pointers for the login and pwd parameters. It is provided primarily for backward compatibility with old programs.

- PQconnectdb Makes a new connection to a backend.

```
PGconn *PQconnectdb(const char *conninfo)
```

This routine opens a new database connection using parameters taken from a string. Unlike PQsetdbLogin(), the parameter set can be extended without changing the function signature, so use of this routine is encouraged for new application programming. The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace. Each parameter setting is in the form keyword = value. (To write a null value or a value containing spaces, surround it with single quotes, eg, keyword = 'a value'. Single quotes within the value must be written as \'. Spaces around the equal sign are optional.) The currently recognized parameter keywords are:

- host -- host to connect to. If a non-zero-length string is specified, TCP/IP communication is used. Without a host name, libpq will connect using a local Unix domain socket.
- port -- port number to connect to at the server host, or socket filename extension for Unix-domain connections.
- dbname -- database name.
- user -- user name for authentication.
- password -- password used if the backend demands password authentication.
- authtype -- authorization type. (No longer used, since the backend now chooses how to authenticate users. libpq still accepts and ignores this keyword for backward compatibility.)
- options -- trace/debug options to send to backend.
- tty -- file or tty for optional debug output from backend.

Like PQsetdbLogin, PQconnectdb uses environment variables or built-in default values for unspecified options.

- `PQconndefaults` Returns the default connection options.

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
        {
                char   *keyword;   /* The keyword of the option */
                char   *envvar;    /* Fallback environment variable
 name */
                char   *compiled;  /* Fallback compiled in default
 value */
                char   *val;       /* Option's value */
                char   *label;     /* Label for field in connect
 dialog */
                char   *dispchar;  /* Character to display for this
 field
                                      in a connect dialog. Values
 are:
                                      ""        Display entered
 value as is
                                      "*"       Password field -
 hide value
                                      "D"       Debug options -
 don't
```

```
                                        create a field by default */
                int     dispsize;  /* Field size in characters for
 dialog */
        };
```

Returns the address of the connection options structure. This may be used to determine all possible PQconnectdb options and their current default values. The return value points to an array of PQconninfoOption structs, which ends with an entry having a NULL keyword pointer. Note that the default values ("val" fields) will depend on environment variables and other context. Callers must treat the connection options data as read-only.

- `PQfinish` Close the connection to the backend. Also frees memory used by the PGconn object.

  ```
  void PQfinish(PGconn *conn)
  ```

  Note that even if the backend connection attempt fails (as indicated by PQstatus), the application should call PQfinish to free the memory used by the PGconn object. The PGconn pointer should not be used after PQfinish has been called.

- `PQreset` Reset the communication port with the backend.

  ```
  void PQreset(PGconn *conn)
  ```

  This function will close the connection to the backend and attempt to reestablish a new connection to the same postmaster, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost.

`libpq` application programmers should be careful to maintain the PGconn abstraction. Use the accessor functions below to get at the contents of PGconn. Avoid directly referencing the fields of the PGconn structure because they are subject to change in the future. (Beginning in Postgres release 6.4, the definition of struct PGconn is not even provided in libpq-fe.h. If you have old code that accesses PGconn fields directly, you can keep using it by including libpq-int.h too, but you are encouraged to fix the code soon.)

- `PQdb` Returns the database name of the connection.

  ```
  char *PQdb(PGconn *conn)
  ```

  PQdb and the next several functions return the values established at connection. These values are fixed for the life of the PGconn object.

- `PQuser` Returns the user name of the connection.

  ```
  char *PQuser(PGconn *conn)
  ```

- `PQpass` Returns the password of the connection.

  ```
  char *PQpass(PGconn *conn)
  ```

- `PQhost` Returns the server host name of the connection.

  ```
  char *PQhost(PGconn *conn)
  ```

- `PQport` Returns the port of the connection.

  ```
  char *PQport(PGconn *conn)
  ```

- `PQtty` Returns the debug tty of the connection.

  ```
  char *PQtty(PGconn *conn)
  ```

- `PQoptions` Returns the backend options used in the connection.

  ```
  char *PQoptions(PGconn *conn)
  ```

- `PQstatus` Returns the status of the connection. The status can be CONNECTION_OK or CONNECTION_BAD.

  ```
  ConnStatusType *PQstatus(PGconn *conn)
  ```

  A failed connection attempt is signaled by status CONNECTION_BAD. Ordinarily, an OK status will remain so until PQfinish, but a communications failure might result in the status changing to CONNECTION_BAD prematurely. In that case the application could try to recover by calling PQreset.

- `PQerrorMessage` Returns the error message most recently generated by an operation on the connection.

  ```
  char *PQerrorMessage(PGconn* conn);
  ```

  Nearly all libpq functions will set PQerrorMessage if they fail. Note that by libpq convention, a non-empty PQerrorMessage will include a trailing newline.

- `PQbackendPID` Returns the process ID of the backend server handling this connection.

  ```
  int PQbackendPID(PGconn *conn);
  ```

  The backend PID is useful for debugging purposes and for comparison to NOTIFY messages (which include the PID of the notifying backend). Note that the PID belongs to a process executing on the database server host, not the local host!

53.2. Query Execution Functions

Once a connection to a database server has been successfully established, the functions described here are used to perform SQL queries and commands.

- `PQexec` Submit a query to Postgres and wait for the result.

  ```
  PGresult *PQexec(PGconn *conn,
                   const char *query);
  ```

  Returns a PGresult pointer or possibly a NULL pointer. A non-NULL pointer will generally be returned except in out-of-memory conditions or serious errors such as inability to send the query to the backend. If a NULL is returned, it should be treated like a PGRES_FATAL_ERROR result. Use PQerrorMessage to get more information about the error.

The `PGresult` structure encapsulates the query result returned by the backend. `libpq` application programmers should be careful to maintain the PGresult abstraction. Use the accessor functions below to get at the contents of PGresult. Avoid directly referencing the fields of the PGresult structure because they are subject to change in the future. (Beginning in Postgres release 6.4, the definition of struct PGresult is not even provided in libpq-fe.h. If you have old code that accesses PGresult fields directly, you can keep using it by including libpq-int.h too, but you are encouraged to fix the code soon.)

- `PQresultStatus` Returns the result status of the query. PQresultStatus can return one of the following values:

  ```
  PGRES_EMPTY_QUERY,
  ```

```
PGRES_COMMAND_OK,       /* the query was a command returning no data
 */
PGRES_TUPLES_OK,        /* the query successfully returned tuples */
PGRES_COPY_OUT,         /* Copy Out (from server) data transfer
 started */
PGRES_COPY_IN,          /* Copy In (to server) data transfer started
 */
PGRES_BAD_RESPONSE,     /* an unexpected response was received */
PGRES_NONFATAL_ERROR,
PGRES_FATAL_ERROR
```

If the result status is PGRES_TUPLES_OK, then the routines described below can be used to retrieve the tuples returned by the query. Note that a SELECT that happens to retrieve zero tuples still shows PGRES_TUPLES_OK. PGRES_COMMAND_OK is for commands that can never return tuples.

- `PQresultErrorMessage` returns the error message associated with the query, or an empty string if there was no error.

  ```
  const char *PQresultErrorMessage(PGresult *res);
  ```

  Immediately following a PQexec or PQgetResult call, PQerrorMessage (on the connection) will return the same string as PQresultErrorMessage (on the result). However, a PGresult will retain its error message until destroyed, whereas the connection's error message will change when subsequent operations are done. Use PQresultErrorMessage when you want to know the status associated with a particular PGresult; use PQerrorMessage when you want to know the status from the latest operation on the connection.

- `PQntuples` Returns the number of tuples (instances) in the query result.

  ```
  int PQntuples(PGresult *res);
  ```

- `PQnfields` Returns the number of fields (attributes) in each tuple of the query result.

  ```
  int PQnfields(PGresult *res);
  ```

- `PQbinaryTuples` Returns 1 if the PGresult contains binary tuple data, 0 if it contains ASCII data.

  ```
  int PQbinaryTuples(PGresult *res);
  ```

  Currently, binary tuple data can only be returned by a query that extracts data from a BINARY cursor.

- `PQfname` Returns the field (attribute) name associated with the given field index. Field indices start at 0.

  ```
  char *PQfname(PGresult *res,
                int field_index);
  ```

- `PQfnumber` Returns the field (attribute) index associated with the given field name.

  ```
  int PQfnumber(PGresult *res,
                char* field_name);
  ```

  -1 is returned if the given name does not match any field.

- `PQftype` Returns the field type associated with the given field index. The integer returned is an internal coding of the type. Field indices start at 0.

  ```
  Oid PQftype(PGresult *res,
  ```

```
            int field_num);
```

- `PQfsize` Returns the size in bytes of the field associated with the given field index. Field indices start at 0.

  ```
  int PQfsize(PGresult *res,
              int field_index);
  ```

  PQfsize returns the space allocated for this field in a database tuple, in other words the size of the server's binary representation of the data type. -1 is returned if the field is variable size.

- `PQfmod` Returns the type-specific modification data of the field associated with the given field index. Field indices start at 0.

  ```
  int PQfmod(PGresult *res,
             int field_index);
  ```

- `PQgetvalue` Returns a single field (attribute) value of one tuple of a PGresult. Tuple and field indices start at 0.

  ```
  char* PQgetvalue(PGresult *res,
                   int tup_num,
                   int field_num);
  ```

  For most queries, the value returned by PQgetvalue is a null-terminated ASCII string representation of the attribute value. But if PQbinaryTuples() is TRUE, the value returned by PQgetvalue is the binary representation of the type in the internal format of the backend server (but not including the size word, if the field is variable-length). It is then the programmer's responsibility to cast and convert the data to the correct C type. The pointer returned by PQgetvalue points to storage that is part of the PGresult structure. One should not modify it, and one must explicitly copy the value into other storage if it is to be used past the lifetime of the PGresult structure itself.

- `PQgetlength` Returns the length of a field (attribute) in bytes. Tuple and field indices start at 0.

  ```
  int PQgetlength(PGresult *res,
                  int tup_num,
                  int field_num);
  ```

  This is the actual data length for the particular data value, that is the size of the object pointed to by PQgetvalue. Note that for ASCII-represented values, this size has little to do with the binary size reported by PQfsize.

- `PQgetisnull` Tests a field for a NULL entry. Tuple and field indices start at 0.

  ```
  int PQgetisnull(PGresult *res,
                  int tup_num,
                  int field_num);
  ```

  This function returns 1 if the field contains a NULL, 0 if it contains a non-null value. (Note that PQgetvalue will return an empty string, not a null pointer, for a NULL field.)

- `PQcmdStatus` Returns the command status string from the SQL command that generated the PGresult.

  ```
  char *PQcmdStatus(PGresult *res);
  ```

- `PQcmdTuples` Returns the number of rows affected by the SQL command.

  ```
  const char *PQcmdTuples(PGresult *res);
  ```

If the SQL command that generated the PGresult was INSERT, UPDATE or DELETE, this returns a string containing the number of rows affected. If the command was anything else, it returns the empty string.

- `PQoidStatus` Returns a string with the object id of the tuple inserted, if the SQL command was an INSERT. Otherwise, returns an empty string.

```
char* PQoidStatus(PGresult *res);
```

- `PQprint` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PQprint(FILE* fout,      /* output stream */
             PGresult* res,
             PQprintOpt* po);

struct _PQprintOpt
        {
                pqbool  header;      /* print output field headings
 and row count */
                pqbool  align;       /* fill align the fields */
                pqbool  standard;    /* old brain dead format */
                pqbool  html3;       /* output html tables */
                pqbool  expanded;    /* expand tables */
                pqbool  pager;       /* use pager for output if
 needed */
                char    *fieldSep;   /* field separator */
                char    *tableOpt;   /* insert to HTML <table ...>
 */
                char    *caption;    /* HTML <caption> */
                char    **fieldName; /* null terminated array of
 replacement field names */
        };
```

This function is intended to replace PQprintTuples(), which is now obsolete. The `psql` program uses PQprint() to display query results.

- `PQprintTuples` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PQprintTuples(PGresult* res,
                   FILE* fout,      /* output stream */
                   int printAttName,/* print attribute names or
 not*/
                   int terseOutput, /* delimiter bars or not?*/
                   int width);      /* width of column, variable
 width if 0*/
```

- `PQdisplayTuples` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PQdisplayTuples(PGresult* res,
                     FILE* fout,           /* output stream */
                     int fillAlign,        /* space fill to align
 columns */
                     const char *fieldSep, /* field separator */
```

```
                        int printHeader,      /* display headers? */
                        int quiet);           /* suppress print of row
 count at end */
```

PQdisplayTuples() was intended to supersede PQprintTuples(), and is in turn superseded by PQprint().

- `PQclear` Frees the storage associated with the PGresult. Every query result should be freed via PQclear when it is no longer needed.

  ```
  void PQclear(PQresult *res);
  ```

  You can keep a PGresult object around for as long as you need it; it does not go away when you issue a new query, nor even if you close the connection. To get rid of it, you must call PQclear. Failure to do this will result in memory leaks in the frontend application.

- `PQmakeEmptyPGresult` Constructs an empty PGresult object with the given status.

  ```
  PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
  ```

  This is libpq's internal routine to allocate and initialize an empty PGresult object. It is exported because some applications find it useful to generate result objects (particularly objects with error status) themselves. If conn is not NULL and status indicates an error, the connection's current errorMessage is copied into the PGresult. Note that PQclear should eventually be called on the object, just as with a PGresult returned by libpq itself.

53.3. Asynchronous Query Processing

The PQexec function is adequate for submitting queries in simple synchronous applications. It has a couple of major deficiencies however:

- PQexec waits for the query to be completed. The application may have other work to do (such as maintaining a user interface), in which case it won't want to block waiting for the response.
- Since control is buried inside PQexec, it is hard for the frontend to decide it would like to try to cancel the ongoing query. (It can be done from a signal handler, but not otherwise.)
- PQexec can return only one PGresult structure. If the submitted query string contains multiple SQL commands, all but the last PGresult are discarded by PQexec.

Applications that do not like these limitations can instead use the underlying functions that PQexec is built from: PQsendQuery and PQgetResult.

- `PQsendQuery` Submit a query to Postgres without waiting for the result(s). TRUE is returned if the query was successfully dispatched, FALSE if not (in which case, use PQerrorMessage to get more information about the failure).

  ```
  int PQsendQuery(PGconn *conn,
                  const char *query);
  ```

  After successfully calling PQsendQuery, call PQgetResult one or more times to obtain the query results. PQsendQuery may not be called again (on the same connection) until PQgetResult has returned NULL, indicating that the query is done.

- `PQgetResult` Wait for the next result from a prior PQsendQuery, and return it. NULL is returned when the query is complete and there will be no more results.

  ```
  PGresult *PQgetResult(PGconn *conn);
  ```

PQgetResult must be called repeatedly until it returns NULL, indicating that the query is done. (If called when no query is active, PQgetResult will just return NULL at once.) Each non-null result from PQgetResult should be processed using the same PGresult accessor functions previously described. Don't forget to free each result object with PQclear when done with it. Note that PQgetResult will block only if a query is active and the necessary response data has not yet been read by PQconsumeInput.

Using PQsendQuery and PQgetResult solves one of PQexec's problems: if a query string contains multiple SQL commands, the results of those commands can be obtained individually. (This allows a simple form of overlapped processing, by the way: the frontend can be handling the results of one query while the backend is still working on later queries in the same query string.) However, calling PQgetResult will still cause the frontend to block until the backend completes the next SQL command. This can be avoided by proper use of three more functions:

- `PQconsumeInput` If input is available from the backend, consume it.

  ```
  int PQconsumeInput(PGconn *conn);
  ```

  PQconsumeInput normally returns 1 indicating "no error", but returns 0 if there was some kind of trouble (in which case PQerrorMessage is set). Note that the result does not say whether any input data was actually collected. After calling PQconsumeInput, the application may check PQisBusy and/or PQnotifies to see if their state has changed. PQconsumeInput may be called even if the application is not prepared to deal with a result or notification just yet. The routine will read available data and save it in a buffer, thereby causing a select(2) read-ready indication to go away. The application can thus use PQconsumeInput to clear the select condition immediately, and then examine the results at leisure.

- `PQisBusy` Returns TRUE if a query is busy, that is, PQgetResult would block waiting for input. A FALSE return indicates that PQgetResult can be called with assurance of not blocking.

  ```
  int PQisBusy(PGconn *conn);
  ```

  PQisBusy will not itself attempt to read data from the backend; therefore PQconsumeInput must be invoked first, or the busy state will never end.

- `PQsocket` Obtain the file descriptor number for the backend connection socket. A valid descriptor will be >= 0; a result of -1 indicates that no backend connection is currently open.

  ```
  int PQsocket(PGconn *conn);
  ```

  PQsocket should be used to obtain the backend socket descriptor in preparation for executing select(2). This allows an application to wait for either backend responses or other conditions. If the result of select(2) indicates that data can be read from the backend socket, then PQconsumeInput should be called to read the data; after which, PQisBusy, PQgetResult, and/or PQnotifies can be used to process the response.

A typical frontend using these functions will have a main loop that uses select(2) to wait for all the conditions that it must respond to. One of the conditions will be input available from the backend, which in select's terms is readable data on the file descriptor identified by PQsocket. When the main loop detects input ready, it should call PQconsumeInput to read the input. It can then call PQisBusy, followed by PQgetResult if PQisBusy returns FALSE. It can also call PQnotifies to detect NOTIFY messages (see "Asynchronous Notification", below). An example is given in the sample programs section.

A frontend that uses PQsendQuery/PQgetResult can also attempt to cancel a query that is still being processed by the backend.

- `PQrequestCancel` Request that Postgres abandon processing of the current query.

```
int PQrequestCancel(PGconn *conn);
```

The return value is TRUE if the cancel request was successfully dispatched, FALSE if not. (If not, PQerrorMessage tells why not.) Successful dispatch is no guarantee that the request will have any effect, however. Regardless of the return value of PQrequestCancel, the application must continue with the normal result-reading sequence using PQgetResult. If the cancellation is effective, the current query will terminate early and return an error result. If the cancellation fails (say because the backend was already done processing the query), then there will be no visible result at all.

Note that if the current query is part of a transaction, cancellation will abort the whole transaction.

PQrequestCancel can safely be invoked from a signal handler. So, it is also possible to use it in conjunction with plain PQexec, if the decision to cancel can be made in a signal handler. For example, psql invokes PQrequestCancel from a SIGINT signal handler, thus allowing interactive cancellation of queries that it issues through PQexec. Note that PQrequestCancel will have no effect if the connection is not currently open or the backend is not currently processing a query.

53.4. Fast Path

Postgres provides a fast path interface to send function calls to the backend. This is a trapdoor into system internals and can be a potential security hole. Most users will not need this feature.

- `PQfn` Request execution of a backend function via the fast path interface.

```
PGresult* PQfn(PGconn* conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               PQArgBlock *args,
               int nargs);
```

The fnid argument is the object identifier of the function to be executed. result_buf is the buffer in which to place the return value. The caller must have allocated sufficient space to store the return value (there is no check!). The actual result length will be returned in the integer pointed to by result_len. If a 4-byte integer result is expected, set result_is_int to 1; otherwise set it to 0. (Setting result_is_int to 1 tells libpq to byte-swap the value if necessary, so that it is delivered as a proper int value for the client machine. When result_is_int is 0, the byte string sent by the backend is returned unmodified.) args and nargs specify the arguments to be passed to the function.

```
typedef struct {
             int len;
             int isint;
             union {
                 int *ptr;
                 int integer;
             } u;
         } PQArgBlock;
```

PQfn always returns a valid PGresult*. The resultStatus should be checked before the result is used. The caller is responsible for freeing the PGresult with PQclear when it is no longer needed.

53.5. Asynchronous Notification

Postgres supports asynchronous notification via the LISTEN and NOTIFY commands. A backend registers its interest in a particular notification condition with the LISTEN command (and can stop listening with the UNLISTEN command). All backends listening on a particular condition will be notified asynchronously when a NOTIFY of that condition name is executed by any backend. No additional information is passed from the notifier to the listener. Thus, typically, any actual data that needs to be communicated is transferred through a database relation. Commonly the condition name is the same as the associated relation, but it is not necessary for there to be any associated relation.

`libpq` applications submit LISTEN and UNLISTEN commands as ordinary SQL queries. Subsequently, arrival of NOTIFY messages can be detected by calling PQnotifies().

- `PQnotifies` Returns the next notification from a list of unhandled notification messages received from the backend. Returns NULL if there are no pending notifications. Once a notification is returned from PQnotifies, it is considered handled and will be removed from the list of notifications.

```
PGnotify* PQnotifies(PGconn *conn);

typedef struct pgNotify
    {
        char        relname[NAMEDATALEN];       /* name of relation
                                                 * containing data
 */
        int         be_pid;                     /* process id of
 backend */
    } PGnotify;
```

  After processing a PGnotify object returned by PQnotifies, be sure to free it with free() to avoid a memory leak. NOTE: in Postgres 6.4 and later, the be_pid is the notifying backend's, whereas in earlier versions it was always your own backend's PID.

The second sample program gives an example of the use of asynchronous notification.

PQnotifies() does not actually read backend data; it just returns messages previously absorbed by another `libpq` function. In prior releases of `libpq`, the only way to ensure timely receipt of NOTIFY messages was to constantly submit queries, even empty ones, and then check PQnotifies() after each PQexec(). While this still works, it is deprecated as a waste of processing power. A better way to check for NOTIFY messages when you have no useful queries to make is to call PQconsumeInput(), then check PQnotifies(). You can use select(2) to wait for backend data to arrive, thereby using no CPU power unless there is something to do. Note that this will work OK whether you use PQsendQuery/PQgetResult or plain old PQexec for queries. You should, however, remember to check PQnotifies() after each PQgetResult or PQexec to see if any notifications came in during the processing of the query.

53.6. Functions Associated with the COPY Command

The COPY command in Postgres has options to read from or write to the network connection used by `libpq`. Therefore, functions are necessary to access this network connection directly so applications may take advantage of this capability.

These functions should be executed only after obtaining a PGRES_COPY_OUT or PGRES_COPY_IN result object from PQexec or PQgetResult.

- `PQgetline` Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer string of size length.

```
int PQgetline(PGconn *conn,
              char *string,
              int length)
```

  Like fgets(3), this routine copies up to length-1 characters into string. It is like gets(3), however, in that it converts the terminating newline into a null character. PQgetline returns EOF at EOF, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read. Notice that the application must check to see if a new line consists of the two characters "\.", which indicates that the backend server has finished sending the results of the copy command. If the application might receive lines that are more than length-1 characters long, care is needed to be sure one recognizes the "\." line correctly (and does not, for example, mistake the end of a long data line for a terminator line). The code in `../src/bin/psql/psql.c` contains routines that correctly handle the copy protocol.

- `PQgetlineAsync` Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer without blocking.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize)
```

  This routine is similar to PQgetline, but it can be used by applications that must read COPY data asynchronously, that is without blocking. Having issued the COPY command and gotten a PGRES_COPY_OUT response, the application should call PQconsumeInput and PQgetlineAsync until the end-of-data signal is detected. Unlike PQgetline, this routine takes responsibility for detecting end-of-data. On each call, PQgetlineAsync will return data if a complete newline- terminated data line is available in libpq's input buffer, or if the incoming data line is too long to fit in the buffer offered by the caller. Otherwise, no data is returned until the rest of the line arrives. The routine returns -1 if the end-of-copy-data marker has been recognized, or 0 if no data is available, or a positive number giving the number of bytes of data returned. If -1 is returned, the caller must next call PQendcopy, and then return to normal processing. The data returned will not extend beyond a newline character. If possible a whole line will be returned at one time. But if the buffer offered by the caller is too small to hold a line sent by the backend, then a partial data line will be returned. This can be detected by testing whether the last returned byte is '\n' or not. The returned string is not null-terminated. (If you want to add a terminating null, be sure to pass a bufsize one smaller than the room actually available.)

- `PQputline` Sends a null-terminated string to the backend server. Returns 0 if OK, EOF if unable to send the string.

```
int PQputline(PGconn *conn,
              char *string);
```

  Note the application must explicitly send the two characters "\." on a final line to indicate to the backend that it has finished sending its data.

- `PQputnbytes` Sends a non-null-terminated string to the backend server. Returns 0 if OK, EOF if unable to send the string.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

  This is exactly like PQputline, except that the data buffer need not be null-terminated since the number of bytes to send is specified directly.

- `PQendcopy` Syncs with the backend. This function waits until the backend has finished the copy. It should either be issued when the last string has been sent to the backend using PQputline or when the last string has been received from the backend using PGgetline. It must be issued or the backend may get "out of sync" with the frontend. Upon return from this function, the backend is ready to receive the next query. The return value is 0 on successful completion, nonzero otherwise.

  ```
  int PQendcopy(PGconn *conn);
  ```

  As an example:

  ```
  PQexec(conn, "create table foo (a int4, b char16, d float8)");
  PQexec(conn, "copy foo from stdin");
  PQputline(conn, "3<TAB>hello world<TAB>4.5\n");
  PQputline(conn,"4<TAB>goodbye world<TAB>7.11\n");
  ...
  PQputline(conn,"\\.\n");
  PQendcopy(conn);
  ```

When using PQgetResult, the application should respond to a PGRES_COPY_OUT result by executing PQgetline repeatedly, followed by PQendcopy after the terminator line is seen. It should then return to the PQgetResult loop until PQgetResult returns NULL. Similarly a PGRES_COPY_IN result is processed by a series of PQputline calls followed by PQendcopy, then return to the PQgetResult loop. This arrangement will ensure that a copy in or copy out command embedded in a series of SQL commands will be executed correctly. Older applications are likely to submit a copy in or copy out via PQexec and assume that the transaction is done after PQendcopy. This will work correctly only if the copy in/out is the only SQL command in the query string.

53.7. `libpq` Tracing Functions

- `PQtrace` Enable tracing of the frontend/backend communication to a debugging file stream.

  ```
  void PQtrace(PGconn *conn
               FILE *debug_port)
  ```

- `PQuntrace` Disable tracing started by PQtrace

  ```
  void PQuntrace(PGconn *conn)
  ```

53.8. `libpq` Control Functions

- `PQsetNoticeProcessor` Control reporting of notice and warning messages generated by libpq.

  ```
  void PQsetNoticeProcessor (PGconn * conn,
          void (*noticeProcessor) (void * arg, const char * message),
          void * arg)
  ```

By default, `libpq` prints "notice" messages from the backend on stderr, as well as a few error messages that it generates by itself. This behavior can be overridden by supplying a callback function that does something else with the messages. The callback function is passed the text of the error message (which includes a trailing newline), plus a void pointer that is the same one passed to `PQsetNoticeProcessor`. (This pointer can be used to access application-specific state if needed.) The default notice processor is simply

```
static void
```

```
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

To use a special notice processor, call `PQsetNoticeProcessor` just after creation of a new PGconn object.

53.9. User Authentication Functions

The frontend/backend authentication process is handled by `PQconnectdb` without any further intervention. The authentication method is now determined entirely by the DBA (see pga_hba.conf(5)). The following routines no longer have any effect and should not be used.

- `fe_getauthname` Returns a pointer to static space containing whatever name the user has authenticated. Use of this routine in place of calls to getenv(3) or getpwuid(3) by applications is highly recommended, as it is entirely possible that the authenticated user name is not the same as value of the USER environment variable or the user's entry in `/etc/passwd`.

  ```
  char *fe_getauthname(char* errorMessage)
  ```

- `fe_setauthsvc` Specifies that `libpq` should use authentication service name rather than its compiled-in default. This value is typically taken from a command-line switch.

  ```
  void fe_setauthsvc(char *name,
                     char* errorMessage)
  ```

  Any error messages from the authentication attempts are returned in the errorMessage argument.

53.10. Environment Variables

The following environment variables can be used to select default connection parameter values, which will be used by PQconnectdb or PQsetdbLogin if no value is directly specified by the calling code. These are useful to avoid hard-coding database names into simple application programs.

- PGHOST sets the default server name. If a non-zero-length string is specified, TCP/IP communication is used. Without a host name, libpq will connect using a local Unix domain socket.

- PGPORT sets the default port or local Unix domain socket file extension for communicating with the Postgres backend.

- PGDATABASE sets the default Postgres database name.

- PGUSER sets the username used to connect to the database and for authentication.

- PGPASSWORD sets the password used if the backend demands password authentication.

- PGREALM sets the Kerberos realm to use with Postgres, if it is different from the local realm. If PGREALM is set, Postgres applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the backend.

- PGOPTIONS sets additional runtime options for the Postgres backend.

- PGTTY sets the file or tty on which debugging messages from the backend server are displayed.

The following environment variables can be used to specify user-level default behavior for every Postgres session:

- PGDATESTYLE sets the default style of date/time representation.
- PGTZ sets the default time zone.

The following environment variables can be used to specify default internal behavior for every Postgres session:

- PGGEQO sets the default mode for the genetic optimizer.
- PGRPLANS sets the default mode to allow or disable right-sided plans in the optimizer.
- PGCOSTHEAP sets the default cost for heap searches for the optimizer.
- PGCOSTINDEX sets the default cost for indexed searches for the optimizer.
- PGQUERY_LIMIT sets the maximum number of rows returned by a query.

Refer to the `SET` SQL command for information on correct values for these environment variables.

53.11. Caveats

The query buffer is 8192 bytes long, and queries over that length will be rejected.

53.12. Sample Programs

53.12.1. Sample Program 1

```
/*
 * testlibpq.c Test the C version of Libpq, the Postgres frontend
 * library.
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
```

```
            *pgtty;
   char       *dbName;
   int         nFields;
   int         i,
               j;

   /* FILE *debug; */

   PGconn     *conn;
   PGresult   *res;

   /*
    * begin, by setting the parameters for a backend connection if
the
    * parameters are null, then the system will try to use reasonable
    * defaults by looking up environment variables or, failing that,
    * using hardwired constants
    */
   pghost = NULL;              /* host name of the backend server */
   pgport = NULL;              /* port of the backend server */
   pgoptions = NULL;           /* special options to start up the
backend
                                * server */
   pgtty = NULL;               /* debugging tty for the backend
server */
   dbName = "template1";

   /* make a connection to the database */
   conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

   /*
    * check to see that the backend connection was successfully made
    */
   if (PQstatus(conn) == CONNECTION_BAD)
   {
       fprintf(stderr, "Connection to database '%s' failed.\n",
dbName);
       fprintf(stderr, "%s", PQerrorMessage(conn));
       exit_nicely(conn);
   }

   /* debug = fopen("/tmp/trace.out","w"); */
   /* PQtrace(conn, debug);  */

   /* start a transaction block */
   res = PQexec(conn, "BEGIN");
   if (PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "BEGIN command failed\n");
       PQclear(res);
       exit_nicely(conn);
   }

   /*
```

```
     * should PQclear PGresult whenever it is no longer needed to
avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch instances from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor CURSOR FOR select * from
pg_database");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);
    res = PQexec(conn, "FETCH ALL in mycursor");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the instances */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);
```

```
    /* fclose(debug); */
}
```

53.12.2. Sample Program 2

```
/*
 * testlibpq2.c Test of the asynchronous notification interface
 *
 * populate a database with the following:
 *
 * CREATE TABLE TBL1 (i int4);
 *
 * CREATE TABLE TBL2 (i int4);
 *
 * CREATE RULE r1 AS ON INSERT TO TBL1 DO [INSERT INTO TBL2 values
 * (new.i); NOTIFY TBL2];
 *
 * Then start up this program After the program has begun, do
 *
 * INSERT INTO TBL1 values (10);
 *
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
```

```
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
server */
    dbName = getenv("USER");    /* change this to the name of your
test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n",
dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "LISTEN TBL2");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
avoid
     * memory leaks
     */
    PQclear(res);

    while (1)
    {

        /*
         * wait a little bit between checks; waiting with select()
         * would be more efficient.
         */
        sleep(1);
        /* collect any asynchronous backend messages */
        PQconsumeInput(conn);
        /* check for asynchronous notify messages */
```

```
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                 "ASYNC NOTIFY of '%s' from backend pid '%d' received
\n",
                    notify->relname, notify->be_pid);
            free(notify);
        }
    }

    /* close the connection to the database and cleanup */
    PQfinish(conn);

}
```

53.12.3. Sample Program 3

```
/*
 * testlibpq3.c Test the C version of Libpq, the Postgres frontend
 * library. tests the binary cursor interface
 *
 *
 *
 * populate a database by doing the following:
 *
 * CREATE TABLE test1 (i int4, d float4, p polygon);
 *
 * INSERT INTO test1 values (1, 3.567, '(3.0, 4.0, 1.0,
 * 2.0)'::polygon);
 *
 * INSERT INTO test1 values (2, 89.05, '(4.0, 3.0, 2.0,
 * 1.0)'::polygon);
 *
 * the expected output is:
 *
 * tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
 * bytes) 2 points   boundbox = (hi=3.000000/4.000000, lo =
 * 1.000000,2.000000) tuple 1: got i = (4 bytes) 2, d = (4 bytes)
 * 89.050003, p = (4 bytes) 2 points   boundbox =
 * (hi=4.000000/3.000000, lo = 2.000000,1.000000)
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo-decls.h"    /* for the POLYGON type */

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}
```

```
main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;
    int         i_fnum,
                d_fnum,
                p_fnum;
    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use reasonable
     * defaults by looking up environment variables or, failing that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */

    dbName = getenv("USER");    /* change this to the name of your
 test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n",
 dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
```

```
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch instances from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR select *
from test1");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in mycursor");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    i_fnum = PQfnumber(res, "i");
    d_fnum = PQfnumber(res, "d");
    p_fnum = PQfnumber(res, "p");

    for (i = 0; i < 3; i++)
    {
        printf("type[%d] = %d, size[%d] = %d\n",
               i, PQftype(res, i),
               i, PQfsize(res, i));
    }
    for (i = 0; i < PQntuples(res); i++)
    {
        int        *ival;
        float      *dval;
        int         plen;
        POLYGON    *pval;

        /* we hard-wire this to the 3 fields we know about */
        ival = (int *) PQgetvalue(res, i, i_fnum);
```

```
        dval = (float *) PQgetvalue(res, i, d_fnum);
        plen = PQgetlength(res, i, p_fnum);

        /*
         * plen doesn't include the length field so need to
         * increment by VARHDSZ
         */
        pval = (POLYGON *) malloc(plen + VARHDRSZ);
        pval->size = plen;
        memmove((char *) &pval->npts, PQgetvalue(res, i, p_fnum),
 plen);
        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d,\n",
               PQgetlength(res, i, i_fnum), *ival);
        printf(" d = (%d bytes) %f,\n",
               PQgetlength(res, i, d_fnum), *dval);
        printf(" p = (%d bytes) %d points \tboundbox = (hi=%f/%f, lo =
 %f,%f)\n",
               PQgetlength(res, i, d_fnum),
               pval->npts,
               pval->boundbox.xh,
               pval->boundbox.yh,
               pval->boundbox.xl,
               pval->boundbox.yl);
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

}
```

Chapter 54. pgtcl

`pgtcl` is a tcl package for front-end programs to interface with Postgres backends. It makes most of the functionality of `libpq` available to tcl scripts.

This package was originally written by Jolly Chen.

54.1. Commands

Table 54.1. `pgtcl` Commands

Command	Description
pg_connect	opens a connection to the backend server
pg_disconnect	closes a connection
pg_conndefaults	get connection options and their defaults
pg_exec	send a query to the backend
pg_result	manipulate the results of a query
pg_select	loop over the result of a SELECT statement
pg_listen	establish a callback for NOTIFY messages
pg_lo_creat	create a large object
pg_lo_open	open a large object
pg_lo_close	close a large object
pg_lo_read	read a large object
pg_lo_write	write a large object
pg_lo_lseek	seek to a position in a large object
pg_lo_tell	return the current seek position of a large object
pg_lo_unlink	delete a large object
pg_lo_import	import a Unix file into a large object
pg_lo_export	export a large object into a Unix file

These commands are described further on subsequent pages.

The pg_lo* routines are interfaces to the Large Object features of Postgres. The functions are designed to mimic the analogous file system functions in the standard Unix file system interface. The pg_lo* routines should be used within a BEGIN/END transaction block because the file descriptor returned by pg_lo_open is only valid for the current transaction. pg_lo_import and pg_lo_export MUST be used in a BEGIN/END transaction block.

54.2. Examples

Here's a small example of how to use the routines:

```
# getDBs :
#   get the names of all the databases at a given host and port number
#   with the defaults being the localhost and port 5432
```

```
#   return them in alphabetical order
proc getDBs { {host "localhost"} {port "5432"} } {
    # datnames is the list to be result
    set conn [pg_connect template1 -host $host -port $port]
    set res [pg_exec $conn "SELECT datname FROM pg_database ORDER BY
 datname"]
    set ntups [pg_result $res -numTuples]
    for {set i 0} {$i < $ntups} {incr i} {
 lappend datnames [pg_result $res -getTuple $i]
    }
    pg_result $res -clear
    pg_disconnect $conn
    return $datnames
}
```

54.3. pgtcl Command Reference Information

Name

pg_connect — opens a connection to the backend server

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_connect -conninfo connectOptions
pg_connect dbName [-host hostName]
  [-port portNumber] [-tty pqtty]
  [-options optionalBackendArgs]
```

Inputs (new style)

connectOptions

A string of connection options, each written in the form keyword = value.

Inputs (old style)

dbName

Specifies a valid database name.

[-host *hostName*]

Specifies the domain name of the backend server for *dbName*.

[-port *portNumber*]

Specifies the IP port number of the backend server for *dbName*.

[-tty *pqtty*]

Specifies file or tty for optional debug output from backend.

[-options *optionalBackendArgs*]

Specifies options for the backend server for *dbName*.

Outputs

dbHandle

If successful, a handle for a database connection is returned. Handles start with the prefix "pgsql".

Description

`pg_connect` opens a connection to the Postgres backend.

Two syntaxes are available. In the older one, each possible option has a separate option switch in the pg_connect statement. In the newer form, a single option string is supplied that can contain multiple option values. See `pg_conndefaults` for info about the available options in the newer syntax.

Usage

XXX thomas 1997-12-24

Name

pg_disconnect — closes a connection to the backend server

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_disconnect` *`dbHandle`*

Inputs

`dbHandle`

Specifies a valid database handle.

Outputs

None

Description

`pg_disconnect` closes a connection to the Postgres backend.

Name

pg_conndefaults — obtain information about default connection parameters

Synopsis

<refsynopsisdivinfo>1998-10-07</refsynopsisdivinfo>

```
pg_conndefaults
```

Inputs

None.

Outputs

option list

The result is a list describing the possible connection options and their current default values. Each entry in the list is a sublist of the format:

{optname label dispchar dispsize value}

where the optname is usable as an option in `pg_connect -conninfo`.

Description

`pg_conndefaults` returns info about the connection options available in `pg_connect -conninfo` and the current default value for each option.

Usage

pg_conndefaults

Name

pg_exec — send a query string to the backend

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_exec dbHandle queryString
```

Inputs

dbHandle

Specifies a valid database handle.

queryString

Specifies a valid SQL query.

Outputs

resultHandle

A Tcl error will be returned if Pgtcl was unable to obtain a backend response. Otherwise, a query result object is created and a handle for it is returned. This handle can be passed to `pg_result` to obtain the results of the query.

Description

`pg_exec` submits a query to the Postgres backend and returns a result. Query result handles start with the connection handle and add a period and a result number.

Note that lack of a Tcl error is not proof that the query succeeded! An error message returned by the backend will be processed as a query result with failure status, not by generating a Tcl error in pg_exec.

Name

pg_result — get information about a query result

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_result resultHandle resultOption
```

Inputs

resultHandle

The handle for a query result.

resultOption

Specifies one of several possible options.

Options

-status

the status of the result.

-error

the error message, if the status indicates error; otherwise an empty string.

-conn

the connection that produced the result.

-oid

if the command was an INSERT, the OID of the inserted tuple; otherwise an empty string.

-numTuples

the number of tuples returned by the query.

-numAttrs

the number of attributes in each tuple.

-assign arrayName

assign the results to an array, using subscripts of the form (tupno,attributeName).

-assignbyidx arrayName ?appendstr?

assign the results to an array using the first attribute's value and the remaining attributes' names as keys. If appendstr is given then it is appended to each key. In short, all but the first field of each tuple are stored into the array, using subscripts of the form (firstFieldValue,fieldNameAppendStr).

-getTuple tupleNumber

returns the fields of the indicated tuple in a list. Tuple numbers start at zero.

-tupleArray tupleNumber arrayName

stores the fields of the tuple in array arrayName, indexed by field names. Tuple numbers start at zero.

-attributes

returns a list of the names of the tuple attributes.

-lAttributes

returns a list of sublists, {name ftype fsize} for each tuple attribute.

-clear

clear the result query object.

Outputs

The result depends on the selected option, as described above.

Description

`pg_result` returns information about a query result created by a prior `pg_exec`.

You can keep a query result around for as long as you need it, but when you are done with it, be sure to free it by executing `pg_result -clear`. Otherwise, you have a memory leak, and Pgtcl will eventually start complaining that you've created too many query result objects.

Name

pg_select — loop over the result of a SELECT statement

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_select dbHandle queryString
  arrayVar queryProcedure
```

Inputs

dbHandle

Specifies a valid database handle.

queryString

Specifies a valid SQL select query.

arrayVar

Array variable for tuples returned.

queryProcedure

Procedure run on each tuple found.

Outputs

resultHandle

the return result is either an error message or a handle for a query result.

Description

`pg_select` submits a SELECT query to the Postgres backend, and executes a given chunk of code for each tuple in the result. The *queryString* must be a SELECT statement. Anything else returns an error. The *arrayVar* variable is an array name used in the loop. For each tuple, *arrayVar* is filled in with the tuple field values, using the field names as the array indexes. Then the *queryProcedure* is executed.

Usage

This would work if table "table" has fields "control" and "name" (and, perhaps, other fields):

```
pg_select $pgconn "SELECT * from table" array {
 puts [format "%5d %s" array(control) array(name)]
}
```

Name

pg_listen — sets or changes a callback for asynchronous NOTIFY messages

Synopsis

<refsynopsisdivinfo>1998-5-22</refsynopsisdivinfo>

```
pg_listen dbHandle notifyName callbackCommand
```

Inputs

dbHandle

Specifies a valid database handle.

notifyName

Specifies the notify condition name to start or stop listening to.

callbackCommand

If present and not empty, provides the command string to execute when a matching notification arrives.

Outputs

None

Description

`pg_listen` creates, changes, or cancels a request to listen for asynchronous NOTIFY messages from the Postgres backend. With a callbackCommand parameter, the request is established, or the command string of an already existing request is replaced. With no callbackCommand parameter, a prior request is canceled.

After a `pg_listen` request is established, the specified command string is executed whenever a NOTIFY message bearing the given name arrives from the backend. This occurs when any Postgres client application issues a NOTIFY command referencing that name. (Note that the name can be, but does not have to be, that of an existing relation in the database.) The command string is executed from the Tcl idle loop. That is the normal idle state of an application written with Tk. In non-Tk Tcl shells, you can execute `update` or `vwait` to cause the idle loop to be entered.

You should not invoke the SQL statements LISTEN or UNLISTEN directly when using `pg_listen`. Pgtcl takes care of issuing those statements for you. But if you want to send a NOTIFY message yourself, invoke the SQL NOTIFY statement using `pg_exec`.

Name

pg_lo_creat — create a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_creat conn mode
```

Inputs

conn

Specifies a valid database connection.

mode

Specifies the access mode for the large object

Outputs

objOid

The oid of the large object created.

Description

`pg_lo_creat` creates an Inversion Large Object.

Usage

mode can be any OR'ing together of INV_READ, INV_WRITE, and INV_ARCHIVE. The OR delimiter character is "|".

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

Name

pg_lo_open — open a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_open` *`conn objOid mode`*

Inputs

`conn`

Specifies a valid database connection.

`objOid`

Specifies a valid large object oid.

`mode`

Specifies the access mode for the large object

Outputs

`fd`

A file descriptor for use in later pg_lo* routines.

Description

`pg_lo_open` open an Inversion Large Object.

Usage

Mode can be either "r", "w", or "rw".

Name

pg_lo_close — close a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_close` *`conn fd`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

A file descriptor for use in later pg_lo* routines.

Outputs

None

Description

`pg_lo_close` closes an Inversion Large Object.

Usage

Name

pg_lo_read — read a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_read` *`conn fd bufVar len`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

`bufVar`

Specifies a valid buffer variable to contain the large object segment.

`len`

Specifies the maximum allowable size of the large object segment.

Outputs

None

Description

`pg_lo_read` reads at most *`len`* bytes from a large object into a variable named *`bufVar`*.

Usage

`bufVar` must be a valid variable name.

Name

pg_lo_write — write a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_write conn fd buf len
```

Inputs

conn

Specifies a valid database connection.

fd

File descriptor for the large object from pg_lo_open.

buf

Specifies a valid string variable to write to the large object.

len

Specifies the maximum size of the string to write.

Outputs

None

Description

`pg_lo_write` writes at most *len* bytes to a large object from a variable *buf*.

Usage

buf must be the actual string to write, not a variable name.

Name

pg_lo_lseek — seek to a position in a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_lseek` *`conn fd offset whence`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

`offset`

Specifies a zero-based offset in bytes.

`whence`

whence can be "SEEK_CUR", "SEEK_END", or "SEEK_SET"

Outputs

None

Description

`pg_lo_lseek` positions to *`offset`* bytes from the beginning of the large object.

Usage

`whence` can be "SEEK_CUR", "SEEK_END", or "SEEK_SET".

Name

pg_lo_tell — return the current seek position of a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_tell` *`conn fd`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

Outputs

`offset`

A zero-based offset in bytes suitable for input to `pg_lo_lseek`.

Description

`pg_lo_tell` returns the current to *`offset`* in bytes from the beginning of the large object.

Usage

Name

pg_lo_unlink — delete a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_unlink` *`conn lobjId`*

Inputs

`conn`

Specifies a valid database connection.

`lobjId`

Identifier for a large object. XXX Is this the same as objOid in other calls?? - thomas 1998-01-11

Outputs

None

Description

`pg_lo_unlink` deletes the specified large object.

Usage

Name

pg_lo_import — import a large object from a Unix file

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_import conn filename
```

Inputs

conn

Specifies a valid database connection.

filename

Unix file name.

Outputs

None XXX Does this return a lobjId? Is that the same as the objOid in other calls? thomas - 1998-01-11

Description

`pg_lo_import` reads the specified file and places the contents into a large object.

Usage

`pg_lo_import` must be called within a BEGIN/END transaction block.

Name

pg_lo_export — export a large object to a Unix file

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_export conn lobjId filename
```

Inputs

conn

Specifies a valid database connection.

lobjId

Large object identifier. XXX Is this the same as the objOid in other calls?? thomas - 1998-01-11

filename

Unix file name.

Outputs

None XXX Does this return a lobjId? Is that the same as the objOid in other calls? thomas - 1998-01-11

Description

`pg_lo_export` writes the specified large object into a Unix file.

Usage

`pg_lo_export` must be called within a BEGIN/END transaction block.

Chapter 55. ODBC Interface

Tim Goeke
Thomas Lockhart

Note

Background information originally by Tim Goeke[1]

ODBC (Open Database Connectivity) is an abstract API which allows you to write applications which can interoperate with various RDBMS servers. ODBC provides a product-neutral interface between frontend applications and database servers, allowing a user or developer to write applications which are transportable between servers from different manufacturers..

55.1. Background

The ODBC API matches up on the backend to an ODBC-compatible data source. This could be anything from a text file to an Oracle or Postgres RDBMS.

The backend access come from ODBC drivers, or vendor specifc drivers that allow data access. psqlODBC is such a driver, along with others that are available, such as the OpenLink ODBC drivers.

Once you write an ODBC application, you *should* be able to connect to *any* back end database, regardless of the vendor, as long as the database schema is the same.

For example. you could have MS SQL Server and Postgres servers which have exactly the same data. Using ODBC, your Windows application would make exactly the same calls and the back end data source would look the same (to the Windows app).

Insight Distributors[2] provides active and ongoing support for the core psqlODBC distribution. They provide a FAQ[3], ongoing development on the code base, and actively participate on the interfaces mailing list[4].

55.2. Windows Applications

In the real world, differences in drivers and the level of ODBC support lessens the potential of ODBC:

- Access, Delphi, and Visual Basic all support ODBC directly.
- Under C++, such as Visual C++, you can use the C++ ODBC API.
- In Visual C++, you can use the CRecordSet class, which wraps the ODBC API set within an MFC 4.2 class. This is the easiest route if you are doing Windows C++ development under Windows NT.

55.2.1. Writing Applications

“If I write an application for Postgres can I write it using ODBC calls to the Postgres server, or is that only when another database program like MS SQL Server or Access needs to access the data?”

The ODBC API is the way to go. For Visual C++ coding you can find out more at Microsoft's web site or in your VC++ docs.

[1] mailto:tgoeke@xpressway.com
[2] http://www.insightdist.com/
[3] http://www.insightdist.com/psqlodbc/
[4] mailto:interfaces@postgresql.org

Visual Basic and the other RAD tools have Recordset objects that use ODBC directly to access data. Using the data-aware controls, you can quickly link to the ODBC back end database (*very* quickly).

Playing around with MS Access will help you sort this out. Try using `File->Get External Data`.

Tip

You'll have to set up a DSN first.

55.3. Unix Installation

ApplixWare has an ODBC database interface supported on at least some platforms. ApplixWare v4.4.1 has been demonstrated under Linux with Postgres v6.4 using the psqlODBC driver contained in the Postgres distribution.

55.3.1. Building the Driver

The first thing to note about the psqlODBC driver (or any ODBC driver) is that there must exist a driver manager on the system where the ODBC driver is to be used. There exists a freeware ODBC driver for Unix called iodbc which can be obtained from various locations on the Net, including at AS200[5]. Instructions for installing iodbc are beyond the scope of this document, but there is a `README` that can be found inside the iodbc compressed .shar file that should explain how to get it up and running.

Having said that, any driver manager that you can find for your platform should support the psqlODBC driver or any ODBC driver.

The Unix configuration files for psqlODBC have recently been extensively reworked to allow for easy building on supported platforms as well as to allow for support of other Unix platforms in the future. The new configuration and build files for the driver should make it a simple process to build the driver on the supported platforms. Currently these include Linux and FreeBSD but we are hoping other users will contribute the necessary information to quickly expand the number of platforms for which the driver can be built.

There are actually two separate methods to build the driver depending on how you received it and these differences come down to only where and how to run configure and make. The driver can be built in a standalone, client-only installation, or can be built as a part of the main Postgres distribution. The standalone installation is convenient if you have ODBC client applications on multiple, heterogeneous platforms. The integrated installation is convenient when the target client is the same as the server, or when the client and server have similar runtime configurations.

Specifically if you have received the psqlODBC driver as part of the Postgres distribution (from now on referred to as an "integrated" build) then you will configure and make the ODBC driver from the top level source directory of the Postgres distribution along with the rest of its libraries. If you received the driver as a standalone package than you will run configure and make from the directory in which you unpacked the driver source.

Integrated Installation

This installation procedure is appropriate for an integrated installation.

1. Specify the `--with-odbc` command-line argument for src/configure:

   ```
   % ./configure --with-odbc
   ```

[5] http://www.as220.org/FreeODBC/iodbc-2.12.shar.Z

```
% make
```

2. Rebuild the Postgres distribution:

```
% make install
```

Once configured, the ODBC driver will be built and installed into the areas defined for the other components of the Postgres system. The installation-wide ODBC configuration file will be placed into the top directory of the Postgres target tree (`POSTGRESDIR`). This can be overridden from the make command-line as

```
% make ODBCINST=filename install
```

Pre-v6.4 Integrated Installation

If you have a Postgres installation older than v6.4, you have the original source tree available, and you want to use the newest version of the ODBC driver, then you may want to try this form of installation.

1. Copy the output tar file to your target system and unpack it into a clean directory.

2. From the directory containing the sources, type:

```
% ./configure
% make
% make POSTGRESDIR=PostgresTopDir install
```

3. (Optional) If you would like to install components into different trees, then you can specify various destinations explicitly:

```
% make BINDIR=bindir  LIBDIR=libdir  HEADERDIR=headerdir
 ODBCINST=instfile install
```

Standalone Installation

A standalone installation is not integrated with or built on the normal Postgres distribution. It should be best suited for building the ODBC driver for multiple, heterogeneous clients who do not have a locally-installed Postgres source tree.

The default location for libraries and headers for the standalone installation is `/usr/local/lib` and `/usr/local/include/iodbc`, respectively. There is another system wide configuration file that gets installed as `/share/odbcinst.ini` (if `/share` exists) or as `/etc/odbcinst.ini` (if `/share` does not exist).

Note

Installation of files into `/share` or `/etc` requires system root privileges. Most installation steps for Postgres do not have this requirement, and you can choose another destination which is writable by your non-root Postgres superuser account instead.

1. The standalone installation distribution can be built from the Postgres distribution or may be obtained from Insight Distributors[6], the current maintainers of the non-Unix sources.

   Copy the zip or gzipped tarfile to an empty directory. If using the zip package unzip it with the command

[6] http://www.insightdist.com/psqlodbc

```
% unzip -a packagename
```

The `-a` option is necessary to get rid of DOS CR/LF pairs in the source files.

If you have the gzipped tar package than simply run

```
tar -xzf packagename
```

- (Optional) To create a tar file for a complete standalone installation from the main Postgres source tree:

2. Configure the main Postgres distribution.

3. Create the tar file:

```
% cd interfaces/odbc
% make standalone
```

4. Copy the output tar file to your target system. Be sure to transfer as a binary file if using ftp.

5. Unpack the tar file into a clean directory.

6. Configure the standalone installation:

```
% ./configure
```

The configuration can be done with options:

```
% ./configure --prefix=rootdir --with-odbc=inidir
```

where `--prefix` installs the libraries and headers in the directories *rootdir*`/lib` and *rootdir*`/include/iodbc`, and `--with-odbc` installs `odbcinst.ini` in the specified directory.

Note that both of these options can also be used from the integrated build but be aware that *when used in the integrated build* `--prefix` will also apply to the rest of your Postgres installation. `--with-odbc` applies only to the configuration file `odbcinst.ini`.

7. Compile and link the source code:

```
% make ODBCINST=instdir
```

You can also override the default location for installation on the 'make' command line. This only applies to the installation of the library and header files. Since the driver needs to know the location of the odbcinst.ini file attempting to override the enviroment variable that specifies its installation directory will probably cause you headaches. It is safest simply to allow the driver to install the odbcinst.ini file in the default directory or the directory you specified on the './configure' command line with --with-odbc.

8. Install the source code:

```
% make POSTGRESDIR=targettree install
```

To override the library and header installation directories separately you need to pass the correct installation variables on the `make install` command line. These variables are `LIBDIR`, `HEADERDIR` and `ODBCINST`. Overriding `POSTGRESDIR` on the make command line will cause

LIBDIR and HEADERDIR to be rooted at the new directory you specify. ODBCINST is independent of POSTGRESDIR.

Here is how you would specify the various destinations explicitly:

```
% make BINDIR=bindir LIBDIR=libdir HEADERDIR=headerdir install
```

For example, typing

```
% make POSTGRESDIR=/opt/psqlodbc install
```

(after you've used ./configure and make) will cause the libraries and headers to be installed in the directories /opt/psqlodbc/lib and /opt/psqlodbc/include/iodbc respectively.

The command

```
% make POSTGRESDIR=/opt/psqlodbc HEADERDIR=/usr/local install
```

should cause the libraries to be installed in /opt/psqlodbc/lib and the headers in /usr/local/include/iodbc. If this doesn't work as expected please contact one of the maintainers.

55.4. Configuration Files

~/.odbc.ini contains user-specified access information for the psqlODBC driver. The file uses conventions typical for Windows Registry files, but despite this restriction can be made to work.

The .odbc.ini file has three required sections. The first is [ODBC Data Sources] which is a list of arbitrary names and descriptions for each database you wish to access. The second required section is the Data Source Specification and there will be one of these sections for each database. Each section must be labeled with the name given in [ODBC Data Sources] and must contain the following entries:

```
Driver = POSTGRESDIR/lib/libpsqlodbc.so
Database=DatabaseName
Servername=localhost
Port=5432
```

Tip

Remember that the Postgres database name is usually a single word, without path names of any sort. The Postgres server manages the actual access to the database, and you need only specify the name from the client.

Other entries may be inserted to control the format of the display. The third required section is [ODBC] which must contain the InstallDir keyword and which may contain other options.

Here is an example .odbc.ini file, showing access information for three databases:

```
[ODBC Data Sources]
DataEntry = Read/Write Database
QueryOnly = Read-only Database
Test = Debugging Database
Default = Postgres Stripped

[DataEntry]
ReadOnly = 0
Servername = localhost
```

```
Database = Sales

[QueryOnly]
ReadOnly = 1
Servername = localhost
Database = Sales

[Test]
Debug = 1
CommLog = 1
ReadOnly = 0
Servername = localhost
Username = tgl
Password = "no$way"
Port = 5432
Database = test

[Default]
Servername = localhost
Database = tgl
Driver = /opt/postgres/current/lib/libpsqlodbc.so

[ODBC]
InstallDir = /opt/applix/axdata/axshlib
```

55.5. ApplixWare

55.5.1. Configuration

ApplixWare must be configured correctly in order for it to be able to access the Postgres ODBC software drivers.

Enabling ApplixWare Database Access

These instructions are for the 4.4.1 release of ApplixWare on Linux. Refer to the *Linux Sys Admin* on-line book for more detailed information.

1. You must modify `axnet.cnf` so that `elfodbc` can find `libodbc.so` (the ODBC driver manager) shared library. This library is included with the ApplixWare distribution, but `axnet.cnf` needs to be modified to point to the correct location.

   As root, edit the file *`applixroot`*`/applix/axdata/axnet.cnf`.

   a. At the bottom of `axnet.cnf`, find the line that starts with

      ```
      #libFor elfodbc /ax/...
      ```

   b. Change line to read

      ```
      libFor elfodbc applixroot/applix/axdata/axshlib/lib
      ```

      which will tell elfodbc to look in this directory for the ODBC support library. If you have installed applix somewhere else, change the path accordingly.

2. Create `.odbc.ini` as described above. You may also want to add the flag

   ```
   TextAsLongVarchar=0
   ```

   to the database-specific portion of `.odbc.ini` so that text fields will not be shown as `**BLOB**`.

Testing ApplixWare ODBC Connections

1. Bring up Applix Data
2. Select the Postgres database of interest.
   a. Select `Query->Choose Server`.
   b. Select ODBC, and click `Browse`. The database you configured in `.odbc.ini` should be shown. Make sure that the `Host: field` is empty (if it is not, axnet will try to contact axnet on another machine to look for the database).
   c. Select the database in the box that was launched by `Browse`, then click `OK`.
   d. Enter username and password in the login identification dialog, and click `OK`.

   You should see “Starting elfodbc server” in the lower left corner of the data window. If you get an error dialog box, see the debugging section below.
3. The 'Ready' message will appear in the lower left corner of the data window. This indicates that you can now enter queries.
4. Select a table from Query->Choose tables, and then select Query->Query to access the database. The first 50 or so rows from the table should appear.

55.5.2. Common Problems

The following messages can appear while trying to make an ODBC connection through Applix Data:

Cannot launch gateway on server

> `elfodbc` can't find `libodbc.so`. Check your `axnet.cnf`.

Error from ODBC Gateway: IM003::[iODBC][Driver Manager]Specified driver could not be loaded

> `libodbc.so` cannot find the driver listed in `.odbc.ini`. Verify the settings.

Server: Broken Pipe

> The driver process has terminated due to some other problem. You might not have an up-to-date version of the Postgres ODBC package.

setuid to 256: failed to launch gateway

> The September release of ApplixWare v4.4.1 (the first release with official ODBC support under Linux) shows problems when usernames exceed eight (8) characters in length. Problem description ontributed by Steve Campbell[7].

[7] mailto:scampbell@lear.com

Author

Contributed by Steve Campbell[8] on 1998-10-20.

The axnet program's security system seems a little suspect. axnet does things on behalf of the user and on a true multiple user system it really should be run with root security (so it can read/write in each user's directory). I would hesitate to recommend this, however, since we have no idea what security holes this creates.

55.5.3. Debugging ApplixWare ODBC Connections

One good tool for debugging connection problems uses the Unix system utility strace.

Debugging with strace

1. Start applixware.
2. Start an strace on the axnet process. For example, if

   ```
   ps -aucx | grep ax
   ```

   shows

   ```
   cary   10432  0.0  2.6  1740   392  ?  S  Oct  9  0:00 axnet
   cary   27883  0.9 31.0 12692  4596  ?  S   10:24  0:04 axmain
   ```

   Then run

   ```
   strace -f -s 1024 -p 10432
   ```

3. Check the strace output.

Note from Cary

Many of the error messages from ApplixWare go to `stderr`, but I'm not sure where `stderr` is sent, so strace is the way to find out.

For example, after getting a “Cannot launch gateway on server”, I ran strace on axnet and got

```
[pid 27947] open("/usr/lib/libodbc.so", O_RDONLY) = -1 ENOENT
    (No such file or directory)
[pid 27947] open("/lib/libodbc.so", O_RDONLY) = -1 ENOENT
    (No such file or directory)
[pid 27947] write(2, "/usr2/applix/axdata/elfodbc:
    can't load library 'libodbc.so'\n", 61) = -1 EIO (I/O error)
```

So what is happening is that applix elfodbc is searching for libodbc.so, but it can't find it. That is why axnet.cnf needed to be changed.

55.5.4. Running the ApplixWare Demo

In order to go through the *ApplixWare Data Tutorial*, you need to create the sample tables that the Tutorial refers to. The ELF Macro used to create the tables tries to use a NULL condition on many of the database columns, and Postgres does not currently allow this option.

[8] mailto:scampbell@lear.com

To get around this problem, you can do the following:

Modifying the ApplixWare Demo

1. Copy `/opt/applix/axdata/eng/Demos/sqldemo.am` to a local directory.
2. Edit this local copy of `sqldemo.am`:
   a. Search for 'null_clause = "NULL"
   b. Change this to null_clause = ""
3. Start Applix Macro Editor.
4. Open the sqldemo.am file from the Macro Editor.
5. Select `File->Compile and Save`.
6. Exit Macro Editor.
7. Start Applix Data.
8. Select `*->Run Macro`
9. Enter the value “sqldemo”, then click `OK`.

   You should see the progress in the status line of the data window (in the lower left corner).
10. You should now be able to access the demo tables.

55.5.5. Useful Macros

You can add information about your database login and password to the standard Applix startup macro file. This is an example `~/axhome/macros/login.am` file: ======= ApplixWare must be configured correctly in order for it to be able to access the Postgres ODBC software drivers.

Enabling ApplixWare Database Access

Note that these instructions are for the 4.4.1 release of ApplixWare on Linux. Refer to the *Linux Sys Admin* on-line book for more detailed information.

1. You must modify `axnet.cnf` so that `elfodbc` can find `libodbc.so` (the ODBC driver manager) shared library. This library is included with the ApplixWare distribution, but `axnet.cnf` needs to be modified to point to the correct location.

   As root, edit the file *`applixroot`*`/applix/axdata/axnet.cnf`.

   a. At the bottom of `axnet.cnf`, find the line that starts with

      ```
      #libFor elfodbc /ax/...
      ```

   b. Change line to read

      ```
      libFor elfodbc applixroot/applix/axdata/axshlib/lib
      ```

      which will tell elfodbc to look in this directory for the ODBC support library. If you have installed applix somewhere else, change the path accordingly.

2. Create `.odbc.ini` as described above. You may also want to add the flag

   `TextAsLongVarchar=0`

   to the database-specific portion of `.odbc.ini` so that text fields will not be shown as `**BLOB**`.

Testing ApplixWare ODBC Connections

1. Bring up Applix Data
2. Select the Postgres database of interest.
   a. Select `Query->Choose Server`.
   b. Select ODBC, and click `Browse`. The database you configured in `.odbc.ini` should be shown. Make sure that the `Host: field` is empty (if it is not, axnet will try to contact axnet on another machine to look for the database).
   c. Select the database in the box that was launched by `Browse`, then click `OK`.
   d. Enter username and password in the login identification dialog, and click `OK`.

   You should see “Starting elfodbc server” in the lower left corner of the data window. If you get an error dialog box, see the debugging section below.
3. The 'Ready' message will appear in the lower left corner of the data window. This indicates that you can now enter queries.
4. Select a table from Query->Choose tables, and then select Query->Query to access the database. The first 50 or so rows from the table should appear.

55.5.6. Common Problems

The following messages can appear while trying to make an ODBC connection through Applix Data:

Cannot launch gateway on server

> `elfodbc` can't find `libodbc.so`. Check your `axnet.cnf`.

Error from ODBC Gateway: IM003::[iODBC][Driver Manager]Specified driver could not be loaded

> `libodbc.so` cannot find the driver listed in `.odbc.ini`. Verify the settings.

Server: Broken Pipe

> The driver process has terminated due to some other problem. You might not have an up-to-date version of the Postgres ODBC package.

55.5.7. Debugging ApplixWare ODBC Connections

One good tool for debugging connection problems uses the Unix system utility strace.

Debugging with strace

1. Start applixware.

2. Start an strace on the axnet process. For example, if

   ```
   ps -aucx | grep ax
   ```

   shows

   ```
   cary   10432  0.0  2.6  1740   392  ?  S  Oct  9  0:00 axnet
   cary   27883  0.9 31.0 12692  4596  ?  S   10:24  0:04 axmain
   ```

   Then run

   ```
   strace -f -s 1024 -p 10432
   ```

3. Check the strace output.

   ### Note from Cary

   Many of the error messages from ApplixWare go to `stderr`, but I'm not sure where `stderr` is sent, so strace is the way to find out.

For example, after getting a “Cannot launch gateway on server”, I ran strace on axnet and got

```
[pid 27947] open("/usr/lib/libodbc.so", O_RDONLY) = -1 ENOENT
    (No such file or directory)
[pid 27947] open("/lib/libodbc.so", O_RDONLY) = -1 ENOENT
    (No such file or directory)
[pid 27947] write(2, "/usr2/applix/axdata/elfodbc:
    can't load library 'libodbc.so'\n", 61) = -1 EIO (I/O error)
```

So what is happening is that applix elfodbc is searching for libodbc.so, but it can't find it. That is why axnet.cnf needed to be changed.

55.5.8. Running the ApplixWare Demo

In order to go through the *ApplixWare Data Tutorial*, you need to create the sample tables that the Tutorial refers to. The ELF Macro used to create the tables tries to use a NULL condition on many of the database columns, and Postgres does not currently allow this option.

To get around this problem, you can do the following:

Modifying the ApplixWare Demo

1. Copy `/opt/applix/axdata/eng/Demos/sqldemo.am` to a local directory.
2. Edit this local copy of `sqldemo.am`:
   a. Search for 'null_clause = "NULL"
   b. Change this to null_clause = ""
3. Start Applix Macro Editor.
4. Open the sqldemo.am file from the Macro Editor.
5. Select `File->Compile and Save`.
6. Exit Macro Editor.

7. Start Applix Data.
8. Select `*->Run Macro`
9. Enter the value “sqldemo”, then click `OK`.

   You should see the progress in the status line of the data window (in the lower left corner).
10. You should now be able to access the demo tables.

55.5.9. Useful Macros

You can add information about your database login and password to the standard Applix startup macro file. This is an example `~/axhome/macros/login.am` file:

```
macro login
    set_set_system_var@("sql_username@","tgl")
    set_system_var@("sql_passwd@","no$way")
endmacro
```

Caution

You should be careful about the file protections on any file containing username and password information.

55.5.10. Supported Platforms

psqlODBC has been built and tested on Linux. There have been reports of success with FreeBSD and with Solaris. There are no known restrictions on the basic code for other platforms which already support Postgres.

Chapter 56. JDBC Interface

Author

Written by Peter T. Mount[1], the author of the JDBC driver.

JDBC is a core API of Java 1.1 and later. It provides a standard set of interfaces to SQL-compliant databases.

Postgres provides a type 4 JDBC Driver. Type 4 indicates that the driver is written in Pure Java, and communicates in the database's own network protocol. Because of this, the driver is platform independent. Once compiled, the driver can be used on any platform.

56.1. Building the JDBC Interface

56.1.1. Compiling the Driver

The driver's source is located in the `src/interfaces/jdbc` directory of the source tree. To compile simply change directory to that directory, and type:

```
% make
```

Upon completion, you will find the archive `postgresql.jar` in the current directory. This is the JDBC driver.

Note

You must use make, not javac, as the driver uses some dynamic loading techniques for performance reasons, and javac cannot cope. The `Makefile` will generate the jar archive.

56.1.2. Installing the Driver

To use the driver, the jar archive postgresql.jar needs to be included in the CLASSPATH.

Example:

I have an application that uses the JDBC driver to access a large database containing astronomical objects. I have the application and the jdbc driver installed in the /usr/local/lib directory, and the java jdk installed in /usr/local/jdk1.1.6.

To run the application, I would use:

export CLASSPATH = \ /usr/local/lib/finder.jar:/usr/local/lib/postgresql.jar:. java uk.org.retep.finder.Main

Loading the driver is covered later on in this chapter.

56.2. Preparing the Database for JDBC

Because Java can only use TCP/IP connections, the Postgres postmaster must be running with the -i flag.

[1] peter@retep.org.uk

Also, the `pg_hba.conf` file must be configured. It's located in the PGDATA directory. In a default installation, this file permits access only by UNIX domain sockets. For the JDBC driver to connect to the same localhost, you need to add something like:

host all 127.0.0.1 255.255.255.255 password

Here access to all databases are possible from the local machine with JDBC.

The JDBC Driver supports trust, ident, password and crypt authentication methods.

56.3. Using the Driver

This section is not intended as a complete guide to JDBC programming, but should help to get you started. For more information refer to the standard JDBC API documentation.

Also, take a look at the examples included with the source. The basic example is used here.

56.4. Importing JDBC

Any source that uses JDBC needs to import the java.sql package, using:

```
import java.sql.*;
```

Important

Do not import the postgresql package. If you do, your source will not compile, as javac will get confused.

56.5. Loading the Driver

Before you can connect to a database, you need to load the driver. There are two methods available, and it depends on your code to the best one to use.

In the first method, your code implicitly loads the driver using the Class.forName() method. For Postgres, you would use:

```
Class.forName(postgresql.Driver);
```

This will load the driver, and while loading, the driver will automatically register itself with JDBC.

Note: The `forName()` method can throw a ClassNotFoundException, so you will need to catch it if the driver is not available.

This is the most common method to use, but restricts your code to use just Postgres. If your code may access another database in the future, and you don't use our extensions, then the second method is advisable.

The second method passes the driver as a parameter to the JVM as it starts, using the -D argument.

Example:

```
% java -Djdbc.drivers=postgresql.Driver example.ImageViewer
```

In this example, the JVM will attempt to load the driver as part of it's initialisation. Once done, the ImageViewer is started.

Now, this method is the better one to use because it allows your code to be used with other databases, without recompiling the code. The only thing that would also change is the URL, which is covered next.

One last thing. When your code then tries to open a Connection, and you get a `No driver available` SQLException being thrown, this is probably caused by the driver not being in the classpath, or the value in the parameter not being correct.

56.6. Connecting to the Database

With JDBC, a database is represented by a URL (Uniform Resource Locator). With Postgres, this takes one of the following forms:

- jdbc:postgresql:*database*
- jdbc:postgresql://*host*/*database*
- jdbc:postgresql://*host*:*port*/*database*

where:

host

The hostname of the server. Defaults to "localhost".

port

The port number the server is listening on. Defaults to the Postgres standard port number (5432).

database

The database name.

To connect, you need to get a Connection instance from JDBC. To do this, you would use the DriverManager.getConnection() method:

Connection db = DriverManager.getConnection(url,user,pwd);

56.7. Issuing a Query and Processing the Result

Any time you want to issue SQL statements to the database, you require a Statement instance. Once you have a Statement, you can use the executeQuery() method to issue a query. This will return a ResultSet instance, which contains the entire result.

56.7.1. Using the Statement Interface

The following must be considered when using the Statement interface:

- You can use a Statement instance as many times as you want. You could create one as soon as you open the connection, and use it for the connections lifetime. You have to remember that only one ResultSet can exist per Statement.

- If you need to perform a query while processing a ResultSet, you can simply create and use another Statement.
- If you are using Threads, and several are using the database, you must use a separate Statement for each thread. Refer to the sections covering Threads and Servlets later in this document if you are thinking of using them, as it covers some important points.

56.7.2. Using the ResultSet Interface

The following must be considered when using the ResultSet interface:

- Before reading any values, you must call `next()`. This returns true if there is a result, but more importantly, it prepares the row for processing.
- Under the JDBC spec, you should access a field only once. It's safest to stick to this rule, although at the current time, the Postgres driver will allow you to access a field as many times as you want.
- You must close a ResultSet by calling `close()` once you have finished with it.
- Once you request another query with the Statement used to create a ResultSet, the currently open instance is closed.

An example is as follows:

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery(select * from mytable);
while(rs.next()) {
    System.out.print(Column 1 returned );
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

56.8. Performing Updates

To perform an update (or any other SQL statement that does not return a result), you simply use the executeUpdate() method:

```
st.executeUpdate(create table basic (a int2, b int2));
```

56.9. Closing the Connection

To close the database connection, simply call the close() method to the Connection:

```
db.close();
```

56.10. Using Large Objects

In Postgres, large objects (also known as *blobs*) are used to hold data in the database that cannot be stored in a normal SQL table. They are stored as a Table/Index pair, and are refered to from your own tables, by an OID value.

Now, there are you methods of using Large Objects. The first is the standard JDBC way, and is documented here. The other, uses our own extension to the api, which presents the libpq large object API to Java,

providing even better access to large objects than the standard. Internally, the driver uses the extension to provide large object support.

In JDBC, the standard way to access them is using the getBinaryStream() method in ResultSet, and setBinaryStream() method in PreparedStatement. These methods make the large object appear as a Java stream, allowing you to use the java.io package, and others, to manipulate the object.

For example, suppose you have a table containing the file name of an image, and a large object containing that image:

```
create table images (imgname name,imgoid oid);
```

To insert an image, you would use:

```
File file = new File(myimage.gif);
FileInputStream fis = new FileInputStream(file);
PreparedStatement ps = conn.prepareStatement(insert into images values
 (?,?));
ps.setString(1,file.getName());
ps.setBinaryStream(2,fis,file.length());
ps.executeUpdate();
ps.close();
fis.close();
```

Now in this example, setBinaryStream transfers a set number of bytes from a stream into a large object, and stores the OID into the field holding a reference to it.

Retrieving an image is even easier (I'm using PreparedStatement here, but Statement can equally be used):

```
PreparedStatement ps = con.prepareStatement(select oid from images
 where name=?);
ps.setString(1,myimage.gif);
ResultSet rs = ps.executeQuery();
if(rs!=null) {
    while(rs.next()) {
        InputStream is = rs.getBinaryInputStream(1);
        // use the stream in some way here
        is.close();
    }
    rs.close();
}
ps.close();
```

Now here you can see where the Large Object is retrieved as an InputStream. You'll also notice that we close the stream before processing the next row in the result. This is part of the JDBC Specification, which states that any InputStream returned is closed when ResultSet.next() or ResultSet.close() is called.

56.11. Postgres Extensions to the JDBC API

Postgres is an extensible database system. You can add your own functions to the backend, which can then be called from queries, or even add your own data types.

Now, as these are facilities unique to us, we support them from Java, with a set of extension API's. Some features within the core of the standard driver actually use these extensions to implement Large Objects, etc.

```
Accessing the extensions

To access some of the extensions, you need to use some extra methods
in the postgresql.Connection class. In this case, you would need to
case the return value of Driver.getConnection().

For example:

    Connection db = Driver.getConnection(url,user,pass);

    // later on
    Fastpath fp = ((postgresql.Connection)db).getFastpathAPI();

Class postgresql.Connection

java.lang.Object
   |
   +----postgresql.Connection

   public class Connection extends Object implements Connection

These are the extra methods used to gain access to our extensions. I
have not listed the methods defined by java.sql.Connection.

 public Fastpath getFastpathAPI() throws SQLException

          This returns the Fastpath API for the current connection.

          NOTE: This is not part of JDBC, but allows access to
functions on the postgresql backend itself.

          It is primarily used by the LargeObject API

          The best way to use this is as follows:

 import postgresql.fastpath.*;
 ...
 Fastpath fp = ((postgresql.Connection)myconn).getFastpathAPI();

          where myconn is an open Connection to postgresql.

        Returns:
                Fastpath object allowing access to functions on the
postgresql backend.

        Throws: SQLException
                by Fastpath when initialising for first time

 public LargeObjectManager getLargeObjectAPI() throws SQLException

          This returns the LargeObject API for the current connection.

          NOTE: This is not part of JDBC, but allows access to
functions on the postgresql backend itself.
```

```
          The best way to use this is as follows:

 import postgresql.largeobject.*;
 ...
 LargeObjectManager lo =
((postgresql.Connection)myconn).getLargeObjectAPI();

          where myconn is an open Connection to postgresql.

        Returns:
                LargeObject object that implements the API

        Throws: SQLException
                by LargeObject when initialising for first time

 public void addDataType(String type,
                         String name)

          This allows client code to add a handler for one of
postgresql's more unique data types. Normally, a data type not known
by the driver is returned by ResultSet.getObject() as a PGobject
instance.

This method allows you to write a class that extends PGobject, and
tell the driver the type name, and class name to use.

The down side to this, is that you must call this method each time a
connection is made.

          NOTE: This is not part of JDBC, but an extension.

          The best way to use this is as follows:

 ...
 ((postgresql.Connection)myconn).addDataType("mytype","my.class.name"-
);
 ...

          where myconn is an open Connection to postgresql.

          The handling class must extend postgresql.util.PGobject

        See Also:
                PGobject

Fastpath

Fastpath is an API that exists within the libpq C interface, and
allows a client machine to execute a function on the database
 backend.
Most client code will not need to use this method, but it's provided
because the Large Object API uses it.
```

```
To use, you need to import the postgresql.fastpath package, using the
line:
     import postgresql.fastpath.*;

Then, in your code, you need to get a FastPath object:
     Fastpath fp = ((postgresql.Connection)conn).getFastpathAPI();

This will return an instance associated with the database connection
that you can use to issue commands. The casing of Connection to
postgresql.Connection is required, as the getFastpathAPI() is one of
our own methods, not JDBC's.

Once you have a Fastpath instance, you can use the fastpath() methods
to execute a backend function.

Class postgresql.fastpath.Fastpath

java.lang.Object
   |
   +----postgresql.fastpath.Fastpath

   public class Fastpath

   extends Object

   This class implements the Fastpath api.

   This is a means of executing functions imbeded in the postgresql
backend from within a java application.

   It is based around the file src/interfaces/libpq/fe-exec.c

   See Also:
          FastpathFastpathArg, LargeObject

Methods

 public Object fastpath(int fnid,
                        boolean resulttype,
                        FastpathArg args[]) throws SQLException

          Send a function call to the PostgreSQL backend

        Parameters:
                fnid - Function id
                resulttype - True if the result is an integer, false
for
                other results
                args - FastpathArguments to pass to fastpath

        Returns:
                null if no data, Integer if an integer result, or
byte[]
                otherwise
```

```
        Throws: SQLException
                if a database-access error occurs.

 public Object fastpath(String name,
                        boolean resulttype,
                        FastpathArg args[]) throws SQLException

          Send a function call to the PostgreSQL backend by name.

Note:
          the mapping for the procedure name to function id needs to
exist, usually to an earlier call to addfunction(). This is the
prefered method to call, as function id's can/may change between
versions of the backend. For an example of how this works, refer to
postgresql.LargeObject

        Parameters:
                name - Function name
                resulttype - True if the result is an integer, false
for
                other results
                args - FastpathArguments to pass to fastpath

        Returns:
                null if no data, Integer if an integer result, or
byte[]
                otherwise

        Throws: SQLException
                if name is unknown or if a database-access error
occurs.

        See Also:
                LargeObject

 public int getInteger(String name,
                       FastpathArg args[]) throws SQLException

          This convenience method assumes that the return value is an
Integer

        Parameters:
                name - Function name
                args - Function arguments

        Returns:
                integer result

        Throws: SQLException
                if a database-access error occurs or no result

 public byte[] getData(String name,
                       FastpathArg args[]) throws SQLException
```

```
          This convenience method assumes that the return value is
binary data

        Parameters:
                name - Function name
                args - Function arguments

        Returns:
                byte[] array containing result

        Throws: SQLException
                if a database-access error occurs or no result

 public void addFunction(String name,
                         int fnid)

          This adds a function to our lookup table.

          User code should use the addFunctions method, which is
 based
upon a query, rather than hard coding the oid. The oid for a function
is not guaranteed to remain static, even on different servers of the
same version.

        Parameters:
                name - Function name
                fnid - Function id

 public void addFunctions(ResultSet rs) throws SQLException

          This takes a ResultSet containing two columns. Column 1
contains the function name, Column 2 the oid.

          It reads the entire ResultSet, loading the values into the
function table.

          REMEMBER to close() the resultset after calling this!!

          Implementation note about function name lookups:

          PostgreSQL stores the function id's and their corresponding
names in the pg_proc table. To speed things up locally, instead of
querying each function from that table when required, a Hashtable is
used. Also, only the function's required are entered into this table,
keeping connection times as fast as possible.

          The postgresql.LargeObject class performs a query upon it's
startup, and passes the returned ResultSet to the addFunctions()
method here.

          Once this has been done, the LargeObject api refers to the
functions by name.
```

```
          Dont think that manually converting them to the oid's will
work. Ok, they will for now, but they can change during development
(there was some discussion about this for V7.0), so this is
implemented to prevent any unwarranted headaches in the future.

        Parameters:
                rs - ResultSet

        Throws: SQLException
                if a database-access error occurs.

        See Also:
                LargeObjectManager

 public int getID(String name) throws SQLException

          This returns the function id associated by its name

          If addFunction() or addFunctions() have not been called for
this name, then an SQLException is thrown.

        Parameters:
                name - Function name to lookup

        Returns:
                Function ID for fastpath call

        Throws: SQLException
                is function is unknown.

Class postgresql.fastpath.FastpathArg

java.lang.Object
   |
   +----postgresql.fastpath.FastpathArg

   public class FastpathArg extends Object

   Each fastpath call requires an array of arguments, the number and
type dependent on the function being called.

   This class implements methods needed to provide this capability.

   For an example on how to use this, refer to the
postgresql.largeobject package

   See Also:
          Fastpath, LargeObjectManager, LargeObject

Constructors

 public FastpathArg(int value)

          Constructs an argument that consists of an integer value
```

```
        Parameters:
                value - int value to set

 public FastpathArg(byte bytes[])

          Constructs an argument that consists of an array of bytes

        Parameters:
                bytes - array to store

 public FastpathArg(byte buf[],
                    int off,
                    int len)

           Constructs an argument that consists of part of a byte
array

        Parameters:
                buf - source array
                off - offset within array
                len - length of data to include

 public FastpathArg(String s)

          Constructs an argument that consists of a String.

        Parameters:
                s - String to store
```

Geometric Data Types

PostgreSQL has a set of datatypes that can store geometric features into a table. These range from single points, lines, and polygons.

We support these types in Java with the postgresql.geometric package.

It contains classes that extend the postgresql.util.PGobject class. Refer to that class for details on how to implement your own data type handlers.

Class postgresql.geometric.PGbox

```
java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGbox
```

   public class PGbox extends PGobject implements Serializable, Cloneable

   This represents the box datatype within postgresql.

```
Variables

 public PGpoint point[]

          These are the two corner points of the box.

Constructors

 public PGbox(double x1,
              double y1,
              double x2,
              double y2)

        Parameters:
                x1 - first x coordinate
                y1 - first y coordinate
                x2 - second x coordinate
                y2 - second y coordinate

 public PGbox(PGpoint p1,
              PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGbox(String s) throws SQLException

        Parameters:
                s - Box definition in PostgreSQL syntax

        Throws: SQLException
                if definition is invalid

 public PGbox()

          Required constructor

Methods

 public void setValue(String value) throws SQLException

          This method sets the value of this object. It should be
overidden, but still called by subclasses.

        Parameters:
                value - a string representation of the value of the
object
        Throws: SQLException
                thrown if value is invalid for this type

        Overrides:
                setValue in class PGobject
```

```
 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGbox in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGcircle

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGcircle

   public class PGcircle extends PGobject implements Serializable,
Cloneable

   This represents postgresql's circle datatype, consisting of a
 point
and a radius

Variables

 public PGpoint center

          This is the centre point

public double radius

          This is the radius

Constructors
```

```
public PGcircle(double x,
                double y,
                double r)

       Parameters:
              x - coordinate of centre
               y - coordinate of centre
               r - radius of circle

public PGcircle(PGpoint c,
                double r)

       Parameters:
               c - PGpoint describing the circle's centre
               r - radius of circle

public PGcircle(String s) throws SQLException

       Parameters:
               s - definition of the circle in PostgreSQL's syntax.

       Throws: SQLException
               on conversion failure

public PGcircle()

         This constructor is used by the driver.

Methods

public void setValue(String s) throws SQLException

       Parameters:
               s - definition of the circle in PostgreSQL's syntax.

       Throws: SQLException
               on conversion failure

       Overrides:
               setValue in class PGobject

public boolean equals(Object obj)

       Parameters:
               obj - Object to compare with

       Returns:
               true if the two boxes are identical

       Overrides:
               equals in class PGobject

public Object clone()
```

```
          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGcircle in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGline

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGline

   public class PGline extends PGobject implements Serializable,
Cloneable

   This implements a line consisting of two points. Currently line is
not yet implemented in the backend, but this class ensures that when
it's done were ready for it.

Variables

 public PGpoint point[]

          These are the two points.

Constructors

 public PGline(double x1,
               double y1,
               double x2,
               double y2)

        Parameters:
                x1 - coordinate for first point
                y1 - coordinate for first point
                x2 - coordinate for second point
                y2 - coordinate for second point

 public PGline(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point
```

```
 public PGline(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

 public PGline()

          reuired by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the line segment in PostgreSQL's
syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGline in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGlseg
```

```
java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGlseg

   public class PGlseg extends PGobject implements Serializable,
Cloneable

   This implements a lseg (line segment) consisting of two points

Variables

 public PGpoint point[]

          These are the two points.

Constructors

 public PGlseg(double x1,
               double y1,
               double x2,
               double y2)

        Parameters:

                x1 - coordinate for first point
                y1 - coordinate for first point
                x2 - coordinate for second point
                y2 - coordinate for second point

 public PGlseg(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGlseg(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

 public PGlseg()

          reuired by the driver

Methods

 public void setValue(String s) throws SQLException
```

```
        Parameters:
                s - Definition of the line segment in PostgreSQL's
syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
               clone in class PGobject

 public String getValue()

        Returns:
                the PGlseg in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGpath

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpath

   public class PGpath extends PGobject implements Serializable,
Cloneable

   This implements a path (a multiple segmented line, which may be
closed)

Variables

 public boolean open
```

```
          True if the path is open, false if closed

 public PGpoint points[]

          The points defining this path

Constructors

 public PGpath(PGpoint points[],
               boolean open)

        Parameters:
                points - the PGpoints that define the path
                open - True if the path is open, false if closed

 public PGpath()

          Required by the driver

 public PGpath(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the path in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned
```

```
        Overrides:
                clone in class PGobject

 public String getValue()

          This returns the polygon in the syntax expected by
postgresql

        Overrides:
                getValue in class PGobject

 public boolean isOpen()

     This returns true if the path is open

 public boolean isClosed()

     This returns true if the path is closed

 public void closePath()

     Marks the path as closed

 public void openPath()

     Marks the path as open

Class postgresql.geometric.PGpoint

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpoint

   public class PGpoint extends PGobject implements Serializable,
Cloneable

   This implements a version of java.awt.Point, except it uses double
to represent the coordinates.

   It maps to the point datatype in postgresql.

Variables

 public double x

          The X coordinate of the point

 public double y

          The Y coordinate of the point
```

```
Constructors

 public PGpoint(double x,
                double y)

        Parameters:
                x - coordinate
                y - coordinate

 public PGpoint(String value) throws SQLException

          This is called mainly from the other geometric types, when
 a
point is imbeded within their definition.

        Parameters:
                value - Definition of this point in PostgreSQL's
syntax

 public PGpoint()

          Required by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of this point in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject
```

```
public String getValue()

        Returns:
                the PGpoint in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

public void translate(int x,
                      int y)

          Translate the point with the supplied amount.

        Parameters:
                x - integer amount to add on the x axis
                y - integer amount to add on the y axis

public void translate(double x,
                      double y)

          Translate the point with the supplied amount.

        Parameters:
                x - double amount to add on the x axis
                y - double amount to add on the y axis

public void move(int x,
                 int y)

          Moves the point to the supplied coordinates.

        Parameters:
                x - integer coordinate
                y - integer coordinate

public void move(double x,
                 double y)

          Moves the point to the supplied coordinates.

        Parameters:
                x - double coordinate
                y - double coordinate

public void setLocation(int x,
                        int y)

          Moves the point to the supplied coordinates. refer to
          java.awt.Point for description of this

        Parameters:
                x - integer coordinate
                y - integer coordinate
```

```
        See Also:
                Point

 public void setLocation(Point p)

          Moves the point to the supplied java.awt.Point refer to
          java.awt.Point for description of this

        Parameters:
                p - Point to move to

        See Also:
                Point

Class postgresql.geometric.PGpolygon

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpolygon

   public class PGpolygon extends PGobject implements Serializable,
Cloneable

   This implements the polygon datatype within PostgreSQL.

Variables

 public PGpoint points[]

          The points defining the polygon

Constructors

 public PGpolygon(PGpoint points[])

          Creates a polygon using an array of PGpoints

        Parameters:
                points - the points defining the polygon

 public PGpolygon(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's syntax.

        Throws: SQLException
                on conversion failure

 public PGpolygon()

          Required by the driver
```

```
Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the polygon in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGpolygon in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Large Objects

Large objects are supported in the standard JDBC specification.
However, that interface is limited, and the api provided by
 PostgreSQL
allows for random access to the objects contents, as if it was a
 local
file.

The postgresql.largeobject package profides to Java the libpq C
interface's large object API. It consists of two classes,
LargeObjectManager, which deals with creating, opening and deleting
large obejects, and LargeObject which deals with an individual object.

Class postgresql.largeobject.LargeObject
```

```
java.lang.Object
   |
   +----postgresql.largeobject.LargeObject

public class LargeObject extends Object

This class implements the large object interface to postgresql.

   It provides the basic methods required to run the interface, plus
 a
pair of methods that provide InputStream and OutputStream classes for
this object.

   Normally, client code would use the getAsciiStream,
getBinaryStream, or getUnicodeStream methods in ResultSet, or
setAsciiStream, setBinaryStream, or setUnicodeStream methods in
PreparedStatement to access Large Objects.

   However, sometimes lower level access to Large Objects are
required, that are not supported by the JDBC specification.

   Refer to postgresql.largeobject.LargeObjectManager on how to gain
access to a Large Object, or how to create one.

   See Also:
          LargeObjectManager

Variables

 public static final int SEEK_SET

          Indicates a seek from the begining of a file

 public static final int SEEK_CUR

          Indicates a seek from the current position

 public static final int SEEK_END

          Indicates a seek from the end of a file

Methods

 public int getOID()

        Returns:
                the OID of this LargeObject

 public void close() throws SQLException

          This method closes the object. You must not call methods in
this object after this is called.
```

```
   Throws: SQLException
               if a database-access error occurs.

 public byte[] read(int len) throws SQLException

          Reads some data from the object, and return as a byte[]
array

        Parameters:
                len - number of bytes to read

        Returns:
                byte[] array containing data read

        Throws: SQLException
                if a database-access error occurs.

 public void read(byte buf[],
                  int off,
                  int len) throws SQLException

          Reads some data from the object into an existing array

        Parameters:
                buf - destination array
                off - offset within array
                len - number of bytes to read

        Throws: SQLException
                if a database-access error occurs.

 public void write(byte buf[]) throws SQLException

          Writes an array to the object


        Parameters:
                buf - array to write

        Throws: SQLException
                if a database-access error occurs.

 public void write(byte buf[],
                   int off,
                   int len) throws SQLException

          Writes some data from an array to the object

        Parameters:
                buf - destination array
                off - offset within array
                len - number of bytes to write

        Throws: SQLException
```

```
                if a database-access error occurs.

 public void seek(int pos,
                  int ref) throws SQLException

          Sets the current position within the object.

          This is similar to the fseek() call in the standard C
library.It allows you to have random access to the large object.

        Parameters:
                pos - position within object
                ref - Either SEEK_SET, SEEK_CUR or SEEK_END
        Throws: SQLException
                if a database-access error occurs.

 public void seek(int pos) throws SQLException

          Sets the current position within the object.

          This is similar to the fseek() call in the standard C
library.It allows you to have random access to the large object.

        Parameters:
                pos - position within object from begining

        Throws: SQLException
                if a database-access error occurs.

 public int tell() throws SQLException

        Returns:
                the current position within the object

        Throws: SQLException
                if a database-access error occurs.

 public int size() throws SQLException

          This method is inefficient, as the only way to find out the
size of the object is to seek to the end, record the current
 position,
then return to the original position.

          A better method will be found in the future.

        Returns:
                the size of the large object

        Throws: SQLException
                if a database-access error occurs.

 public InputStream getInputStream() throws SQLException
```

```
          Returns an InputStream from this object.

          This InputStream can then be used in any method that
requires an InputStream.

        Throws: SQLException
                if a database-access error occurs.

 public OutputStream getOutputStream() throws SQLException

          Returns an OutputStream to this object

          This OutputStream can then be used in any method that
requires an OutputStream.

        Throws: SQLException
                if a database-access error occurs.

Class postgresql.largeobject.LargeObjectManager

java.lang.Object
   |
   +----postgresql.largeobject.LargeObjectManager

public class LargeObjectManager extends Object

This class implements the large object interface to postgresql.

   It provides methods that allow client code to create, open and
delete large objects from the database. When opening an object, an
instance of postgresql.largeobject.LargeObject is returned, and its
methods then allow access to the object.

This class can only be created by postgresql.Connection

To get access to this class, use the following segment of code:

 import postgresql.largeobject.*;
 Connection  conn;
 LargeObjectManager lobj;
 ... code that opens a connection ...
 lobj = ((postgresql.Connection)myconn).getLargeObjectAPI();

Normally, client code would use the getAsciiStream, getBinaryStream,
or getUnicodeStream methods in ResultSet, or setAsciiStream,
setBinaryStream, or setUnicodeStream methods in PreparedStatement to
access Large Objects.

   However, sometimes lower level access to Large Objects are
required, that are not supported by the JDBC specification.

   Refer to postgresql.largeobject.LargeObject on how to manipulate
the contents of a Large Object.
```

```
   See Also:
          LargeObject

Variables

 public static final int WRITE

          This mode indicates we want to write to an object

 public static final int READ

          This mode indicates we want to read an object

 public static final int READWRITE

          This mode is the default. It indicates we want read and
write access to a large object

Methods

 public LargeObject open(int oid) throws SQLException

          This opens an existing large object, based on its OID. This
          method assumes that READ and WRITE access is required (the
default).

        Parameters:
                oid - of large object

        Returns:
                LargeObject instance providing access to the object

        Throws: SQLException
                on error

 public LargeObject open(int oid,
                         int mode) throws SQLException

          This opens an existing large object, based on its OID

        Parameters:
                oid - of large object
                mode - mode of open

        Returns:
                LargeObject instance providing access to the object

        Throws: SQLException
                on error

 public int create() throws SQLException

          This creates a large object, returning its OID.
```

```
          It defaults to READWRITE for the new object's attributes.

        Returns:
                oid of new object

        Throws: SQLException
                on error

 public int create(int mode) throws SQLException

          This creates a large object, returning its OID

        Parameters:
                mode - a bitmask describing different attributes of
the
                new object

        Returns:
                oid of new object

        Throws: SQLException
                on error

 public void delete(int oid) throws SQLException

          This deletes a large object.

        Parameters:
                oid - describing object to delete

        Throws: SQLException
                on error

 public void unlink(int oid) throws SQLException

          This deletes a large object.

          It is identical to the delete method, and is supplied as
 the
C API uses unlink.

        Parameters:
                oid - describing object to delete

        Throws: SQLException
                on error
```

Object Serialisation

PostgreSQL is not a normal SQL Database. It is far more extensible than most other databases, and does support Object Oriented features that are unique to it.

One of the consequences of this, is that you can have one table refer to a row in another table. For example:

```
test=> create table users (username name,fullname text);
CREATE
test=> create table server (servername name,adminuser users);
CREATE
test=> insert into users values ('peter','Peter Mount');
INSERT 2610132 1
test=> insert into server values ('maidast',2610132::users);
INSERT 2610133 1
test=> select * from users;
username|fullname
--------+--------------
peter   |Peter Mount
(1 row)

test=> select * from server;
servername|adminuser
----------+---------
maidast   |  2610132
(1 row)
```

Ok, the above example shows that we can use a table name as a field, and the row's oid value is stored in that field.

What does this have to do with Java?

In Java, you can store an object to a Stream as long as it's class implements the java.io.Serializable interface. This process, known as Object Serialization, can be used to store complex objects into the database.

Now, under JDBC, you would have to use a LargeObject to store them. However, you cannot perform queries on those objects.

What the postgresql.util.Serialize class does, is provide a means of storing an object as a table, and to retrieve that object from a table. In most cases, you would not need to access this class direct, but you would use the PreparedStatement.setObject() and ResultSet.getObject() methods. Those methods will check the objects class name against the table's in the database. If a match is found, it assumes that the object is a Serialized object, and retrieves it from that table. As it does so, if the object contains other serialized objects, then it recurses down the tree.

Sound's complicated? In fact, it's simpler than what I wrote - it's just difficult to explain.

The only time you would access this class, is to use the create() methods. These are not used by the driver, but issue one or more "create table" statements to the database, based on a Java Object or Class that you want to serialize.

Oh, one last thing. If your object contains a line like:

```
     public int oid;
```

then, when the object is retrieved from the table, it is set to the oid within the table. Then, if the object is modified, and re-serialized, the existing entry is updated.

If the oid variable is not present, then when the object is serialized, it is always inserted into the table, and any existing entry in the table is preserved.

Setting oid to 0 before serialization, will also cause the object to be inserted. This enables an object to be duplicated in the database.

Class postgresql.util.Serialize

```
java.lang.Object
   |
   +----postgresql.util.Serialize
```

   public class Serialize extends Object

   This class uses PostgreSQL's object oriented features to store Java Objects. It does this by mapping a Java Class name to a table in the database. Each entry in this new table then represents a Serialized instance of this class. As each entry has an OID (Object IDentifier), this OID can be included in another table. This is too complex to show here, and will be documented in the main documents in more detail.

Constructors

```
 public Serialize(Connection c,
                  String type) throws SQLException
```

          This creates an instance that can be used to serialize ordeserialize a Java object from a PostgreSQL table.

Methods

```
 public Object fetch(int oid) throws SQLException
```

          This fetches an object from a table, given it's OID

        Parameters:
                oid - The oid of the object

        Returns:
                Object relating to oid

        Throws: SQLException
                on error

```
 public int store(Object o) throws SQLException
```

```
          This stores an object into a table, returning it's OID.

          If the object has an int called OID, and it is > 0, then
that value is used for the OID, and the table will be updated. If the
value of OID is 0, then a new row will be created, and the value of
OID will be set in the object. This enables an object's value in the
database to be updateable. If the object has no int called OID, then
the object is stored. However if the object is later retrieved,
amended and stored again, it's new state will be appended to the
table, and will not overwrite the old entries.

        Parameters:
                o - Object to store (must implement Serializable)

        Returns:
                oid of stored object

        Throws: SQLException
                on error

 public static void create(Connection con,
                           Object o) throws SQLException

          This method is not used by the driver, but it creates a
table, given a Serializable Java Object. It should be used before
serializing any objects.

        Parameters:
                c - Connection to database
                o - Object to base table on

        Throws: SQLException
                on error

                   Returns:
                Object relating to oid

        Throws: SQLException
                on error

 public int store(Object o) throws SQLException

          This stores an object into a table, returning it's OID.

          If the object has an int called OID, and it is > 0, then
that value is used for the OID, and the table will be updated. If the
value of OID is 0, then a new row will be created, and the value of
OID will be set in the object. This enables an object's value in the
database to be updateable. If the object has no int called OID, then
the object is stored. However if the object is later retrieved,
amended and stored again, it's new state will be appended to the
table, and will not overwrite the old entries.
```

```
        Parameters:
                o - Object to store (must implement Serializable)

        Returns:
                oid of stored object

        Throws: SQLException
                on error

 public static void create(Connection con,
                           Object o) throws SQLException

          This method is not used by the driver, but it creates a
table, given a Serializable Java Object. It should be used before
serializing any objects.

        Parameters:
                c - Connection to database
                o - Object to base table on

        Throws: SQLException
                on error

 public static void create(Connection con,
                           Class c) throws SQLException

          This method is not used by the driver, but it creates a
table, given a Serializable Java Object. It should be used before
serializing any objects.

        Parameters:
                c - Connection to database
                o - Class to base table on

        Throws: SQLException
                on error

 public static String toPostgreSQL(String name) throws SQLException

          This converts a Java Class name to a postgresql table, by
          replacing . with _

          Because of this, a Class name may not have _ in the name.

          Another limitation, is that the entire class name (including
          packages) cannot be longer than 31 characters (a limit
forced by PostgreSQL).

        Parameters:
                name - Class name

        Returns:
                PostgreSQL table name
```

```
        Throws: SQLException
                on error

 public static String toClassName(String name) throws SQLException

          This converts a postgresql table to a Java Class name, by
          replacing _ with .

        Parameters:
                name - PostgreSQL table name

        Returns:
                Class name

        Throws: SQLException
                on error
```

Utility Classes

The postgresql.util package contains classes used by the internals of the main driver, and the other extensions.

Class postgresql.util.PGmoney

```
java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.util.PGmoney
```

   public class PGmoney extends PGobject implements Serializable, Cloneable

   This implements a class that handles the PostgreSQL money type

Variables

```
 public double val

          The value of the field
```

Constructors

```
 public PGmoney(double value)

        Parameters:
                value - of field

 public PGmoney(String value) throws SQLException

          This is called mainly from the other geometric types, when
 a
point is imbeded within their definition.
```

```
        Parameters:
                value - Definition of this point in PostgreSQL's
syntax

 public PGmoney()

          Required by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of this point in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGpoint in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.util.PGobject

java.lang.Object
   |
   +----postgresql.util.PGobject

   public class PGobject extends Object implements Serializable,
```

```
Cloneable

   This class is used to describe data types that are unknown by JDBC
Standard.
    A call to postgresql.Connection permits a class that extends this
class to be associated with a named type. This is how the
postgresql.geometric package operates.
    ResultSet.getObject() will return this class for any type that is
not recognised on having it's own handler. Because of this, any
postgresql data type is supported.

Constructors

 public PGobject()

          This is called by postgresql.Connection.getObject() to
create the object.

Methods

 public final void setType(String type)

          This method sets the type of this object.

          It should not be extended by subclasses, hence its final

        Parameters:
                type - a string describing the type of the object

 public void setValue(String value) throws SQLException

          This method sets the value of this object. It must be
overidden.

        Parameters:
                value - a string representation of the value of the
                object

        Throws: SQLException
                thrown if value is invalid for this type

 public final String getType()

          As this cannot change during the life of the object, it's
final.

        Returns:
                the type name of this object

 public String getValue()

          This must be overidden, to return the value of the object,
in the form required by postgresql.
```

```
        Returns:
                the value of this object

 public boolean equals(Object obj)

          This must be overidden to allow comparisons of objects

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class Object

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class Object

 public String toString()

          This is defined here, so user code need not overide it.

        Returns:
                the value of this object, in the syntax expected by
postgresql

        Overrides:
                toString in class Object

Class postgresql.util.PGtokenizer

java.lang.Object
   |
   +----postgresql.util.PGtokenizer

   public class PGtokenizer extends Object

   This class is used to tokenize the text output of postgres.

   We could have used StringTokenizer to do this, however, we needed
to handle nesting of '(' ')' '[' ']' '<' and '>' as these are used by
the geometric data types.

   It's mainly used by the geometric classes, but is useful in
 parsing
any output from custom data types output from postgresql.

   See Also:
          PGbox, PGcircle, PGlseg, PGpath, PGpoint, PGpolygon
```

```
Constructors

 public PGtokenizer(String string,
                    char delim)

          Create a tokeniser.

        Parameters:
                string - containing tokens
                delim - single character to split the tokens

Methods

 public int tokenize(String string,
                     char delim)

          This resets this tokenizer with a new string and/or
delimiter.

        Parameters:
                string - containing tokens
                delim - single character to split the tokens

 public int getSize()

        Returns:
                the number of tokens available

 public String getToken(int n)

        Parameters:
                n - Token number ( 0 ... getSize()-1 )

        Returns:
                The token value

 public PGtokenizer tokenizeToken(int n,
                                  char delim)

          This returns a new tokenizer based on one of our tokens.
 The
geometric datatypes use this to process nested tokens (usually
PGpoint).

        Parameters:
                n - Token number ( 0 ... getSize()-1 )
                delim - The delimiter to use

        Returns:
                A new instance of PGtokenizer based on the token

 public static String remove(String s,
                             String l,
```

```
                                String t)

          This removes the lead/trailing strings from a string

        Parameters:
                s - Source string
                l - Leading string to remove
                t - Trailing string to remove

        Returns:
                String without the lead/trailing strings

public void remove(String l,
                   String t)

          This removes the lead/trailing strings from all tokens

        Parameters:
                l - Leading string to remove
                t - Trailing string to remove

public static String removePara(String s)

          Removes ( and ) from the beginning and end of a string

        Parameters:
                s - String to remove from

        Returns:
                String without the ( or )

public void removePara()

          Removes ( and ) from the beginning and end of all tokens

        Returns:
                String without the ( or )

public static String removeBox(String s)

          Removes [ and ] from the beginning and end of a string

        Parameters:
                s - String to remove from

        Returns:
                String without the [ or ]

public void removeBox()

          Removes [ and ] from the beginning and end of all tokens

        Returns:
                String without the [ or ]
```

```
 public static String removeAngle(String s)

          Removes < and > from the beginning and end of a string

        Parameters:
                s - String to remove from

        Returns:
                String without the < or >

 public void removeAngle()

          Removes < and > from the beginning and end of all tokens

        Returns:
                String without the < or >
```

Class postgresql.util.Serialize

This was documented earlier under Object Serialisation.

Class postgresql.util.UnixCrypt

```
java.lang.Object
   |
   +----postgresql.util.UnixCrypt

   public class UnixCrypt extends Object

   This class provides us with the ability to encrypt passwords when
sent over the network stream

   Contains static methods to encrypt and compare passwords with Unix
encrypted passwords.

   See John Dumas's Java Crypt page for the original source.

   http://www.zeh.com/local/jfd/crypt.html
```

Methods

```
 public static final String crypt(String salt,
                                  String original)

          Encrypt a password given the cleartext password and a
"salt".

        Parameters:
                salt - A two-character string representing the salt
used
                to iterate the encryption engine in lots of different
                ways. If you are generating a new encryption then this
                value should be randomised.
```

```
                original - The password to be encrypted.

        Returns:
                A string consisting of the 2-character salt followed
by
                the encrypted password.

 public static final String crypt(String original)

          Encrypt a password given the cleartext password. This
 method
generates a random salt using the 'java.util.Random' class.

        Parameters:
                original - The password to be encrypted.

        Returns:
                A string consisting of the 2-character salt followed
by
                the encrypted password.

 public static final boolean matches(String encryptedPassword,
                                     String enteredPassword)

          Check that enteredPassword encrypts to encryptedPassword.

        Parameters:
                encryptedPassword - The encryptedPassword. The first
two characters are assumed to be the salt. This string would be the
same as one found in a Unix /etc/passwd file.
                enteredPassword - The password as entered by the user
(or otherwise aquired).

        Returns:
                true if the password should be considered correct.
```

Using the driver in a multi Threaded or Servlet environment

A problem with many JDBC drivers, is that only one thread can use a Connection at any one time - otherwise a thread could send a query while another one is receiving results, and this would be a bad thing for the database engine.

PostgreSQL 6.4, brings thread safety to the entire driver. Standard JDBC was thread safe in 6.3.x, but the Fastpath API wasn't.

So, if your application uses multiple threads (which most decent ones would), then you don't have to worry about complex schemes to ensure only one uses the database at any time.

If a thread attempts to use the connection while another is using it, it will wait until the other thread has finished it's current operation.

```
If it's a standard SQL statement, then the operation is sending the
statement, and retrieving any ResultSet (in full).

If it's a Fastpath call (ie: reading a block from a LargeObject),
 then
it's the time to send, and retrieve that block.

This is fine for applications & applets, but can cause a performance
problem with servlets.

With servlets, you can have a heavy load on the connection. If you
have several threads performing queries, then each one will pause,
which may not be what you are after.

To solve this, you would be advised to create a pool of Connections.

When ever a thread needs to use the database, it asks a manager class
for a Connection. It hands a free connection to the thread, and marks
it as busy. If a free connection is not available, it opens one.

Once the thread has finished with it, it returns it to the manager,
who can then either close it, or add it to the pool. The manager
 would
also check that the connection is still alive, and remove it from the
pool if it's dead.

So, with servlets, it's up to you to use either a single connection,
or a pool. The plus side for a pool is that threads will not be hit
 by
the bottle neck caused by a single network connection. The down side,
is that it increases the load on the server, as a backend is created
for each Connection.

It's up to you, and your applications requirements.
```

56.12. Further Reading

If you have not yet read it, I'd advise you read the JDBC API Documentation (supplied with Sun's JDK), and the JDBC Specification. Both are available on JavaSoft's web site[2].

My own web site[3] contains updated information not included in this document, and also includes precompiled drivers for v6.4, and earlier.

[2] http://www.javasoft.com
[3] http://www.retep.org.uk

Part VI. Developer's Guide

The Developer's Guide includes discussion of design decisions and suggestions for future development.

Table of Contents

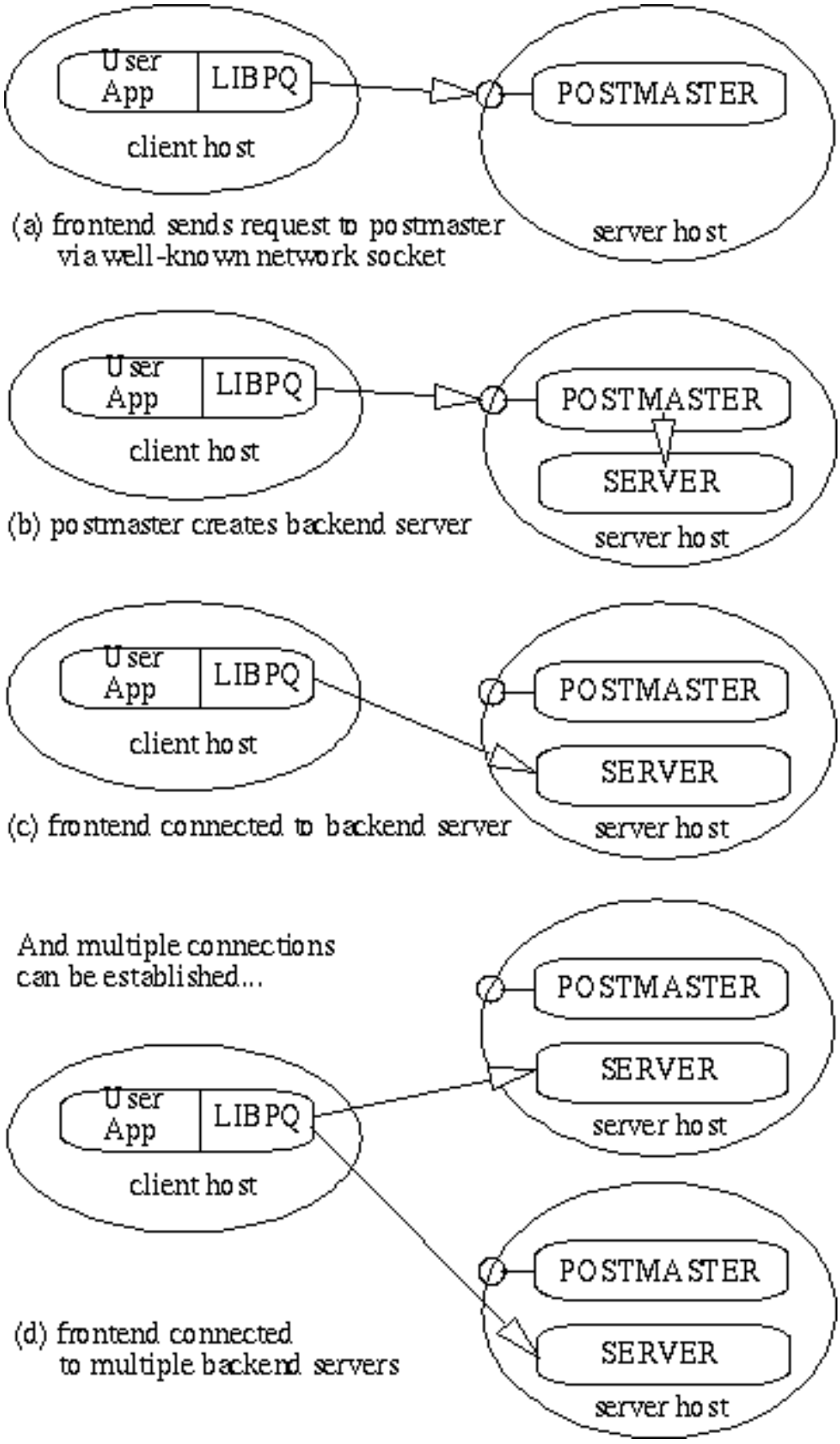
Chapter 57. Architecture

57.1. Postgres Architectural Concepts

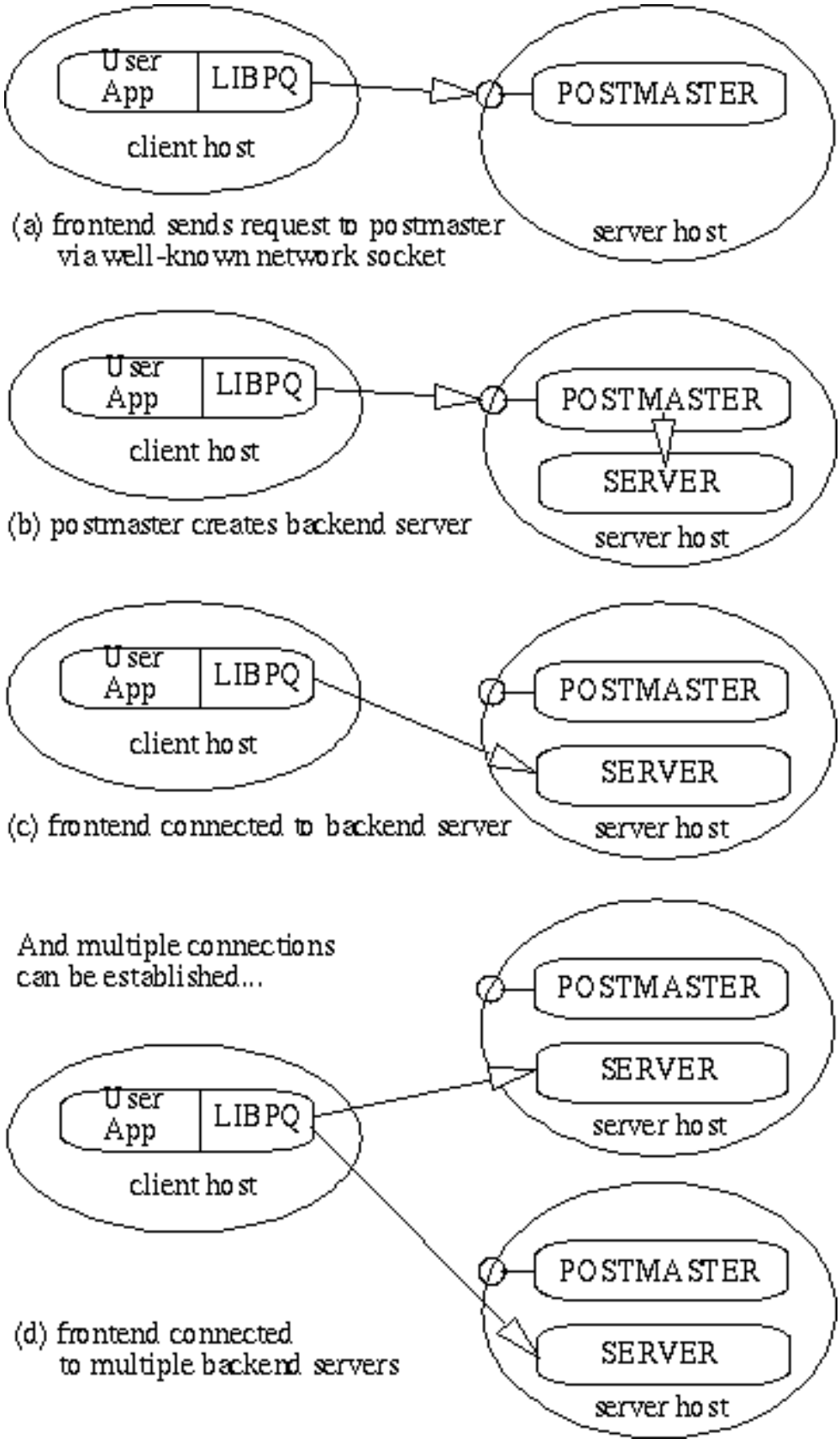
Before we continue, you should understand the basic Postgres system architecture. Understanding how the parts of Postgres interact will make the next chapter somewhat clearer. In database jargon, Postgres uses a simple "process per-user" client/server model. A Postgres session consists of the following cooperating UNIX processes (programs):

- A supervisory daemon process (postmaster),
- the user's frontend application (e.g., the psql program), and
- the one or more backend database servers (the postgres process itself).

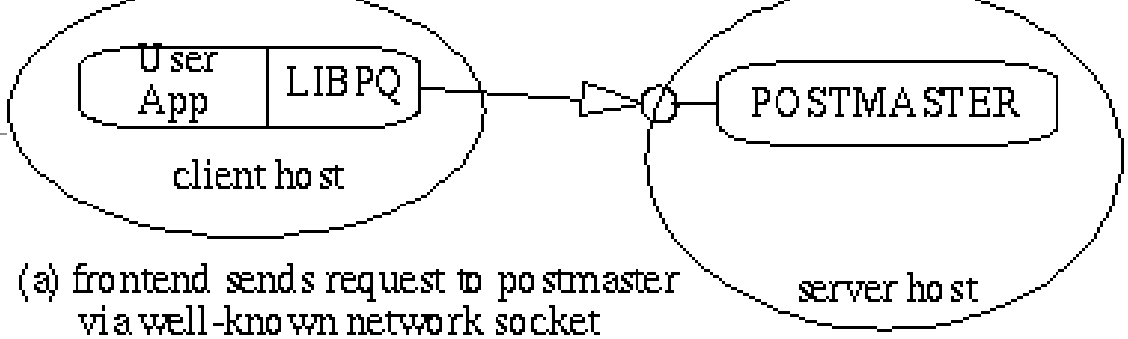
A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called an installation or site. Frontend applications that wish to access a given database within an installation make calls to the library. The library sends user requests over the network to the postmaster (


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

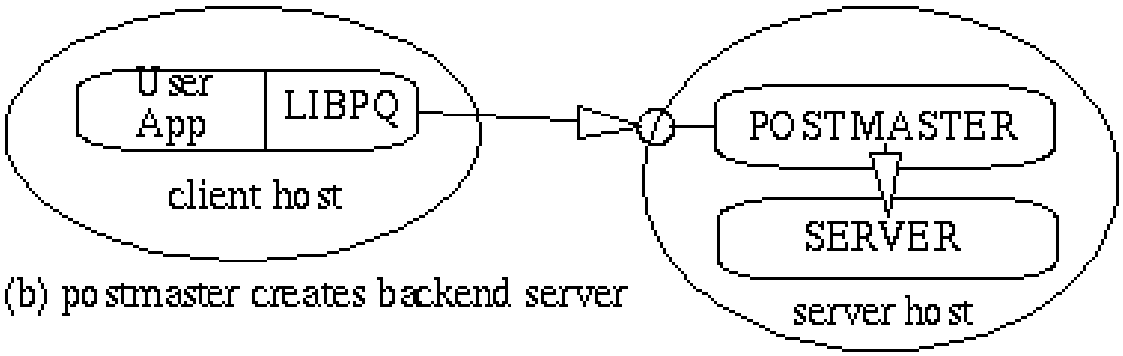
(a)), which in turn starts a new backend server process (


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

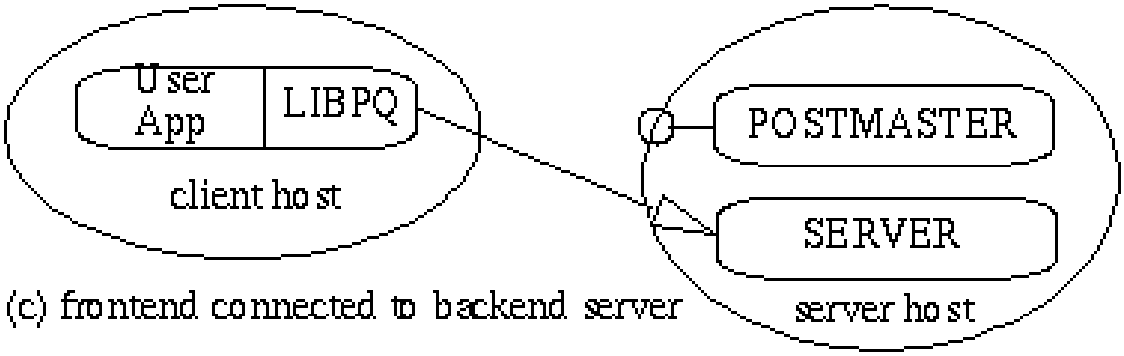
(b))



(a) frontend sends request to postmaster via well-known network socket

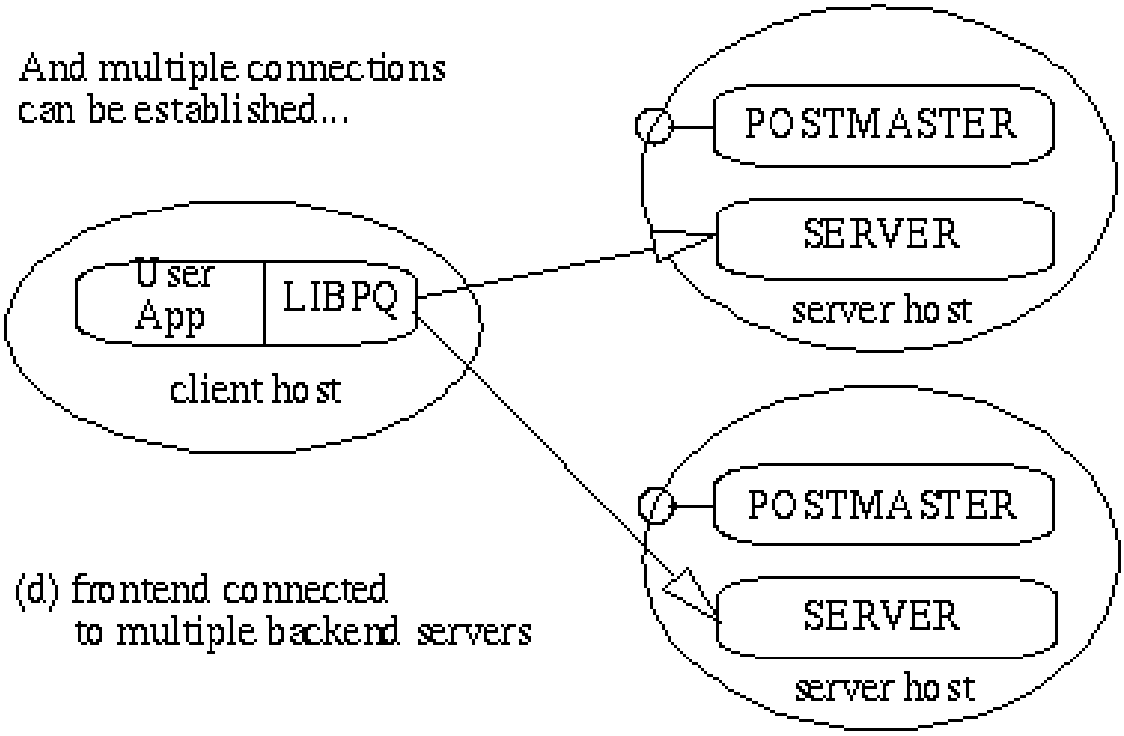


(b) postmaster creates backend server



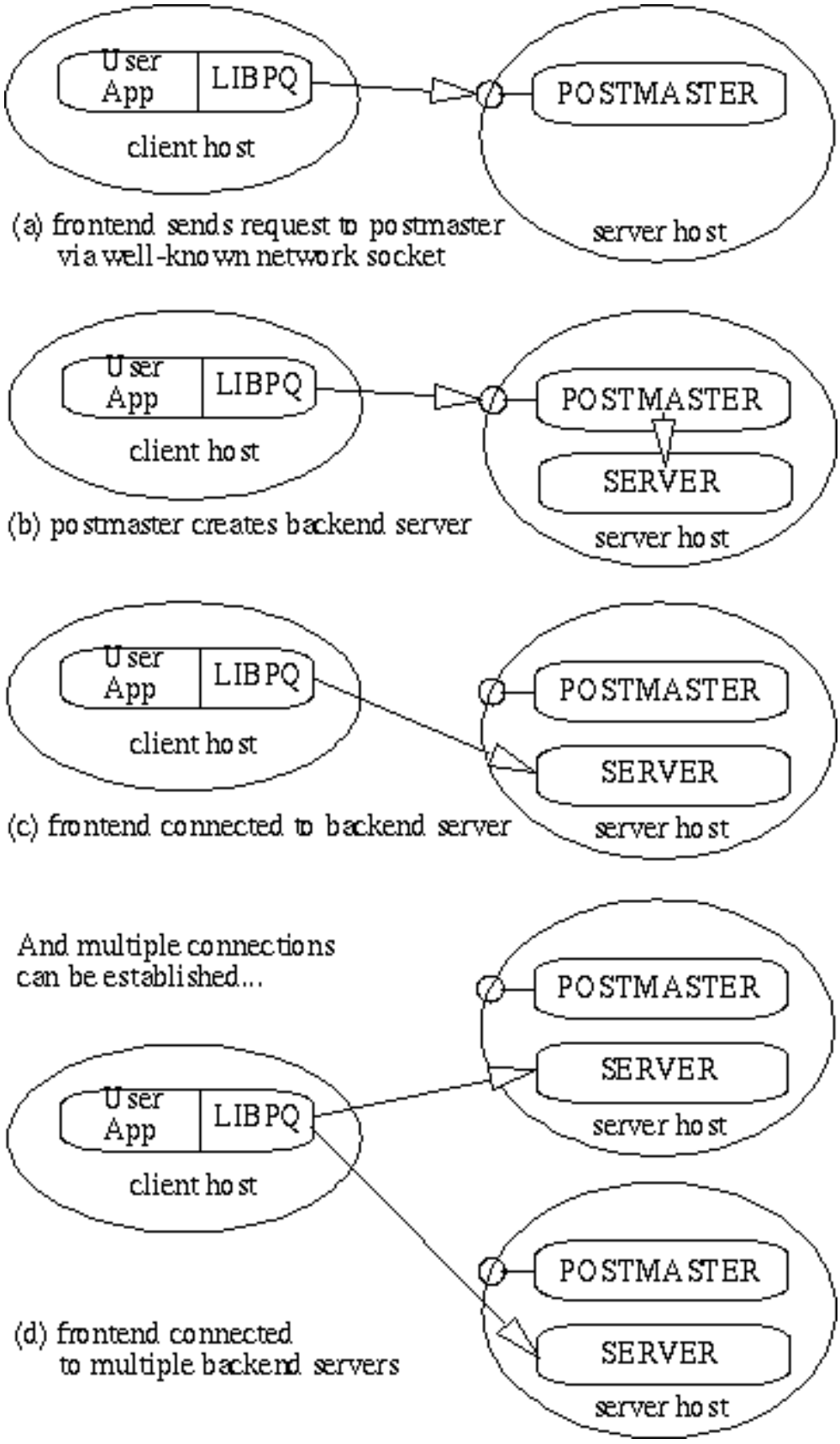
(c) frontend connected to backend server

And multiple connections can be established...



(d) frontend connected to multiple backend servers

and connects the frontend process to the new server (


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

(c)). From that point on, the frontend process and the backend server communicate without intervention by the postmaster. Hence, the postmaster is always running, waiting for requests, whereas frontend and backend processes come and go. The `libpq` library allows a single frontend to make multiple connections to backend processes. However, the frontend application is still a single-threaded process. Multithreaded frontend/backend connections are not currently supported in `libpq`. One implication of this architecture is that the postmaster and the backend always run on the same machine (the database server), while the frontend application may run anywhere. You should keep this in mind, because the files that can be accessed on a client machine may not be accessible (or may only be accessed using a different filename) on the database server machine. You should also be aware that the postmaster and postgres servers run with the user-id of the Postgres "superuser." Note that the Postgres superuser does not have to be a special user (e.g., a user named "postgres"). Furthermore, the Postgres superuser should definitely not be the UNIX superuser, "root"! In any case, all files relating to a database should belong to this Postgres superuser.

Chapter 58. Genetic Query Optimization in Database Systems

Martin Utesch, University of Mining and Technology

Author

Written by Martin Utesch[1] for the Institute of Automatic Control at the University of Mining and Technology in Freiberg, Germany.

58.1. Query Handling as a Complex Optimization Problem

Among all relational operators the most difficult one to process and optimize is the *join*. The number of alternative plans to answer a query grows exponentially with the number of `joins` included in it. Further optimization effort is caused by the support of a variety of *join methods* (e.g., nested loop, index scan, merge join in Postgres) to process individual `joins` and a diversity of *indices* (e.g., r-tree, b-tree, hash in Postgres) as access paths for relations.

The current Postgres optimizer implementation performs a *near- exhaustive search* over the space of alternative strategies. This query optimization technique is inadequate to support database application domains that involve the need for extensive queries, such as artificial intelligence.

The Institute of Automatic Control at the University of Mining and Technology, in Freiberg, Germany, encountered the described problems as its folks wanted to take the Postgres DBMS as the backend for a decision support knowledge based system for the maintenance of an electrical power grid. The DBMS needed to handle large `join` queries for the inference machine of the knowledge based system.

Performance difficulties within exploring the space of possible query plans arose the demand for a new optimization technique being developed.

In the following we propose the implementation of a *Genetic Algorithm* as an option for the database query optimization problem.

58.2. Genetic Algorithms (GA)

The GA is a heuristic optimization method which operates through determined, randomized search. The set of possible solutions for the optimization problem is considered as a *population* of *individuals*. The degree of adaption of an individual to its environment is specified by its *fitness*.

The coordinates of an individual in the search space are represented by *chromosomes*, in essence a set of character strings. A *gene* is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be *binary* or *integer*.

Through simulation of the evolutionary operations *recombination*, *mutation*, and *selection* new generations of search points are found that show a higher average fitness than their ancestors.

[1] utesch@aut.tu-freiberg.de

According to the "comp.ai.genetic" FAQ it cannot be stressed too strongly that a GA is not a pure random search for a solution to a problem. A GA uses stochastic processes, but the result is distinctly non-random (better than random).

```
Structured Diagram of a GA:
---------------------------

P(t)    generation of ancestors at a time t
P''(t)  generation of descendants at a time t

+=========================================+
|>>>>>>>>>>>  Algorithm GA  <<<<<<<<<<<<<|
+=========================================+
| INITIALIZE t := 0                       |
+=========================================+
| INITIALIZE P(t)                         |
+=========================================+
| evalute FITNESS of P(t)                 |
+=========================================+
| while not STOPPING CRITERION do         |
|   +-------------------------------------+
|   | P'(t)  := RECOMBINATION{P(t)}       |
|   +-------------------------------------+
|   | P''(t) := MUTATION{P'(t)}           |
|   +-------------------------------------+
|   | P(t+1) := SELECTION{P''(t) + P(t)}  |
|   +-------------------------------------+
|   | evalute FITNESS of P''(t)           |
|   +-------------------------------------+
|   | t := t + 1                          |
+===+=====================================+
```

58.3. Genetic Query Optimization (GEQO) in Postgres

The GEQO module is intended for the solution of the query optimization problem similar to a traveling salesman problem (TSP). Possible query plans are encoded as integer strings. Each string represents the `join` order from one relation of the query to the next. E. g., the query tree

```
       /\
      /\ 2
     /\ 3
    4  1
```

is encoded by the integer string '4-1-3-2', which means, first join relation '4' and '1', then '3', and then '2', where 1, 2, 3, 4 are relids in Postgres.

Parts of the GEQO module are adapted from D. Whitley's Genitor algorithm.

Specific characteristics of the GEQO implementation in Postgres are:

- Usage of a *steady state* GA (replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time;

- Usage of *edge recombination crossover* which is especially suited to keep edge losses low for the solution of the TSP by means of a GA;
- Mutation as genetic operator is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

The GEQO module gives the following benefits to the Postgres DBMS compared to the Postgres query optimizer implementation:

- Handling of large `join` queries through non-exhaustive search;
- Improved cost size approximation of query plans since no longer plan merging is needed (the GEQO module evaluates the cost for a query plan as an individual).

58.4. Future Implementation Tasks for Postgres GEQO

58.4.1. Basic Improvements

58.4.1.1. Improve freeing of memory when query is already processed

With large `join` queries the computing time spent for the genetic query optimization seems to be a mere *fraction* of the time Postgres needs for freeing memory via routine `MemoryContextFree`, file `backend/utils/mmgr/mcxt.c`. Debugging showed that it get stucked in a loop of routine `OrderedElemPop`, file `backend/utils/mmgr/oset.c`. The same problems arise with long queries when using the normal Postgres query optimization algorithm.

58.4.1.2. Improve genetic algorithm parameter settings

In file `backend/optimizer/geqo/geqo_params.c`, routines `gimme_pool_size` and `gimme_number_generations`, we have to find a compromise for the parameter settings to satisfy two competing demands:

- Optimality of the query plan
- Computing time

58.4.1.3. Find better solution for integer overflow

In file `backend/optimizer/geqo/geqo_eval.c`, routine `geqo_joinrel_size`, the present hack for MAXINT overflow is to set the Postgres integer value of `rel->size` to its logarithm. Modifications of `Rel` in `backend/nodes/relation.h` will surely have severe impacts on the whole Postgres implementation.

58.4.1.4. Find solution for exhausted memory

Memory exhaustion may occur with more than 10 relations involved in a query. In file `backend/optimizer/geqo/geqo_eval.c`, routine `gimme_tree` is recursively called. Maybe I forgot something to be freed correctly, but I dunno what. Of course the `rel` data structure of the `join` keeps growing and growing the more relations are packed into it. Suggestions are welcome :-(

58.4.2. Further Improvements

Enable bushy query tree processing within Postgres; that may improve the quality of query plans.

References

Reference information for GEQ algorithms.

The Hitch-Hiker's Guide to Evolutionary Computation. Jörg Heitkötter and David Beasley. InterNet resource. *The Design and Implementation of the Postgres Query Optimizer*. Z. Fong. University of California, Berkeley Computer Science Department. *Fundamentals of Database Systems*. R. Elmasri and S. Navathe. The Benjamin/Cummings Pub., Inc..

Chapter 59. Frontend/Backend Protocol

Phil Thompson

Note

Written by Phil Thompson[1]. Updates for protocol 2.0 by Tom Lane[2].

Postgres uses a message-based protocol for communication between frontends and backends. The protocol is implemented over TCP/IP and also on Unix sockets. Postgres v6.3 introduced version numbers into the protocol. This was done in such a way as to still allow connections from earlier versions of frontends, but this document does not cover the protocol used by those earlier versions.

This document describes version 2.0 of the protocol, implemented in Postgres v6.4 and later.

Higher level features built on this protocol (for example, how `libpq` passes certain environment variables after the connection is established) are covered elsewhere.

59.1. Overview

The three major components are the frontend (running on the client) and the postmaster and backend (running on the server). The postmaster and backend have different roles but may be implemented by the same executable.

A frontend sends a startup packet to the postmaster. This includes the names of the user and the database the user wants to connect to. The postmaster then uses this, and the information in the pg_hba.conf(5) file to determine what further authentication information it requires the frontend to send (if any) and responds to the frontend accordingly.

The frontend then sends any required authentication information. Once the postmaster validates this it responds to the frontend that it is authenticated and hands over the connection to a backend. The backend then sends a message indicating successful startup (normal case) or failure (for example, an invalid database name).

Subsequent communications are query and result packets exchanged between the frontend and the backend. The postmaster takes no further part in ordinary query/result communication. (However, the postmaster is involved when the frontend wishes to cancel a query currently being executed by its backend. Further details about that appear below.)

When the frontend wishes to disconnect it sends an appropriate packet and closes the connection without waiting for a response for the backend.

Packets are sent as a data stream. The first byte determines what should be expected in the rest of the packet. The exception is packets sent from a frontend to the postmaster, which comprise a packet length then the packet itself. The difference is historical.

[1] mailto:phil@river-bank.demon.co.uk
[2] mailto:tgl@sss.pgh.pa.us

59.2. Protocol

This section describes the message flow. There are four different types of flows depending on the state of the connection: startup, query, function call, and termination. There are also special provisions for notification responses and command cancellation, which can occur at any time after the startup phase.

59.2.1. Startup

Startup is divided into an authentication phase and a backend startup phase.

Initially, the frontend sends a StartupPacket. The postmaster uses this info and the contents of the pg_hba.conf(5) file to determine what authentication method the frontend must use. The postmaster then responds with one of the following messages:

ErrorResponse

The postmaster then immediately closes the connection.

AuthenticationOk

The postmaster then hands over to the backend. The postmaster takes no further part in the communication.

AuthenticationKerberosV4

The frontend must then take part in a Kerberos V4 authentication dialog (not described here) with the postmaster. If this is successful, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationKerberosV5

The frontend must then take part in a Kerberos V5 authentication dialog (not described here) with the postmaster. If this is successful, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationUnencryptedPassword

The frontend must then send an UnencryptedPasswordPacket. If this is the correct password, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationEncryptedPassword

The frontend must then send an EncryptedPasswordPacket. If this is the correct password, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

If the frontend does not support the authentication method requested by the postmaster, then it should immediately close the connection.

After sending AuthenticationOk, the postmaster attempts to launch a backend process. Since this might fail, or the backend might encounter a failure during startup, the frontend must wait for the backend to acknowledge successful startup. The frontend should send no messages at this point. The possible messages from the backend during this phase are:

BackendKeyData

This message is issued after successful backend startup. It provides secret-key data that the frontend must save if it wants to be able to issue cancel requests later. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

ReadyForQuery

Backend startup is successful. The frontend may now issue query or function call messages.

ErrorResponse

Backend startup failed. The connection is closed after sending this message.

NoticeResponse

A warning message has been issued. The frontend should display the message but continue listening for ReadyForQuery or ErrorResponse.

The ReadyForQuery message is the same one that the backend will issue after each query cycle. Depending on the coding needs of the frontend, it is reasonable to consider ReadyForQuery as starting a query cycle (and then BackendKeyData indicates successful conclusion of the startup phase), or to consider ReadyForQuery as ending the startup phase and each subsequent query cycle.

59.2.2. Query

A Query cycle is initiated by the frontend sending a Query message to the backend. The backend then sends one or more response messages depending on the contents of the query command string, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new query or function call.

The possible response messages from the backend are:

CompletedResponse

An SQL command completed normally.

CopyInResponse

The backend is ready to copy data from the frontend to a relation. The frontend should then send a CopyDataRows message. The backend will then respond with a CompletedResponse message with a tag of "COPY".

CopyOutResponse

The backend is ready to copy data from a relation to the frontend. It then sends a CopyDataRows message, and then a CompletedResponse message with a tag of "COPY".

CursorResponse

The query was either an insert(l), delete(l), update(l), fetch(l) or a select(l) command. If the transaction has been aborted then the backend sends a CompletedResponse message with a tag of "*ABORT STATE*". Otherwise the following responses are sent.

For an insert(l) command, the backend then sends a CompletedResponse message with a tag of "INSERT *oid rows*" where *rows* is the number of rows inserted, and *oid* is the object ID of the inserted row if *rows* is 1, otherwise *oid* is 0.

For a delete(l) command, the backend then sends a CompletedResponse message with a tag of "DELETE *rows*" where *rows* is the number of rows deleted.

For an update(l) command, the backend then sends a CompletedResponse message with a tag of "UPDATE *rows*" where *rows* is the number of rows deleted.

For a fetch(l) or select(l) command, the backend sends a RowDescription message. This is then followed by an AsciiRow or BinaryRow message (depending on whether a binary cursor was specified) for each row being returned to the frontend. Finally, the backend sends a CompletedResponse message with a tag of "SELECT".

EmptyQueryResponse

An empty query string was recognized. (The need to specially distinguish this case is historical.)

ErrorResponse

An error has occurred.

ReadyForQuery

Processing of the query string is complete. A separate message is sent to indicate this because the query string may contain multiple SQL commands. (CompletedResponse marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the query. Notices are in addition to other responses, ie. the backend will continue processing the command.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message.

Actually, it is possible for NoticeResponse to arrive even when the frontend is not expecting any kind of message, that is, the backend is nominally idle. (In particular, the backend can be commanded to terminate by its postmaster. In that case it will send a NoticeResponse before closing the connection.) It is recommended that the frontend check for such asynchronous notices just before issuing any new command.

Also, if the frontend issues any listen(l) commands then it must be prepared to accept NotificationResponse messages at any time; see below.

59.2.3. Function Call

A Function Call cycle is initiated by the frontend sending a FunctionCall message to the backend. The backend then sends one or more response messages depending on the results of the function call, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new query or function call.

The possible response messages from the backend are:

ErrorResponse

An error has occurred.

FunctionResultResponse

The function call was executed and returned a result.

FunctionVoidResponse

The function call was executed and returned no result.

ReadyForQuery

Processing of the function call is complete. ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the function call. Notices are in addition to other responses, ie. the backend will continue processing the command.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message. Also, if it issues any listen(l) commands then it must be prepared to accept NotificationResponse messages at any time; see below.

59.2.4. Notification Responses

If a frontend issues a listen(l) command, then the backend will send a NotificationResponse message (not to be confused with NoticeResponse!) whenever a notify(l) command is executed for the same notification name.

Notification responses are permitted at any point in the protocol (after startup), except within another backend message. Thus, the frontend must be prepared to recognize a NotificationResponse message whenever it is expecting any message. Indeed, it should be able to handle NotificationResponse messages even when it is not engaged in a query.

NotificationResponse

A notify(l) command has been executed for a name for which a previous listen(l) command was executed. Notifications may be sent at any time.

It may be worth pointing out that the names used in listen and notify commands need not have anything to do with names of relations (tables) in the SQL database. Notification names are simply arbitrarily chosen condition names.

59.2.5. Cancelling Requests in Progress

During the processing of a query, the frontend may request cancellation of the query by sending an appropriate request to the postmaster. The cancel request is not sent directly to the backend for reasons of implementation efficiency: we don't want to have the backend constantly checking for new input from the frontend during query processing. Cancel requests should be relatively infrequent, so we make them slightly cumbersome in order to avoid a penalty in the normal case.

To issue a cancel request, the frontend opens a new connection to the postmaster and sends a CancelRequest message, rather than the StartupPacket message that would ordinarily be sent across a new connection. The postmaster will process this request and then close the connection. For security reasons, no direct reply is made to the cancel request message.

A CancelRequest message will be ignored unless it contains the same key data (PID and secret key) passed to the frontend during connection startup. If the request matches the PID and secret key for a currently executing backend, the postmaster signals the backend to abort processing of the current query.

The cancellation signal may or may not have any effect --- for example, if it arrives after the backend has finished processing the query, then it will have no effect. If the cancellation is effective, it results in the current command being terminated early with an error message.

The upshot of all this is that for reasons of both security and efficiency, the frontend has no direct way to tell whether a cancel request has succeeded. It must continue to wait for the backend to respond to the query. Issuing a cancel simply improves the odds that the current query will finish soon, and improves the odds that it will fail with an error message instead of succeeding.

Since the cancel request is sent to the postmaster and not across the regular frontend/backend communication link, it is possible for the cancel request to be issued by any process, not just the frontend whose query is to be canceled. This may have some benefits of flexibility in building multiple-process applications. It also introduces a security risk, in that unauthorized persons might try to cancel queries. The security risk is addressed by requiring a dynamically generated secret key to be supplied in cancel requests.

59.2.6. Termination

The normal, graceful termination procedure is that the frontend sends a Terminate message and immediately closes the connection. On receipt of the message, the backend immediately closes the connection and terminates.

An ungraceful termination may occur due to software failure (i.e., core dump) at either end. If either frontend or backend sees an unexpected closure of the connection, it should clean up and terminate. The frontend has the option of launching a new backend by recontacting the postmaster, if it doesn't want to terminate itself.

59.3. Message Data Types

This section describes the base data types used in messages.

Int*n*(*i*)

An *n* bit integer in network byte order. If *i* is specified it is the literal value. Eg. Int16, Int32(42).

LimString*n*(*s*)

A character array of exactly *n* bytes interpreted as a '\0' terminated string. The '\0' is omitted if there is insufficient room. If *s* is specified it is the literal value. Eg. LimString32, LimString64("user").

String(*s*)

A conventional C '\0' terminated string with no length limitation. A frontend should always read the full string even though it may have to discard characters if its buffers aren't big enough.

Note

Is 8193 bytes the largest allowed size?

If *s* is specified it is the literal value. Eg. String, String("user").

Byte*n*(*c*)

Exactly *n* bytes. If *c* is specified it is the literal value. Eg. Byte, Byte1('\n').

59.4. Message Formats

This section describes the detailed format of each message. Each can be sent by either a frontend (F), a postmaster/backend (B), or both (F & B).

AsciiRow (B)

Byte1('D')

Identifies the message as an ASCII data row. (A prior RowDescription message defines the number of fields in the row and their data types.)

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 (MSB) of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 (LSB) of the 1st byte, the 9th field corresponds to bit 7 of the 2nd byte, and so on. Each bit is set if the value of the corresponding field is not NULL. If the number of fields is not a multiple of 8, the remainder of the last byte in the bit map is wasted.

Then, for each field with a non-NULL value, there is the following:

Int32

Specifies the size of the value of the field, including this size.

Byte*n*

Specifies the value of the field itself in ASCII characters. *n* is the above size minus 4. There is no trailing '\0' in the field data; the front end must add one if it wants one.

AuthenticationOk (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(0)

Specifies that the authentication was successful.

AuthenticationKerberosV4 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(1)

Specifies that Kerberos V4 authentication is required.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(2)

Specifies that Kerberos V5 authentication is required.

AuthenticationUnencryptedPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(3)

Specifies that an unencrypted password is required.

AuthenticationEncryptedPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(4)

Specifies that an encrypted password is required.

Byte2

The salt to use when encrypting the password.

BackendKeyData (B)

Byte1('K')

Identifies the message as cancellation key data. The frontend must save these values if it wishes to be able to issue CancelRequest messages later.

Int32

The process ID of this backend.

Int32

The secret key of this backend.

BinaryRow (B)

Byte1('B')

Identifies the message as a binary data row. (A prior RowDescription message defines the number of fields in the row and their data types.)

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 (MSB) of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 (LSB) of the 1st byte, the 9th field corresponds to bit 7 of the 2nd byte, and so on. Each bit is set if the value of the corresponding field is not NULL. If the number of fields is not a multiple of 8, the remainder of the last byte in the bit map is wasted.

Then, for each field with a non-NULL value, there is the following:

Int32

Specifies the size of the value of the field, excluding this size.

Byte*n*

Specifies the value of the field itself in binary format. *n* is the above size.

CancelRequest (F)

Int32(16)

The size of the packet in bytes.

Int32(80877102)

The cancel request code. The value is chosen to contain "1234" in the most significant 16 bits, and "5678" in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

Int32

The process ID of the target backend.

Int32

The secret key for the target backend.

CompletedResponse (B)

Byte1('C')

Identifies the message as a completed response.

String

The command tag. This is usually (but not always) a single word that identifies which SQL command was completed.

CopyDataRows (B & F)

This is a stream of rows where each row is terminated by a Byte1('\n'). This is then followed by the sequence Byte1('\\'), Byte1('.'), Byte1('\n').

CopyInResponse (B)

Byte1('G')

Identifies the message as a Start Copy In response. The frontend must now send a CopyDataRows message.

CopyOutResponse (B)

Byte1('H')

Identifies the message as a Start Copy Out response. This message will be followed by a CopyDataRows message.

CursorResponse (B)

Byte1('P')

Identifies the message as a cursor response.

String

The name of the cursor. This will be "blank" if the cursor is implicit.

EmptyQueryResponse (B)

Byte1('I')

Identifies the message as a response to an empty query string.

String("")

Unused.

EncryptedPasswordPacket (F)

Int32

The size of the packet in bytes.

String

The encrypted (using crypt()) password.

ErrorResponse (B)

Byte1('E')

Identifies the message as an error.

String

The error message itself.

FunctionCall (F)

Byte1('F')

Identifies the message as a function call.

String("")

Unused.

Int32

Specifies the object ID of the function to call.

Int32

Specifies the number of arguments being supplied to the function.

Then, for each argument, there is the following:

Int32

Specifies the size of the value of the argument, excluding this size.

Byte*n*

Specifies the value of the field itself in binary format. *n* is the above size.

FunctionResultResponse (B)

Byte1('V')

Identifies the message as a function call result.

Byte1('G')

Specifies that a nonempty result was returned.

Int32

Specifies the size of the value of the result, excluding this size.

Byte*n*

Specifies the value of the result itself in binary format. *n* is the above size.

Byte1('0')

Unused. (Strictly speaking, FunctionResultResponse and FunctionVoidResponse are the same thing but with some optional parts to the message.)

FunctionVoidResponse (B)

Byte1('V')

Identifies the message as a function call result.

Byte1('0')

Specifies that an empty result was returned.

NoticeResponse (B)

Byte1('N')

Identifies the message as a notice.

String

The notice message itself.

NotificationResponse (B)

Byte1('A')

Identifies the message as a notification response.

Int32

The process ID of the notifying backend process.

String

The name of the condition that the notify has been raised on.

Query (F)

Byte1('Q')

Identifies the message as a query.

String

The query string itself.

ReadyForQuery (B)

Byte1('Z')

Identifies the message type. ReadyForQuery is sent whenever the backend is ready for a new query cycle.

RowDescription (B)

Byte1('T')

Identifies the message as a row description.

Int16

Specifies the number of fields in a row (may be zero).

Then, for each field, there is the following:

String

Specifies the field name.

Int32

Specifies the object ID of the field type.

Int16

Specifies the type size.

Int32

Specifies the type modifier.

StartupPacket (F)

Int32(296)

The size of the packet in bytes.

Int32

The protocol version number. The most significant 16 bits are the major version number. The least 16 significant bits are the minor version number.

LimString64

The database name, defaults to the user name if empty.

LimString32

The user name.

LimString64

Any additional command line arguments to be passed to the backend by the postmaster.

LimString64

Unused.

LimString64

The optional tty the backend should use for debugging messages.

Terminate (F)

Byte1('X')

Identifies the message as a termination.

UnencryptedPasswordPacket (F)

Int32

The size of the packet in bytes.

String

The unencrypted password.

Chapter 60. Postgres Signals

Massimo Dal Zotto

Note

Contributed by Massimo Dal Zotto[1]

Postgres uses the following signals for communication between the postmaster and backends:

Table 60.1. Postgres Signals

Signal	postmaster Action	Server Action
SIGHUP	kill(*,sighup)	read_pg_options
SIGINT	die	cancel query
SIGQUIT	kill(*,sigterm)	handle_warn
SIGTERM	kill(*,sigterm), kill(*,9), die	die
SIGPIPE	ignored	die
SIGUSR1	kill(*,sigusr1), die	quickdie
SIGUSR2	kill(*,sigusr2)	async notify (SI flush)
SIGCHLD	reaper	ignored (alive test)
SIGTTIN	ignored	
SIGTTOU	ignored	
SIGCONT	dumpstatus	
SIGFPE		FloatExceptionHandler

Note

"kill(*,signal)" means sending a signal to all backends.

The main changes to the old signal handling are the use of SIGQUIT instead of SIGHUP to handle warns, SIGHUP to re-read the pg_options file and the redirection to all active backends of SIGHUP, SIGTERM, SIGUSR1 and SIGUSR2 sent to the postmaster. In this way these signals sent to the postmaster can be sent automatically to all the backends without need to know their pids. To shut down postgres one needs only to send a SIGTERM to postmaster and it will stop automatically all the backends.

The SIGUSR2 signal is also used to prevent SI cache table overflow which happens when some backend doesn't process SI cache for a long period. When a backend detects the SI table full at 70% it simply sends a signal to the postmaster which will wake up all idle backends and make them flush the cache.

The typical use of signals by programmers could be the following:

```
# stop postgres
kill -TERM $postmaster_pid

# kill all the backends
kill -QUIT $postmaster_pid
```

[1] mailto:dz@cs.unitn.it

```
# kill only the postmaster
kill -INT $postmaster_pid

# change pg_options
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $postmaster_pid

# change pg_options only for a backend
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $backend_pid
cat old_pg_options > $DATA_DIR/pg_options
```

Chapter 61. gcc Default Optimizations

Brian Gallew

Note

Contributed by Brian Gallew[1]

Configuring gcc to use certain flags by default is a simple matter of editing the `/usr/local/lib/gcc-lib/`*`platform`*`/`*`version`*`/specs` file. The format of this file pretty simple. The file is broken into sections, each of which is three lines long. The first line is "*`section_name`*:" (e.g. "*asm:"). The second line is a list of flags, and the third line is blank.

The easiest change to make is to append the desired default flags to the list in the appropriate section. As an example, let's suppose that I have linux running on a '486 with gcc 2.7.2 installed in the default location. In the file /usr/local/lib/gcc-lib/i486-linux/2.7.2/specs, 13 lines down I find the following section:

```
- ----------SECTION----------
*cc1:


- ----------SECTION----------
```

As you can see, there aren't any default flags. If I always wanted compiles of C code to use "-m486 -fomit-frame-pointer", I would change it to look like:

```
- ----------SECTION----------
*cc1:
- -m486 -fomit-frame-pointer

- ----------SECTION----------
```

If I wanted to be able to generate 386 code for another, older linux box lying around, I'd have to make it look like this:

```
- ----------SECTION----------
*cc1:
%{!m386:-m486} -fomit-frame-pointer

- ----------SECTION----------
```

This will always omit frame pointers, any will build 486-optimized code unless -m386 is specified on the command line.

You can actually do quite a lot of customization with the specs file. Always remember, however, that these changes are global, and affect all users of the system.

[1] mailto:geek+@cmu.edu

Chapter 62. Backend Interface

Backend Interface (BKI) files are scripts that are input to the Postgres backend running in the special "bootstrap" mode that allows it to perform database functions without a database system already existing. BKI files can therefore be used to create the database system in the first place. initdb uses BKI files to do just that: to create a database system. However, initdb's BKI files are generated internally. It generates them using the files `global1.bki.source` and `local1.template1.bki.source`, which it finds in the Postgres "library" directory. They get installed there as part of installing Postgres. These .source files get build as part of the Postgres build process, by a build program called genbki. genbki takes as input Postgres source files that double as genbki input that builds tables and C header files that describe those tables.

Related information may be found in documentation for initdb, createdb, and the SQL command `CREATE DATABASE`.

62.1. BKI File Format

The Postgres backend interprets BKI files as described below. This description will be easier to understand if the `global1.bki.source` file is at hand as an example. (As explained above, this .source file isn't quite a BKI file, but you'll be able to guess what the resulting BKI file would be anyway).

Commands are composed of a command name followed by space separated arguments. Arguments to a command which begin with a "$" are treated specially. If "$$" are the first two characters, then the first "$" is ignored and the argument is then processed normally. If the "$" is followed by space, then it is treated as a NULL value. Otherwise, the characters following the "$" are interpreted as the name of a macro causing the argument to be replaced with the macro's value. It is an error for this macro to be undefined.

Macros are defined using

```
define macro macro_name = macro_value
```

and are undefined using

```
undefine macro macro_name
```

and redefined using the same syntax as define.

Lists of general commands and macro commands follow.

62.2. General Commands

OPEN *classname*

Open the class called *classname* for further manipulation.

CLOSE [*classname*]

Close the open class called *classname*. It is an error if *classname* is not already opened. If no *classname* is given, then the currently open class is closed.

PRINT

Print the currently open class.

INSERT [OID=*oid_value*] (*value1 value2* ...)

Insert a new instance to the open class using *value1*, *value2*, etc., for its attribute values and *oid_value* for its OID. If *oid_value* is not "0", then this value will be used as the instance's object identifier. Otherwise, it is an error.

INSERT (*value1 value2* ...)

As above, but the system generates a unique object identifier.

CREATE *classname* (*name1* = *type1* [,*name2* = *type2*[,...]])

Create a class named *classname* with the attributes given in parentheses.

OPEN (*name1* = *type1* [,*name2* = *type2*[,...]]) AS *classname*

Open a class named *classname* for writing but do not record its existence in the system catalogs. (This is primarily to aid in bootstrapping.)

DESTROY *classname*

Destroy the class named *classname*.

DEFINE INDEX *indexname* ON *class_name* USING *amname* (*opclass attr* | (*function*(*attr*))

Create an index named *indexname* on the class named *classname* using the *amname* access method. The fields to index are called *name1*, *name2* etc., and the operator collections to use are *collection_1*, *collection_2* etc., respectively.

Note

This last sentence doesn't reference anything in the example. Should be changed to make sense. - Thomas 1998-08-04

62.3. Macro Commands

DEFINE FUNCTION *macro_name* AS *rettype function_name*(*args*)

Define a function prototype for a function named *macro_name* which has its value of type *rettype* computed from the execution *function_name* with the arguments *args* declared in a C-like manner.

DEFINE MACRO *macro_name* FROM FILE *filename*

Define a macro named *macro_name* which has its value read from the file called *filename*.

62.4. Debugging Commands

Note

This section on debugging commands was commented-out in the original documentation. Thomas 1998-08-05

r

Randomly print the open class.

m -1

Toggle display of time information.

m 0

Set retrievals to now.

m 1 Jan 1 01:00:00 1988

Set retrievals to snapshots of the specfied time.

m 2 Jan 1 01:00:00 1988, Feb 1 01:00:00 1988

Set retrievals to ranges of the specified times. Either time may be replaced with space if an unbounded time range is desired.

&A *classname natts name1 type1 name2 type2 ...*

Add *natts* attributes named *name1*, *name2*, etc. of types *type1*, *type2*, etc. to the class *classname*.

&RR *oldclassname newclassname*

Rename the *oldclassname* class to *newclassname*.

&RA classname oldattname newattname *classname oldattname newattname*

Rename the *oldattname* attribute in the class named *classname* to *newattname*.

62.5. Example

The following set of commands will create the “pg_opclass” class containing the *int_ops* collection as an object with an OID of *421*, print out the class, and then close it.

```
create pg_opclass (opcname=name)
open pg_opclass
insert oid=421 (int_ops)
print
close pg_opclass
```

Chapter 63. Page Files

A description of the database file default page format.

This section provides an overview of the page format used by Postgres classes. User-defined access methods need not use this page format.

In the following explanation, a *byte* is assumed to contain 8 bits. In addition, the term *item* refers to data which is stored in Postgres classes.

63.1. Page Structure

The following table shows how pages in both normal Postgres classes and Postgres index classes (e.g., a B-tree index) are structured.

Table 63.1. Sample Page Layout

Item	Description
itemPointerData	
filler	
itemData...	
Unallocated Space	
ItemContinuationData	
Special Space	
``ItemData 2''	
``ItemData 1''	
ItemIdData	
PageHeaderData	

The first 8 bytes of each page consists of a page header (PageHeaderData). Within the header, the first three 2-byte integer fields (*lower*, *upper*, and *special*) represent byte offsets to the start of unallocated space, to the end of unallocated space, and to the start of *special space*. Special space is a region at the end of the page which is allocated at page initialization time and which contains information specific to an access method. The last 2 bytes of the page header, *opaque*, encode the page size and information on the internal fragmentation of the page. Page size is stored in each page because frames in the buffer pool may be subdivided into equal sized pages on a frame by frame basis within a class. The internal fragmentation information is used to aid in determining when page reorganization should occur.

Following the page header are item identifiers (*ItemIdData*). New item identifiers are allocated from the first four bytes of unallocated space. Because an item identifier is never moved until it is freed, its index may be used to indicate the location of an item on a page. In fact, every pointer to an item (*ItemPointer*) created by Postgres consists of a frame number and an index of an item identifier. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a set of attribute bits which affect its interpretation.

The items themselves are stored in space allocated backwards from the end of unallocated space. Usually, the items are not interpreted. However when the item is too long to be placed on a single page or when fragmentation of the item is desired, the item is divided and each piece is handled as distinct items in the following manner. The first through the next to last piece are placed in an item continuation structure

(*ItemContinuationData*). This structure contains itemPointerData which points to the next piece and the piece itself. The last piece is handled normally.

63.2. Files

`data/`

Location of shared (global) database files.

`data/base/`

Location of local database files.

63.3. Bugs

The page format may change in the future to provide more efficient access to large objects.

This section contains insufficient detail to be of any assistance in writing a new access method.

Part VII. Appendices

Additional related information.

Table of Contents

Appendix A. Documentation

Thomas Lockhart
Tom Ivar Helbekkmo

The purpose of documentation is to make Postgres easier to learn, use, and develop. The documentation set should describe the Postgres system, language, and interfaces. It should be able to answer common questions and to allow a user to find those answers on his own without resorting to mailing list support.

A.1. Documentation Roadmap

Postgres has four primary documentation formats:

- Plain text for pre-installation information.
- HTML, for on-line browsing and reference.
- Hardcopy, for in-depth reading and reference.
- man pages, for quick reference.

Table A.1. Postgres Documentation Products

File	Description	
./COPYRIGHT	Copyright notice	
./INSTALL	Installation instructions (text from sgml->rtf->text)	
./README	Introductory info	
./register.txt	Registration message during make	
./doc/bug.template	Bug report template	
./doc/postgres.tar.gz	Integrated docs (HTML)	
./doc/programmer.ps.gz	Programmer's Guide (Postscript)	
./doc/programmer.tar.gz	Programmer's Guide (HTML)	
./doc/reference.ps.gz	Reference Manual (Postscript)	
./doc/reference.tar.gz	Reference Manual (HTML)	
./doc/tutorial.ps.gz	Introduction (Postscript)	
./doc/tutorial.tar.gz	Introduction (HTML)	
./doc/user.ps.gz	User's Guide (Postscript)	
./doc/user.tar.gz	User's Guide (HTML)	

There are man pages available for installation, as well as a large number of plain-text README-type files throughout the Postgres source tree.

A.2. Documentation Sources

Documentation sources include plain text files, man pages, and html. However, most new Postgres documentation will be written using the *Standard Generalized Markup Language* (SGML) DocBook[1]

[1] http://www.ora.com/davenport/

Document Type Definition (DTD). Much of the existing documentation has been or will be converted to SGML.

The purpose of SGML is to allow an author to specify the structure and content of a document (e.g. using the DocBook DTD), and to have the document style define how that content is rendered into a final form (e.g. using Norm Walsh's stylesheets).

Documentation has accumulated from several sources. As we integrate and assimilate existing documentation into a coherent documentation set, the older versions will become obsolete and will be removed from the distribution. However, this will not happen immediately, and will not happen to all documents at the same time. To ease the transition, and to help guide developers and writers, we have defined a transition roadmap.

Here is the documentation plan for v6.5:

- Start compiling index information for the User's and Administrator's Guides.
- Write more sections for the User's Guide covering areas outside the reference pages. This would include introductory information and suggestions for approaches to typical design problems.
- Merge information in the existing man pages into the reference pages and User's Guide. Condense the man pages down to reminder information, with references into the primary doc set.
- Convert the new sgml reference pages to new man pages, replacing the existing man pages.
- Convert all source graphics to CGM format files for portability. Currently we mostly have Applix Graphics sources from which we can generate .gif output. One graphic is only available in .gif and .ps, and should be redrawn or removed.

A.2.1. Document Structure

There are currently five separate documents written in DocBook. Each document has a container source document which defines the DocBook environment and other document source files. These primary source files are located in `doc/src/sgml/`, along with many of the other source files used for the documentation. The primary source files are:

postgres.sgml

This is the integrated document, including all other documents as parts. Output is generated in HTML since the browser interface makes it easy to move around all of the documentation by just clicking. The other documents are available in both HTML and hardcopy.

tutorial.sgml

The introductory tutorial, with examples. Does not include programming topics, and is intended to help a reader unfamiliar with SQL. This is the "getting started" document.

user.sgml

The User's Guide. Includes information on data types and user-level interfaces. This is the place to put information on "why".

reference.sgml

The Reference Manual. Includes Postgres SQL syntax. This is the place to put information on "how".

programming.sgml

The Programmer's Guide. Includes information on Postgres extensibility and on the programming interfaces.

admin.sgml

The Administrator's Guide. Include installation and release notes.

A.3. The Documentation Project

Packaged documentation is available in both HTML and *Postscript* formats. These are available as part of the standard Postgres installation. We discuss here working with the documentation sources and generating documentation packages.

Note

This is the first release of new Postgres documentation in three years. The content and environment are in flux and still evolving.

The purpose of SGML is to allow an author to specify the structure and content of a document (e.g. using the DocBook DTD), and to have the document style define how that content is rendered into a final form (e.g. using Norm Walsh's stylesheets).

See Introduction to DocBook[2] for a nice "quickstart" summary of DocBook features. DocBook Elements[3] provides a powerful cross-reference for features of DocBook.

This documentation set is constructed using several tools, including James Clark's jade[4] and Norm Walsh's Modular DocBook Stylesheets[5].

Currently, hardcopy is produced by importing *Rich Text Format* (RTF) output from jade into ApplixWare for minor formatting fixups then exporting as a Postscript file.

TeX[6] is a supported format for jade output, but was not used at this time for several reasons, including the inability to make minor format fixes before committing to hardcopy and generally inadequate table support in the TeX stylesheets.

A.3.1. Styles and Conventions

DocBook has a rich set of tags and constructs, and a suprisingly large percentage are directly and obviously useful for well-formed documentation. The Postgres documentation set has only recently been adapted to SGML, and in the near future several sections of the set will be selected and maintained as prototypical examples of DocBook usage. Also, a short summary of DocBook tags will be included below.

A.3.2. Authoring Tools

The current Postgres documentation set is written using a plain text editor (or emacs/psgml; see below) with the content marked up using SGML.

[2] http://nis-www.lanl.gov/~rosalia/mydocs/docbook-intro.html
[3] http://www.ora.com/homepages/dtdparse/docbook/3.0/
[4] http://www.jclark.com/jade/
[5] http://www.berkshire.net/~norm/docbook/dsssl
[6] http://sunsite.unc.edu/pub/packages/TeX/systems/unix/

SGML and DocBook do not suffer from an oversupply of open-source authoring tools. The most common toolset is the emacs/xemacs editing package with the psgml feature extension. On some systems (e.g. RedHat Linux) these tools are provided in a typical full installation.

A.3.2.1. emacs/psgml

When using emacs/psgml, a comfortable way of working with these separate files of book parts is to insert a proper DOCTYPE declaration while you're editing them. If you are working on this source, for instance, it's an appendix chapter, so you would specify the document as an "appendix" instance of a DocBook document by making the first line look like this:

```
!doctype appendix PUBLIC "-//Davenport//DTD DocBook V3.0//EN"
```

This means that anything and everything that reads SGML will get it right, and I can verify the document with "nsgmls -s docguide.sgml".

A.4. Building Documentation

GNU make is used to build documentation from the DocBook sources. There are a few environment definitions which may need to be set or modified for your installation. The `Makefile` looks for `doc/../src/Makefile` and (implicitly) for `doc/../src/Makefile.custom` to obtain environment information. On my system, the `src/Makefile.custom` looks like

```
# Makefile.custom
# Thomas Lockhart 1998-03-01

POSTGRESDIR= /opt/postgres/current
CFLAGS+= -m486
YFLAGS+= -v

# documentation

HSTYLE= /home/tgl/SGML/db107.d/docbook/html
PSTYLE= /home/tgl/SGML/db107.d/docbook/print
```

where HSTYLE and PSTYLE determine the path to `docbook.dsl` for HTML and hardcopy (print) stylesheets, respectively. These stylesheet file names are for Norm Walsh's Modular Style Sheets; if other stylesheets are used then one can define HDSL and PDSL as the full path and file name for the stylesheet, as is done above for HSTYLE and PSTYLE. On many systems, these stylesheets will be found in packages installed in `/usr/lib/sgml/`, `/usr/share/lib/sgml/`, or `/usr/local/lib/sgml/`.

HTML documentation packages can be generated from the SGML source by typing

```
% cd doc/src
% make tutorial.tar.gz
% make user.tar.gz
% make admin.tar.gz
% make programmer.tar.gz
% make postgres.tar.gz
% make install
```

These packages can be installed from the main documentation directory by typing

```
% cd doc
% make install
```

A.5. Hardcopy Generation for v6.4

The hardcopy Postscript documentation is generated by converting the SGML source code to RTF, then importing into ApplixWare-4.4.1. After a little cleanup (see the following section) the output is "printed" to a postscript file.

Some figures were redrawn to avoid having bitmap GIF files in the hardcopy documentation. One figure, of the system catalogs, was sufficiently complex that there was not time to redraw it. It was converted to fit using the following commands:

```
% convert -monochrome -v -geometry 500x500'>' catalogs.ps catalogs.gif
% convert -v -crop 400x500 catalogs.gif catalogs-cropped.gif
```

A.5.1. RTF Cleanup Procedure

Several items must be addressed in generating Postscript hardcopy:

Applixware RTF Cleanup

Applixware does not seem to do a complete job of importing RTF generated by jade/MSS. In particular, all text is given the “Header1” style attribute label, although the text formatting itself is acceptable. Also, the Table of Contents page numbers do not refer to the section listed in the table, but rather refer to the page of the ToC itself.

1. Generate the RTF input by typing

   ```
   % cd doc/src/sgml
   % make tutorial.rtf
   ```

2. Open a new document in Applix Words and then import the RTF file.

3. Print out the existing Table of Contents, to mark up in the following few steps.

4. Insert figures into the document. Center each figure on the page using the centering margins button.

   Not all documents have figures. You can grep the SGML source files for the string “Graphic” to identify those parts of the documentation which may have figures. A few figures are replicated in various parts of the documentation.

5. Work through the document, adjusting page breaks and table column widths.

6. If a bibliography is present, Applix Words seems to mark all remaining text after the first title as having an underlined attribute. Select all remaining text, turn off underlining using the underlining button, then explicitly underline each document and book title.

7. Work through the document, marking up the ToC hardcopy with the actual page number of each ToC entry.

8. Replace the right-justified incorrect page numbers in the ToC with correct values. This only takes a few minutes per document.

9. Save the document as native Applix Words format to allow easier last minute editing later.

10. Export the document to a file in Postscript format.

11. Compress the Postscript file using gzip. Place the compressed file into the `doc` directory.

A.6. Toolsets

We have documented experience with two installation methods for the various tools that are needed to process the documentation. One is installation from RPMs on Linux, the other is a general installation from original distributions of the individual tools. Both will be described below.

We understand that there are some other packaged distributions for these tools. FreeBSD seems to have them available. Please report package status to the docs mailing list and we will include that information here.

A.6.1. RPM installation on Linux

Install RPMs[7] for Jade and related packages.

A.6.2. Manual installation of tools

This is a brief run-through of the process of obtaining and installing the software you'll need to edit DocBook source with Emacs and process it with Norman Walsh's DSSSL style sheets to create HTML and RTF.

These instructions do not cover new jade/DocBook support in the sgml-tools package. The authors have not tried this package since it adopted DocBook, but it is almost certainly a good candidate for use.

A.6.2.1. Prerequisites

What you need:

- A working installation of GCC 2.7.2
- A working installation of Emacs 19.19 or later
- An unzip program for UNIX to unpack things

What you must fetch:

- James Clark's Jade version 1.1[8]
- DocBook version 3.0[9]
- Norman Walsh's Modular Stylesheets version 1.19[10]
- Lennart Staflin's PSGML version 1.0.1[11]

Important URLs:

- The Jade web page[12]
- The DocBook web page[13]

[7] ftp://ftp.cygnus.com/pub/home/rosalia/
[8] ftp://ftp.jclark.com/pub/jade/jade1_1.zip
[9] http://www.ora.com/davenport/docbook/current/docbk30.zip
[10] http://nwalsh.com/docbook/dsssl/db119.zip
[11] ftp://ftp.lysator.liu.se/pub/sgml/psgml-1.0.1.tar.gz
[12] http://www.jclark.com/jade/
[13] http://www.ora.com/davenport/

- The Modular Stylesheets web page[14]
- The PSGML web page[15]
- Steve Pepper's Whirlwind Guide[16]
- Robin Cover's database of SGML software[17]

A.6.2.2. Installing Jade

Installing Jade

1. Read the installation instructions at the above listed URL.
2. Unzip the distribution kit in a suitable place. The command to do this will be something like

   ```
   unzip -aU jade1_1.zip
   ```

3. Jade is not built using GNU Autoconf, so you'll need to edit a `Makefile` yourself. Since James Clark has been good enough to prepare his kit for it, it is a good idea to make a build directory (named for your machine architecture, perhaps) under the main directory of the Jade distribution, copy the file `Makefile` from the main directory into it, edit it there, and then run `make` there.

   However, the `Makefile` does need to be edited. There is a file called `Makefile.jade` in the main directory, which is intended to be used with `make -f Makefile.jade` when building Jade (as opposed to just SP, the SGML parser kit that Jade is built upon). We suggest that you don't do that, though, since there is more that you need to change than what is in `Makefile.jade`, so you'd have to edit one of them anyway.

   Go through the `Makefile`, reading James' instructions and editing as needed. There are various variables that need to be set. Here is a collected summary of the most important ones, with typical values:

   ```
   prefix = /usr/local
   XDEFINES = -DSGML_CATALOG_FILES_DEFAULT=\"/usr/local/share/sgml/
   catalog\"
   XLIBS = -lm
   RANLIB = ranlib
   srcdir = ..
   XLIBDIRS = grove spgrove style
   XPROGDIRS = jade
   ```

   Note the specification of where to find the default catalog of SGML support files -- you may want to change that to something more suitable for your own installation. If your system doesn't need the above settings for the math library and the `ranlib` command, leave them as they are in the `Makefile`.

4. Type `make` to build Jade and the various SP tools.
5. Once the software is built, `make install` will do the obvious.

[14] http://nwalsh.com/docbook/dsssl/
[15] http://www.lysator.liu.se/projects/about_psgml.html
[16] http://www.infotek.no/sgmltool/guide.htm
[17] http://www.sil.org/sgml/publicSW.html

A.6.2.3. Installing the DocBook DTD Kit

Installing the DocBook DTD Kit

1. You'll want to place the files that make up the DocBook DTD kit in the directory you built Jade to expect them in, which, if you followed our suggestion above, is `/usr/local/share/sgml/`. In addition to the actual DocBook files, you'll need to have a `catalog` file in place, for the mapping of document type specifications and external entity references to actual files in that directory. You'll also want the ISO character set mappings, and probably one or more versions of HTML.

   One way to install the various DTD and support files and set up the `catalog` file, is to collect them all into the above mentioned directory, use a single file named `CATALOG` to describe them all, and then create the file `catalog` as a catalog pointer to the former, by giving it the single line of content:

   ```
   CATALOG /usr/local/share/sgml/CATALOG
   ```

2. The `CATALOG` file should then contain three types of lines. The first is the (optional) SGML declaration, thus:

   ```
   SGMLDECL docbook.dcl
   ```

   Next, the various references to DTD and entity files must be resolved. For the DocBook files, these lines look like this:

   ```
   PUBLIC "-//Davenport//DTD DocBook V3.0//EN" docbook.dtd
   PUBLIC "-//USA-DOD//DTD Table Model 951010//EN" cals-tbl.dtd
   PUBLIC "-//Davenport//ELEMENTS DocBook Information Pool V3.0//EN"
    dbpool.mod
   PUBLIC "-//Davenport//ELEMENTS DocBook Document Hierarchy V3.0//
   EN" dbhier.mod
   PUBLIC "-//Davenport//ENTITIES DocBook Additional General Entities
    V3.0//EN" dbgenent.mod
   ```

3. Of course, a file containing these comes with the DocBook kit. Note that the last item on each of these lines is a file name, given here without a path. You can put the files in subdirectories of your main SGML directory if you like, of course, and modify the reference in the `CATALOG` file. DocBook also references the ISO character set entities, so you need to fetch and install these (they are available from several sources, and are easily found by way of the URLs listed above), along with catalog entries for all of them, such as:

   ```
   PUBLIC "ISO 8879-1986//ENTITIES Added Latin 1//EN" ISO/ISOlat1
   ```

   Note how the file name here contains a directory name, showing that we've placed the ISO entity files in a subdirectory named `ISO`. Again, proper catalog entries should accompany the entity kit you fetch.

A.6.2.4. Installing Norman Walsh's DSSSL Style Sheets

Installing Norman Walsh's DSSSL Style Sheets

1. Read the installation instructions at the above listed URL.

2. To install Norman's style sheets, simply unzip the distribution kit in a suitable place. A good place to dot this would be `/usr/local/share`, which places the kit in a directory tree under `/usr/local/share/docbook`. The command will be something like

```
unzip -aU db107.zip
```

3. One way to test the installation is to build the HTML and RTF forms of the PostgreSQL manual.

   a. To build the HTML files, go to the SGML source directory, `doc/src/sgml`, and say

   ```
   jade -t sgml -d /usr/local/share/docbook/html/docbook.dsl -
   D ../graphics postgres.sgml
   ```

   `book1.htm` is the top level node of the output..

   b. To generate the RTF output, ready for importing into your favorite word processing system and printing, type:

   ```
   jade -t rtf -d /usr/local/share/docbook/print/docbook.dsl -
   D ../graphics postgres.sgml
   ```

A.6.2.5. Installing PSGML

Installing PSGML

1. Read the installation instructions at the above listed URL.

2. Unpack the distribution file, run configure, make and make install to put the byte-compiled files and info library in place.

3. Then add the following lines to your `/usr/local/share/emacs/site-lisp/site-start.el` file to make Emacs properly load PSGML when needed:

   ```
   (setq load-path
         (cons "/usr/local/share/emacs/site-lisp/psgml" load-path))
   (autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t)
   ```

4. (Optional) If you want to use PSGML when editing HTML too, also add this:

   ```
   (setq auto-mode-alist
         (cons '("\\.s?html?\\'" . sgml-mode) auto-mode-alist))
   ```

5. (Optional) There is one important thing to note with PSGML: its author assumed that your main SGML DTD directory would be `/usr/local/lib/sgml`. If, as in the examples in this chapter, you use `/usr/local/share/sgml`, you have to compensate for this.

   a. (Optional) You can set the `SGML_CATALOG_FILES` environment variable.

   b. (Optional) You can customize your PSGML installation (its manual tells you how).

   c. (Optional) You can even edit the source file `psgml.el` before compiling and installing PSGML, changing the hard-coded paths to match your own default.

A.6.2.6. Installing JadeTeX

If you want to, you can also install JadeTeX to use TeX as a formatting backend for Jade. Note that this is still quite unpolished software, and will generate printed output that is inferior to what you get from the RTF backend. Still, it works all right, especially for simpler documents that don't use tables, and as both JadeTeX and the style sheets are under continuous improvement, it will certainly get better over time.

To install and use JadeTeX, you will need a working installation of TeX and LaTeX2e, including the supported tools and graphics packages, Babel, AMS fonts and AMS-LaTeX, the PSNFSS extension and companion kit of "the 35 fonts", the dvips program for generating PostScript, the macro packages fancyhdr, hyperref, minitoc, url and ot2enc, and of course JadeTeX itself. All of these can be found on your friendly neighborhood CTAN site.

JadeTeX does not at the time of writing come with much of an installation guide, but there is a `makefile` which shows what is needed. It also includes a directory `cooked`, wherein you'll find some of the macro packages it needs, but not all, and not complete -- at least last we looked.

Before building the `jadetex.fmt` format file, you'll probably want to edit the `jadetex.ltx` file, to change the configuration of Babel to suit your locality. The line to change looks something like

```
\RequirePackage[german,french,english]{babel}[1997/01/23]
```

and you should obviously list only the languages you actually need, and have configured Babel for.

With JadeTeX working, you should be able to generate and format TeX output for the PostgreSQL manuals by giving the commands (as above, in the `doc/src/sgml` directory)

```
jade -t tex -d /usr/local/share/docbook/print/docbook.dsl -D ../
graphics postgres.sgml
jadetex postgres.tex
jadetex postgres.tex
dvips postgres.dvi
```

Of course, when you do this, TeX will stop during the second run, and tell you that its capacity has been exceeded. This is, as far as we can tell, because of the way JadeTeX generates cross referencing information. TeX can, of course, be compiled with larger data structure sizes. The details of this will vary according to your installation.

A.7. Alternate Toolsets

sgml-tools v2.x now supports jade and DocBook. It may be the preferred toolset for working with SGML but we have not had a chance to evaluate the new package.

Appendix B. Contacts

- Thomas Lockhart[1] works on SQL standards compliance and documentation.

[1] http://alumni.caltech.edu/~lockhart

SQL References

Selected references and readings for SQL and Postgres.

SQL Reference Books

Reference texts for SQL features.

The Practical SQL Handbook. Bowman et al, 1993. Using Structured Query Language. 3. Judity Bowman, Sandra Emerson, and Marcy Damovsky. 0-201-44787-8. 1996. Addison-Wesley. Copyright © 1997 Addison-Wesley Longman, Inc..

A Guide to the SQL Standard. Date and Darwen, 1997. A user's guide to the standard database language SQL. 4. C. J. Date and Hugh Darwen. 0-201-96426-0. 1997. Addison-Wesley. Copyright © 1997 Addison-Wesley Longman, Inc..

Understanding the New SQL. Melton and Simon, 1993. A complete guide. Jim Melton and Alan R. Simon. 1-55860-245-3. 1993. Morgan Kaufmann. Copyright © 1993 Morgan Kaufmann Publishers, Inc..

PostgreSQL-Specific Documentation

This section is for related documentation.

The PostgreSQL. Administrator's Guide. The Administrator's Guide. Thomas Lockhart. 1998-10-01. The PostgreSQL Global Development Group.

The PostgreSQL. Developer's Guide. The Developer's Guide. Thomas Lockhart. 1998-10-01. The PostgreSQL Global Development Group.

The PostgreSQL. Programmer's Guide. The Programmer's Guide. Thomas Lockhart. 1998-10-01. The PostgreSQL Global Development Group.

The PostgreSQL. Tutorial Introduction. The Tutorial. Thomas Lockhart. 1998-10-01. The PostgreSQL Global Development Group.

The PostgreSQL. User's Guide. The User's Guide. Thomas Lockhart. 1998-10-01. The PostgreSQL Global Development Group.

The Postgres95. User Manual. Yu and Chen, 1995. A. Yu and J. Chen. . Sept. 5, 1995. University of California, Berkeley CA.

Proceedings and Articles

This section is for articles and newsletters.

Partial indexing in POSTGRES : research project. Olson, 1993. Nels Olson. 1993. UCB Engin T7.49.1993 O676. University of California, Berkeley CA.

A Unified Framework for Version Modeling Using Production Rules in a Database System. Ong and Goh, 1990. L. Ong and J. Goh. April, 1990. ERL Technical Memorandum M90/33. University of California, Berkeley CA.

The Postgres. Data Model. Rowe and Stonebraker, 1987. L. Rowe and M. Stonebraker. Sept. 1987. VLDB Conference, Brighton, England. 1987. .

Generalized partial indexes[1]. . P. Seshadri and A. Swami. March 1995. Eleventh International Conference on Data Engineering. . 1995. Cat. No.95CH35724. IEEE Computer Society Press.

The Design of Postgres. . Stonebraker and Rowe, 1986. M. Stonebraker and L. Rowe. May 1986. Conference on Management of Data, Washington DC. ACM-SIGMOD. 1986. .

The Design of the Postgres. Rules System. Stonebraker, Hanson, Hong, 1987. M. Stonebraker, E. Hanson, and C. H. Hong. Feb. 1987. Conference on Data Engineering, Los Angeles, CA. IEEE. 1987. .

The Postgres. Storage System. Stonebraker, 1987. M. Stonebraker. Sept. 1987. VLDB Conference, Brighton, England. 1987. .

A Commentary on the Postgres. Rules System. Stonebraker et al, 1989. M. Stonebraker, M. Hearst, and S. Potamianos. Sept. 1989. Record 18(3). SIGMOD. 1989. .

The case for partial indexes (DBMS)[2]. Stonebraker, M, 1989b. M. Stonebraker. Dec. 1989. Record 18(no.4):4-11. SIGMOD. 1989. .

The Implementation of Postgres. . Stonebraker, Rowe, Hirohama, 1990. M. Stonebraker, L. A. Rowe, and M. Hirohama. March 1990. Transactions on Knowledge and Data Engineering 2(1). IEEE. .

On Rules, Procedures, Caching and Views in Database Systems. Stonebraker et al, ACM, 1990. M. Stonebraker and et al. June 1990. Conference on Management of Data. ACM-SIGMOD. .

[1] http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z
[2] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf