PostgreSQL

PostgreSQL

Covering v6.3 for general release

Table of Contents

List of Figures

List of Tables

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, has been 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 [[STON86]] and the definition of the initial data model appeared in [[ROWE87]]. The design of the rule system at that time was described in [[STON87a]]. The rationale and architecture of the storage manager were detailed in [[STON87b]].

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 [[STON90a]], to a few external users in June 1989. In response to a critique of the first rule system ([[STON89]]), the rule system was redesigned ([[STON90b]]) and Version 2 was released in June 1990

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

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] picked up the code and commercialized it. Postgres became the primary data manager for the Sequoia 2000[4] 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[5] and Jolly Chen[6] 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).

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

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

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%, and backend startup speed has decreased 80%.

1.3. About This Release

From now on, We will use Postgres to mean PostgreSQL.

PostgreSQL is available without cost. This manual describes version 6.3 of PostgreSQL.

Check the Administrator's Guide for a list of currently supported machines. In general, PostgreSQL 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.

Reference Manual

Detailed reference information on command syntax. At the moment, this manual is very sparse, but eventually should contain information similar to that in the man pages.

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[7] 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[8] 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[9] mailing list is the place to get going.

1.5. Copyrights and Trademarks

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

Chapter 2. Architecture

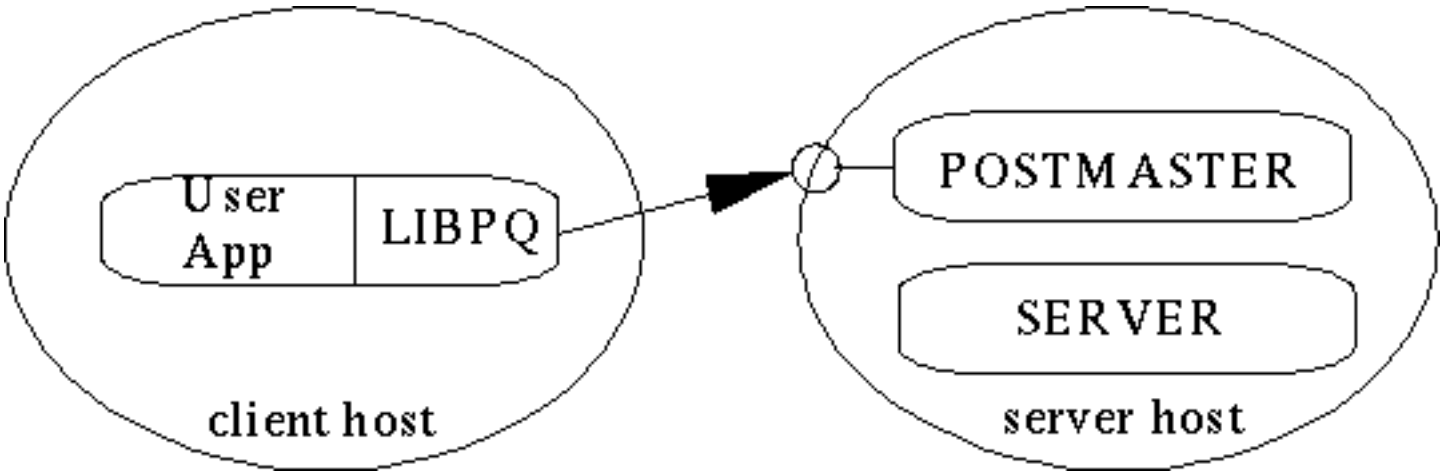
2.1. Postgres Architectural Concepts

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 (Figure 2.1), 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 [programmers-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 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 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        char16[][]
);
```

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 *char16* (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
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
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
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
time	time	time of day
timespan	interval	general-use time span
timestamp	timestamp with time zone	date/time
varchar(n)	character varying(n)	variable-length character string

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

Postgres Function	SQL92 Constant	Description
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 cases in the standard which are ill considered and which should not live through subsequent standards. Postgres will not make great efforts to conform to these cases. However, these cases tend to be little-used and obsure, and a typical user is not likely to run into them.

Although most of the input and output functions corresponding to the base types (e.g., integers and floating point numbers) do some error-checking, some are not particularly rigorous about it. More importantly, few of the operators and functions (e.g., addition and multiplication) perform any error-checking at all. Consequently, many of the numeric operators can (for example) silently underflow or overflow.

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.

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
int2	2 bytes	Fixed-precision	-32768 to +32767
int4	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
float4	4 bytes	Variable-precision	7 decimal places
float8	8 bytes	Variable-precision	14 decimal places

The *exact numerics* `decimal` and `numeric` have fully implemented syntax but currently (Postgres v6.3) support only a small range of precision and/or range values.

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 Numeric Types

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

The *numeric* should eventually replace the money type. It has a fully implemented syntax but currently (Postgres v6.3) support only a small range of precision and/or range values and cannot 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 are currently other fixed-length character types. These provide no additional functionality and are likely to be deprecated in the future.

Table 8.6. Postgres Specialty Character Types

Character Type	Storage	Description
char2	2 bytes	Two characters
char4	4 bytes	Four characters
char8	8 bytes	Eight characters
char16	16 bytes	Sixteen characters

8.4. Date/Time Types

There are two fundamental kinds of date and time measurements: clock time and time interval. Both quantities have 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 date and time.

Other date and time types are available also, 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 precis ion
time	4 bytes	SQL92 type	wide range
timespan	12 bytes	best general time interval	wide range, high precision
timestamp	4 bytes	SQL92 type	limited range

Table 8.8. Postgres Date/Time Ranges

Date/Time Type	Earliest	Latest	Resolution
abstime	1901-12-14	2038-01-19	1 sec

Date/Time Type	Earliest	Latest	Resolution
date	4713 BC	no limit	1 day
datetime	4713 BC	no limit	1 microsec to 14 digits
interval	no limit	no limit	1 microsec
reltime	-68 years	+68 years	1 sec
time	00:00:00.00	23:59:59.99	1 microsec
timespan	no limit	no limit	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. For example, although the date type does not have an associated time zone, the time type can. The default time zone is specified as a constant 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 will vary through the year.

To obviate 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 v6.3 (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.
- `set timezone` SQL 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

Constant	Description
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. 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)
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.

Chapter 9. Operators

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)
```

Table 9.1. 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 sensit ive	'thomas' !~ '*.Thomas*.'
!~*	Does not match (regex), case insens itive	'thomas' !~ '*.vadim*.'

Table 9.2. Postgres Numerical Operators

Operator	Description	Usage
!	Factorial	3 !
!!	Factorial (left operator)	!! 3
%	Modulo	5 % 4
%	Truncate	% 4.5
*	Multiplication	2 * 3

Operator	Description	Usage
+	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

Table 9.3. 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
<<	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

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 9.4. 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?	

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 `\dd` command to show these operators.

Chapter 10. Functions

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

Some functions defined for text are also available for char() and varchar().

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`).

Table 10.1. Mathematical Functions

Function	Returns	Description	Example
float(int)	float8	convert integer to floating point	float(2)
float4(int)	float4	convert integer to floating point	float4(2)
int	integer(float)	convert floating point to integer	integer(2.0)

Many of the string functions are available for text, varchar(), and char() types. At the moment, some functions are available only for the text type.

Table 10.2. String Functions

Function	Returns	Description	Example
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)
upper(text)	text	convert text to upper case	upper('tom')

Table 10.3. Date/Time Functions

Function	Returns	Description	Example
isfinite(abstime)	bool	TRUE if this is a finite time	isfinite('now'::abstime)
datetime(abstime)	datetime	convert to datetime	datetime('now'::abstime)
datetime(date)	datetime	convert to datetime	datetime('today'::date)

Function	Returns	Description	Example
datetime(date,time)	datetime	convert to datetime	datetime('1998-02-24':: datetime, '23:07'::time);
age(datetime,datetime)	timespan	span preserving months and years	age('now','1957-06-13':: datetime)
date_part(text,datetime)	float8	specified portion of date field	date_part('dow','now':: datetime)
date_trunc(text,datetime)	datetime	truncate date at specified units	date_trunc('month','now':: abstime)
isfinite(datetime)	bool	TRUE if this is a finite time	isfinite('now'::datetime)
abstime(datetime)	abstime	convert to abstime	abstime('now'::datetime)
timespan(reltime)	timespan	convert to timespan	timespan('4 hours'::reltime)
datetime(date,time)	datetime	convert to datetime	datetime('1998-02-25':: date,'06:41'::time)
date_part(text,timespan)	float8	specified portion of time field	date_part('hour','4 hrs 3 mins'::timespan)
isfinite(timespan)	bool	TRUE if this is a finite time	isfinite('4 hrs'::timespan)
reltime(timespan)	reltime	convert to reltime	reltime('4 hrs'::timespan)

Table 10.4. Geometric Functions

Function	Returns	Description	Example
box(point,point)	box	convert points to box	box('(0,0)'::point,'(1,1)':: point)
area(box)	float8	area of box	area('((0,0),(1,1))'::box)
isopen(path)	bool	TRUE if this is an open path	isopen('[(0,0),(1,1),(2,0)]':: path)
isclosed(path)	bool	TRUE if this is a closed path	isclosed('((0,0),(1,1),(2,0)) '::path)
circle(point,float8)	circle	convert to circle	circle('(0,0)'::point,2.0)
polygon(npts,circle)	polygon	convert to polygon with npts points	polygon(12,'((0,0),2.0)':: circle)
center(circle)	float8	center of object	center('((0,0),2.0)'::circle)
radius(circle)	float8	radius of circle	radius('((0,0),2.0)'::circle)
diameter(circle)	float8	diameter of circle	diameter('((0,0),2.0)':: circle)
area(circle)	float8	area of circle	area('((0,0),2.0)'::circle)

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

Table 10.5. SQL92 Text Functions

Function	Returns	Description	Example
position(text in text)	int4	extract specified substring	position('o' in 'Tom')

Function	Returns	Description	Example
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')

Chapter 11. 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        char16[][]
);
```

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 *char16* (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 12. 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 13. 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 [MELT93][1] or [DATE93]. You should also be aware that some features are not part of the ANSI standard.

13.1. 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.

13.2. 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.

13.3. Populating a Class with Instances

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

[1] refs.html#MELT93

```
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.

13.4. 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;
```

13.5. 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.

13.6. 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.

13.7. 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';
```

13.8. 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.

13.9. 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 14. Disk Storage

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

Chapter 15. 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.

15.1. 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 16. pgaccess

This section needs to be written. Volunteers?

Part III. Administrator's Guide

Installation and maintenance information.

Table of Contents

Chapter 17. Ports

17.1. Currently Supported Platforms

Postgres is available free of charge. This manual describes version 6.3 of Postgres. The authors have compiled and tested Postgres on the following platforms:

Table 17.1. Supported Platforms

OS	Processor	Version	Reported	Remarks
AIX 4.1.x-4.2	RS6000	v6.3	1998-03-01	4.1.4.0,4.2 (Darren King[1]), 4.1.5 (Andreas Zeugswetter[2]); 3.2.5 confirmed on v6.2.1 (Frank Dana[3])
BSDi	x86	v6.3	1998-03-01	(Bruce Momjian[4]
FreeBSD 2.2.x-3.x	x86	v6.3	1998-03-01	(Tatsuo Ishii[5], Marc Fournier[6])
NetBSD 1.3	x86	v6.3	1998-03-01	(Brook Milligan[7])
NetBSD 1.3	Sparc	v6.3	1998-03-01	(Tom I Helbekkmo[8])
NetBSD 1.3	VAX	v6.3	1998-03-01	(Tom I Helbekkmo[9])
DGUX 5.4R4.11	m88k	v6.3	1998-03-01	(Brian E Gallew[10])
HPUX 10.20	PA-RISC	v6.3	1998-03-01	9.0.x confirmed on v6.2.1 (Stan Brown[11])
IRIX 6.x	MIPS	v6.3	1998-03-01	5.x is different (Andrew Martin[12])
Digital 4.0	Alpha	v6.3.2	1998-04-16	reported working for DUnix/v3.2g (Pedro J. Lobo[13])
linux 2.0.x	Alpha	v6.3.2	1998-04-16	mostly successful (Ryan Kirkpatrick[14], Jeff Sturm [15])
linux 2.0.x	x86	v6.3.2	1998-04-16	(Thomas Lockhart[16], Tatsuo Ishii[17])

[1] mailto:darrenk@insightdist.com
[2] mailto:Andreas.Zeugswetter@telecom.at
[3] mailto:danaf@ans.net
[4] mailto:maillist@candle.pha.pa.us
[5] mailto:t-ishii@sra.co.jp
[6] mailto:scrappy@hub.org
[7] mailto:brook@trillium.NMSU.Edu
[8] mailto:tih@hamartun.priv.no
[9] mailto:tih@hamartun.priv.no
[10] mailto:geek+@cmu.edu
[11] mailto:stanb@awod.com
[12] mailto:martin@biochemistry.ucl.ac.uk
[13] mailto:pjlobo@euitt.upm.es
[14] mailto:rkirkpat@nag.cs.colorado.edu
[15] mailto:jsturm@zenacomp.com
[16] mailto:lockhart@alumni.caltech.edu
[17] mailto:t-ishii@sra.co.jp

OS	Processor	Version	Reported	Remarks
linux 2.0.x	Sparc	v6.3	1998-03-01	(Tom Szybist[18])
mklinux	PPC	v6.3	1998-03-01	(Tatsuo Ishii[19])
SCO	x86	v6.3	1998-03-01	partial success (Billy G. Allie[20])
Solaris	x86	v6.3	1998-03-01	(Marc Fournier[21])
Solaris 2.5.1-2.6	x86	v6.3	1998-03-01	(Marc Fournier[22], Tatsuo Ishii[23])
SunOS 4.1.4	Sparc	v6.3	1998-03-01	patches submitted (Tatsuo Ishii[24])
SVR4	MIPS	v6.3	1998-03-01	similar to v6.2.1; "mostly working" (Frank Ridderbusch[25])
SVR4 4.4	m88k	v6.2.1	1998-03-01	confirmed with patching (Doug Winterburn[26])
Unixware	x86	v6.3	1998-03-01	aka UNIVEL (Billy G. Allie[27])
NextStep	x86	v6.x	1998-03-01	client-only support; v1.0.9 worked with patches (David Wetzel[28])

17.2. 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 17.2. 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[29])
NetBSD	m68k	v6.3	1998-03-01	Amiga, HP300, Mac; not yet working (Henry Hotz[30])

[18] mailto:szybist@boxhill.com
[19] mailto:t-ishii@sra.co.jp
[20] mailto:Bill.Allie@mug.org
[21] mailto:scrappy@hub.org
[22] mailto:scrappy@hub.org
[23] mailto:t-ishii@sra.co.jp
[24] mailto:t-ishii@sra.co.jp
[25] mailto:ridderbusch.pad@sni.de
[26] mailto:dlw@seavme.xroads.com
[27] mailto:Bill.Allie@mug.org
[28] mailto:dave@turbocat.de
[29] mailto:dmill@globalnet.co.uk
[30] mailto:hotz@jpl.nasa.gov

OS	Processor	Version	Reported	Remarks
Ultrix	MIPS,VAX?	v6.x	1998-03-01	no recent reports; obsolete?
Windows NT	all	v6.3	1998-03-01	not library compatible; client side maybe; use ODBC/JDBC
Windows	x86	v6.3	1998-03-01	not library compatible; client side maybe; use ODBC/JDBC

Note that Windows ports of the frontend are apparently possible using third-party Posix porting tools and libraries.

Chapter 18. Installation

Complete installation instructions for Postgres v6.3.

This procedure is This is based on the installation instructions for Postgres v6.3 found in `$PGROOT/INSTALL`. Up to date information on Postgres may be found at www.postgresql.org[1].

The installation 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).

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 vary wildly on what options you should use on each platform. *Use common sense* before typing in these commands.

Our Makefiles require GNU make (called "gmake" in this document) and also assume that install accepts BSD options. The INSTALL variable in the Makefiles is set to the BSD-compatible version of install. On some systems, you will have to find a BSD-compatible install (eg. bsdinst, which comes with the MIT X Window System distribution).

18.1. Requirements to Run Postgres

Information on supported platforms is another chapter. In general, most Unix-compatible platforms with modern libraries should be able to run Postgres.

You should have at least 8 MB of memory and at least 45 MB of disk space to hold the source, binaries, and user databases. After installation you may reduce this to about 3 Mbytes plus space for user databases.

18.2. Installation Procedure

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 account postgres if it does not already exist.

[1] http://www.postgresql.org

3. Log into account postgres.

   - Check that you have sufficient disk space. You will need about 17 Mbytes for /usr/src/pgsql, about 2 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`.

4. Ftp file ftp://ftp.postgresql.org/pub/postgresql-v6.3.tar.gz 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.

   To install it, type the following:

   ```
   cd
   gunzip -c flex-2.5.4.tar.gz | tar xvf -
   cd flex-2.5.4
   configure --prefix=/usr
   make
   make check
   # You must be root when typing the next line.
   make install
   cd
   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 link /usr/bin/flex++ which points to flex.

6. 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. Also, do not use the pg_dumpall script from v6.0 or everything will be owned by the Postgres super user. Type (with the gunzip line and the following line typed as one line):

   ```
   cd
   gunzip -c postgresql-v6.3.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 > 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, 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, use "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 files /usr/src/pgsql/migration/*.

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 "???" replaced by the process id for process "postmaster". (Do not use the id for process "grep postmaster".) Type kill ??? with "???" modified as indicated.

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; be consistant throughout this procedure. 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-v6.3.tar.gz | tar xvf -
    ```

11. Configure the source code for your system. It is this step at which you can specify your actual source path and installation paths for the build process (see the --prefix option below). Type

    ```
    cd /usr/src/pgsql/src
    ./configure
    ```

The configure program will list the template files available and ask you to choose one. A lot of times, an appropriate template file is chosen for you, and you can just press Enter to accept the default. If the default is not appropriate, then type in the appropriate template file and press Enter. (If you do this, then send email to scrappy@hub.org stating the output of the program './config.guess' and what the template file should be.)

Once you have entered the template file, you will be asked a number of questions about your particular configuration. These can be skipped by adding parameters to the configure command above. The following parameters can be tagged onto the end of the configure command:

```
       --prefix=BASEDIR   Selects a different base directory for
 the
                          installation of the Postgres
 configuration.
                          The default is /usr/local/pgsql.

       --enable-hba       Enables Host Based Authentication (
DEFAULT)

       --disable-hba      Disables Host Based Authentication

       --enable-locale    Enables USE_LOCALE

       --disable-locale   Disables USE_LOCALE (DEFAULT)

       --enable-cassert   Enables ASSERT_CHECKING

       --disable-cassert  Disables ASSERT_CHECKING (DEFAULT)

       --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.
                          (If the configure script cannot find the
                          specified template file, it will ask you
 for
                          one).

       --with-pgport=PORT Sets the port that the postmaster
 process
                          listens for incoming connections on.
 The
                          default for this is port 5432.
```

As an example, here is the configure script I use on a Sparc Solaris 2.5 system with /opt/postgres being the install base.

```
   ./configure --prefix=/opt/postgres \
--with-template=sparc_solaris-gcc --with-pgport=5432 \
--enable-hba --disable-locale
```

Of course, in a real shell, you would type these three lines all on the same line.

12. 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.)

    If your computer does not have gmake (GNU make) then try running make instead throughout the rest of these notes.

    Please note that 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 an error 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.

13. 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.

14. If necessary, tell UNIX how to find your shared libraries. If you are using Linux-ELF do ONE of the following, preferably the first:

    a. (Optional) As root, edit file /etc/ld.so.conf. Add 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.

15. If it has not already been done, then prepare account postgres for using Postgres. Any account that will use Postgres must be similarily prepared. (The following instructions are for a bash shell. Adapt accordingly for other shells.)

    Add the following lines to your login 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
```

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
```

16. Create the database. *Do not do the following as root!* This would be a major security hole. Type

    ```
    initdb
    ```

17. 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 finsihed. If you are upgrading from v6.0 you can copy file pg_hba.conf from your old database on top of the one in your new database, rather than redoing this from scratch.

18. You may wish to skip the regression tests. However, we think skipping the tests is a BAD idea!

    The file /usr/src/pgsql/src/test/regress/README has detailed instructions for running and interpreting the regression tests. A short version follows here:

    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). DO NOT RUN POSTMASTER FROM THE ROOT ACCOUNT.

19. Run the regression tests. Type

    ```
    cd
    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 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
    ```

    "Failed" tests may have failed due to slightly different error messages, output formatting, failure to set the timezone correctly for your platform, etc. "Failures" of this type do not indicate a problem with Postgres.

    For a i686/Linux-ELF platform, no tests failed since this is the v6.3 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 v6.2.1, and that previous versions of Postgres have been in use successfully for some time now.

After running the tests, type

```
destroydb regression
cd /usr/src/pgsql/src/test/regress
gmake clean
```

20. Stop the postmaster as described in step 7. Then restore the timezone to it's normal setting. If you changed the timezone by modifying environment variable TZ then one way to do this is to log out of, then back into, account postgres.

21. Start the postmaster daemon running. Type

    ```
    cd
    nohup postmaster > server.log 2>&1 &
    ```

    Run postmaster from your Postgres super user account (typically account postgres). DO NOT RUN POSTMASTER FROM THE ROOT ACCOUNT.

22. If you haven't already done so, this would be a good time to modify your computer so that it will automatically start postmaster whenever you boot your computer. Here are some suggestions on how to do this, contributed by various users. Whatever you do, postmaster must be run by user 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. a) 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" b) 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. Note: Unlike most other examples, this one has been tested. c) In RedHat v4.0 Linux edit file /etc/inittab to contain the following 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.) d) The contrib/linux area of the Postgres distribution has an example init.d script compatible with and tested using recent RedHat packages.

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: a) Run the SQL command vacuum. This will clean up your database. b) Back up your system. (You should probably keep the last few backups on hand.) Ideally, 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 install 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 YourTable set PathCol = UpgradePath(PathCol);
    update YourTable set PolyCol = UpgradePoly(PolyCol);
    ...
    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. 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 # Print out the man pages. man -a -t /usr/local/pgsql/man/*/* > manpage.ps gshp -sOUTPUTFILE=manpage.hp manpage.ps rm manpage.ps lpr -l -s -r manpage.hp # Print out the Postgres95 User Manual, version 1.0, # Sept. 5, 1996. cd /usr/src/pgsql/doc gshp -sOUTPUTFILE=userguide.hp userguide.ps lpr -l -s -r userguide.hp If you are a developer, you will probably want to also print out the Postgres Implemention Guide, version 1.0, October 1, 1995. This is a WWW document located at http://www.postgresql.org/docs/impguide.

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 telling us the following: - The version of Postgres (v6.2.1, 6.1.1, beta 970703, etc.). - Your operating system (i.e. RedHat v4.0 Linux v2.0.26). - 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!

18.3. 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
   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.

18.4. 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.

Mail questions to pgsql-questions@postgresql.org. For more information on the various mailing lists, see http://www.postgresql.org and look for the mailing lists.

18.5. Porting Notes

> **Note**
>
> For some ports, these notes may be out of date.

18.5.1. Ultrix4.x

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

18.5.2. Linux

18.5.2.1. 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.

18.5.2.2. 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[2]

18.5.3. BSD/OS

For BSD/OS 2.0 and 2.01, you will need to get the GNU dld library.

18.5.4. NeXT

The NeXT port for v1.09 was supplied by Tom R. Hageman[3]. 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.

[2] sneaker@powergrid.electriciti.com
[3] mailto:tom@basil.icce.rug.nl

We have no recent reports of successful NeXT installations (for v6.2.1). However, the client-side libraries should work even if the backend is not supported.

Chapter 19. Runtime Environment

Figure 19.1. Postgres file layout

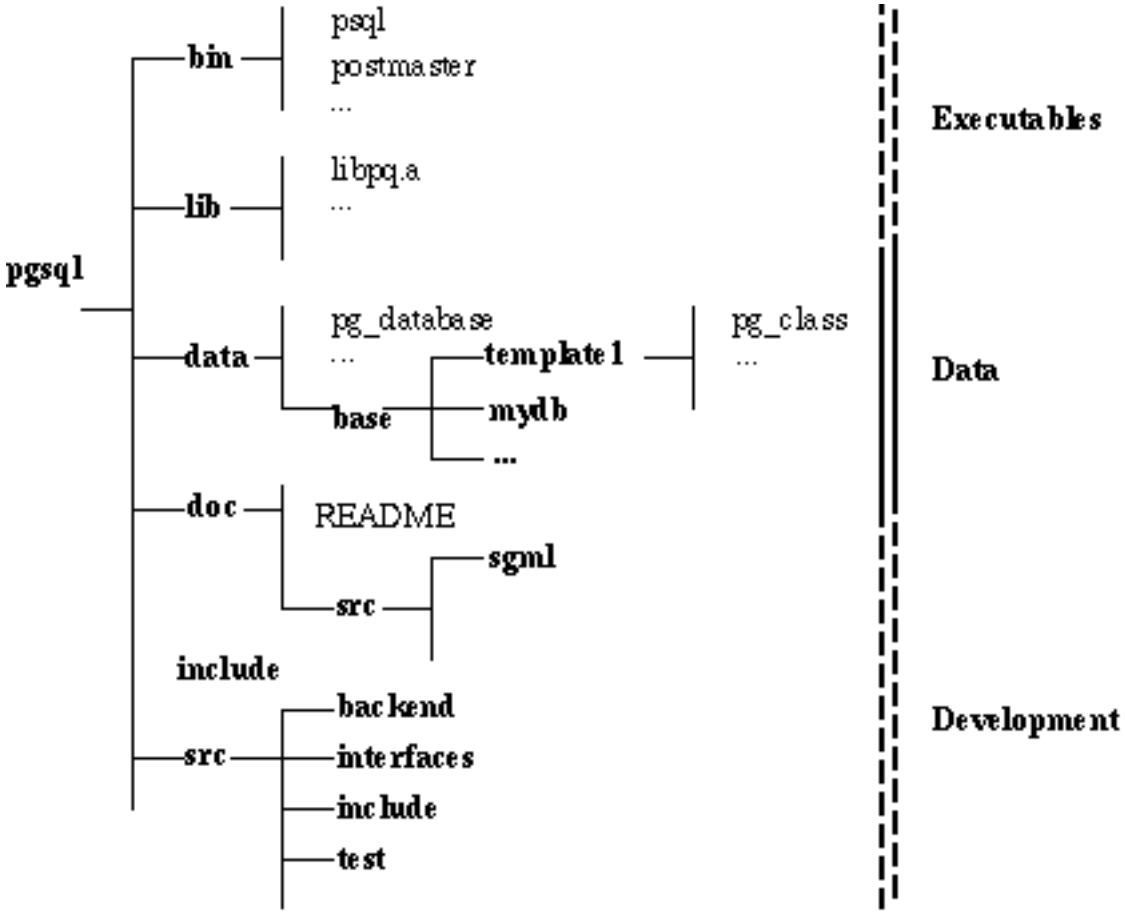


Figure 19.1 shows how the Postgres distribution is laid out when installed in the default way. For simplicity, we will assume that Postgres has been installed in the directory `/usr/local/pgsql`. Therefore, wherever you see the directory `/usr/local/pgsql` you should substitute the name of the directory 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 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.

19.1. 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 postgresql 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 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 location: `/usr/lib/locale` (Linux, Solaris), `/usr/share/locale` (Linux), `/usr/lib/nls/loc` (DUX 4.0) Check man locale for right place. Under Linux I did a symbolical link between `/usr/lib/locale` and `/usr/share/locale` to be sure next libc will not break my locale.

[1] http://www.sai.msu.su/~megera/postgres/

19.1.1. 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 USE_LOCALE variable.

19.1.2. What are the Drawbacks?

There is one evident drawback of using locale - it's speed ! So, use locale only if you really need it.

Chapter 20. 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 section of this manual titled, "Administering Postgres." 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 21. 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 22. Disk Management

22.1. 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 23. 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
 '4322'?
```

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 24. 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.

24.1. 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.

24.2. Accessing a Database

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

- running the Postgres terminal monitor programs (monitor or 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 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 “/* ... */”

24.3. 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 25. Database Recovery

This section needs to be written. Volunteers?

Chapter 26. 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).

26.1. 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
```

26.2. 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.
```

26.3. 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
```

26.4. 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.

26.4.1. 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.

26.4.2. 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.

26.4.3. 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.

26.4.4. 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.

26.4.5. 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.

26.4.6. 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;
```

26.4.7. 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.

26.4.8. 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 27. Release Notes

> **Note**
>
> Should include the migration notes from `migration/`.

The release notes have not yet been integrated into the new documentation. Check for plain text files in the top of the distribution directory tree and in the `migration/` directory for current information.

27.1. Release 6.3

TBD

27.2. Release 6.2.1

> **Note**
>
> v6.2.1 was a bug-fix and usability release on v6.2. Needs only a few notes.

27.3. Release 6.2

> **Note**
>
> This should include information based on Bruce's release summary.

27.4. Release 6.1

> **Note**
>
> This should include information based on Bruce's release summary.

The regression tests have been adapted and extensively modified for the v6.1 release of PostgreSQL.

Three new data types (datetime, timespan, and circle) have been added to the native set of PostgreSQL types. Points, boxes, paths, and polygons have had their output formats made consistant across the data types. The polygon output in misc.out has only been spot-checked for correctness relative to the original regression output.

PostgreSQL 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).

27.5. Timing Results

These timing results are from running the regression test with the command

```
% time make runtest
```

Timing under Linux 2.0.27 seems to have a roughly 5% variation from run to run, presumably due to the timing vagaries of multitasking systems.

27.5.1. v6.3

```
  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
```

27.5.2. v6.1

```
  Time   System
  06:12  Pentium Pro 180, 32MB, 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 28. 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 an extremely brief overview 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.

28.1. Copyrights and Trademarks

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

Chapter 29. Architecture

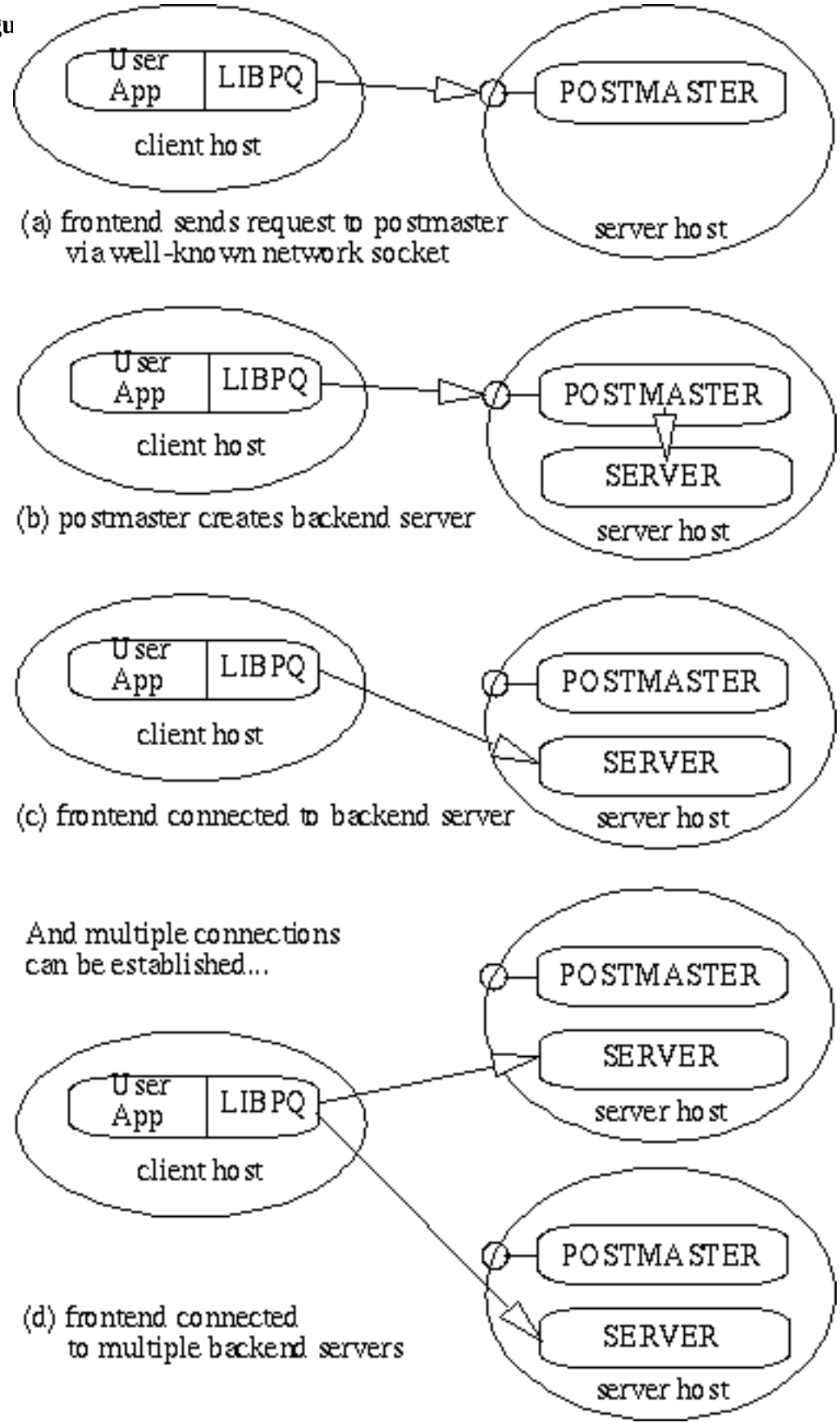
29.1. Postgres Architectural Concepts

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 (Figure 29.1(a)), which in turn starts a new backend server process (Figure 29.1(b))

Figu


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

and connects the frontend process to the new server (Figure 29.1(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 30. 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

30.1. 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.

30.2. 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.

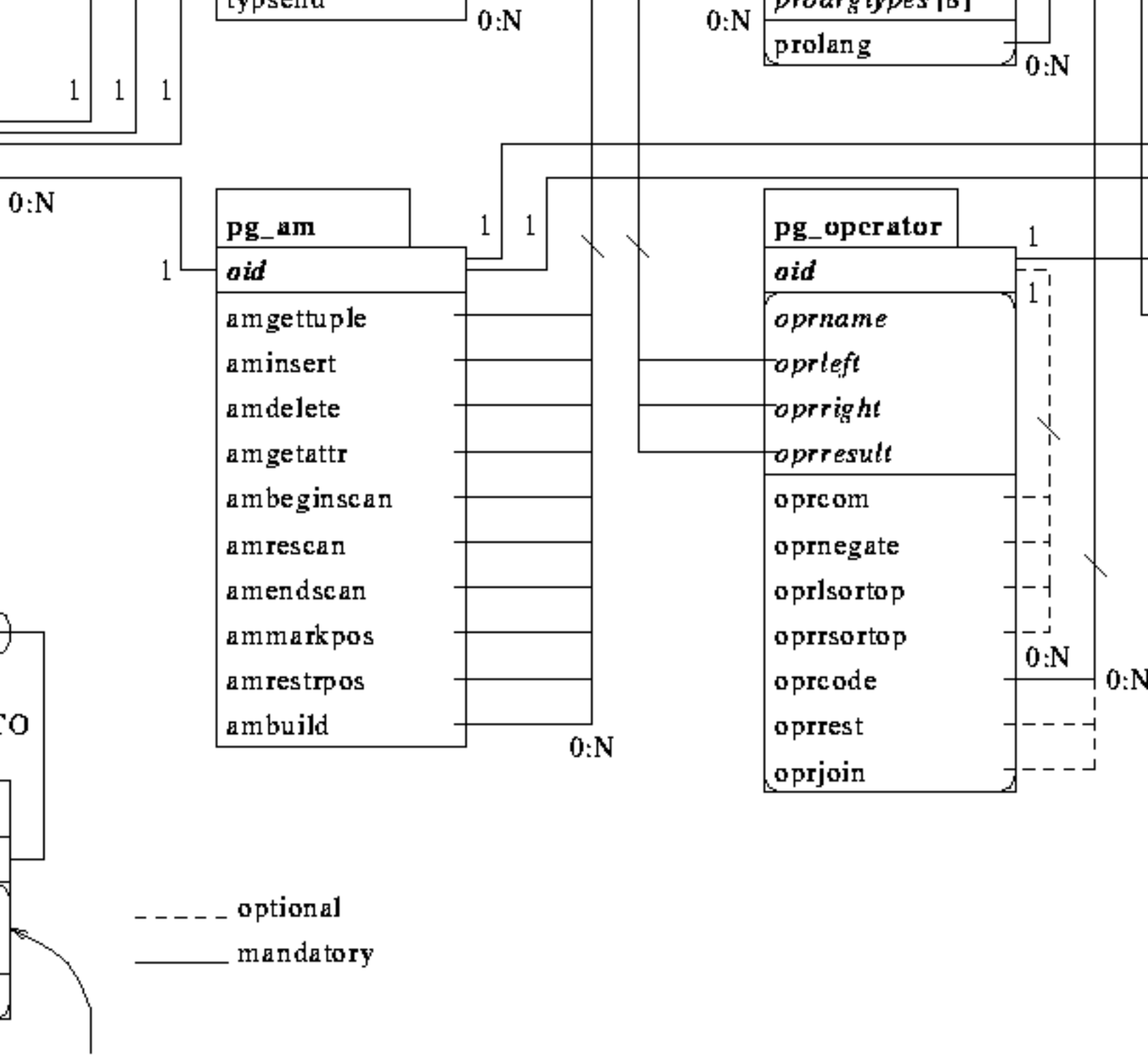
30.3. About the Postgres System Catalogs

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 30.1. 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



indicates these key values are alternate primary keys
(i.e., this class is generally identified by oid but may be
identified by the non-oid primary key in other contexts).

Figure 3. The major POSTGRES system catalogs.

The Reference Manual gives a more detailed explanation of these catalogs and their attributes. However, Figure 30.1 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 31. 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`.

31.1. Query Language (SQL) Functions

31.1.1. 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      |
     +-------+
```

31.1.2. 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\'::char16 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 |
     +--+
```

31.2. Programming Language Functions

31.2.1. 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 the Postgres char16 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"
#include "utils/palloc.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"  /* for char16, etc. */
#include "utils/palloc.h" /* for palloc */
int
add_one(int arg)
{
    return(arg + 1);
}
char16 *
concat16(char16 *arg1, char16 *arg2)
{
    char16 *new_c16 = (char16 *) palloc(sizeof(char16));
```

```
    memset((void *) new_c16, 0, sizeof(char16));
    (void) strncpy(new_c16, arg1, 16);
    return (char16 *)(strncat(new_c16, arg2, 16));
}
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 concat16(char16, char16) RETURNS char16
     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).

31.2.2. 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"  /* for char16, etc. */
#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 char16, 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.

31.2.3. 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 usually a good idea to include that file as well.

- 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 32. 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`.

32.1. User-Defined Types

32.1.1. 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.

32.1.2. 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 33. 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 34. 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 35. The Postgres Rule System

Production rule systems are conceptually simple, but there are many subtle points involved in actually using them. Consequently, we will not attempt to explain the actual syntax and operation of the Postgres rule system here. Instead, you should read ??? to understand some of these points and the theoretical foundations of the Postgres rule system before trying to use rules. The discussion in this section is intended to provide an overview of the Postgres rule system and point the user at helpful references and examples. The "query rewrite" rule system 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 ??? as well as ???.

Chapter 36. 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 30.1. 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 36.1. 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, closin g, 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 36.2. 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 36.3. `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 `oid`s 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 `oid`s 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 `oid`s 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 37. 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:

[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/

```
-- -- there's a memory leak in rtree poly_ops!!
-- -- create index pix2 on polytmp using rtree (p poly_ops);
```

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 38. 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 linkedit it in a special way. This section briefly describes how to perform the compilation and linkediting 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.

38.1. ULTRIX

It is very easy to build dynamically-loaded object files under ULTRIX. ULTRIX does not have any sharedlibrary 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.

38.2. 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!

38.3. 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
                   or
              % 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 39. Triggers

While the current version of Postgres has various client interfaces such as Perl, Tcl, Python and C, it lacks an actual *Procedural Language* (PL). We hope to have a proper PL one day. In the meantime it is 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.

39.1. 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...

39.2. 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.
```

39.3. 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!

39.4. 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 40. Server Programming Interface

Vadim Mikheev

The *Server Programming Interface* (SPI) is an attempt to give users the ability to run SQL queries inside user-defined C functions. Given the lack of a proper *Procedural Language* (PL) in the current version of Postgres, SPI is the only way to write server-stored procedures and triggers. In the future SPI will be used as the "workhorse" for a PL.

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.

40.1. Interface Functions

SPI_connect

SPI_connect — Connects your procedure to the SPI manager.

Synopsis

```
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.

SPI_finish

SPI_finish — Disconnects your procedure from the SPI manager.

Synopsis

```
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.

SPI_exec

SPI_exec — Creates an execution plan (parser+planner+optimizer) and executes a query.

Synopsis

```
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.

SPI_prepare

SPI_prepare — Connects your procedure to the SPI manager.

Synopsis

```
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.

SPI_saveplan

SPI_saveplan — Saves a passed plan

Synopsis

```
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.

SPI_execp

SPI_execp — Executes a plan prepared or returned by SPI_saveplan

Synopsis

```
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.

40.2. Interface Support Functions

All functions described below may be used by connected and unconnected procedures.

SPI_copytuple

SPI_copytuple — Makes copy of tuple in upper Executor context

Synopsis

```
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

SPI_modifytuple

SPI_modifytuple — Modifies tuple of relation

Synopsis

```
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.

SPI_fnumber

SPI_fnumber — Finds the attribute number for specified attribute

Synopsis

```
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.

SPI_fname

SPI_fname — Finds the attribute name for the specified attribute

Synopsis

```
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.

SPI_getvalue

SPI_getvalue — Returns the string value of the specified attribute

Synopsis

```
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.

SPI_getbinval

SPI_getbinval — Returns the binary value of the specified attribute

Synopsis

```
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.

SPI_gettype

SPI_gettype — Returns the type name of the specified attribute

Synopsis

```
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.

SPI_gettypeid

SPI_gettypeid — Returns the type OID of the specified attribute

Synopsis

```
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

SPI_getrelname

SPI_getrelname — Returns the name of the specified relation

Synopsis

```
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.

SPI_palloc

SPI_palloc — Allocates memory in upper Executor context

Synopsis

```
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

SPI_repalloc

SPI_repalloc — Re-allocates memory in upper Executor context

Synopsis

```
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

SPI_pfree

SPI_pfree — Frees memory from upper Executor context

Synopsis

```
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

40.3. 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!

40.4. 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.

40.5. 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 41. libpq

libpq is the application programming interface to Postgres. libpq is a set of library routines which allows client programs to pass queries to the Postgres backend server and to receive the results of these queries. This version of the documentation describes the C interface library. Three short programs are included at the end of this section to show how to write programs that use libpq. There are several 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.

41.1. Control and Initialization

The following environment variables can be used to set up default environment values to avoid hard-coding database names into an application program:

- PGHOST sets the default server name.
- PGOPTIONS sets additional runtime options for the Postgres backend.
- PGPORT sets the default port for communicating with the Postgres backend.
- PGTTY sets the file or tty on which debugging messages from the backend server are displayed.
- PGDATABASE sets the default Postgres database name.
- 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 enabled.

41.2. Database Connection Functions

The following routines deal with making a connection to a backend from a C program.

- 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 is checked. If the environment variable is also not set, then hardwired defaults are used. PQsetdbLogin always returns a valid PGconn pointer. The PQstatus (see below) command should be called to ensure that a connection was properly

made before queries are sent via the connection. libpq 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 as they are subject to change in the future.

- 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.

- PQconndefaults Returns the database name of the connection.

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
 {
  char   *keyword;   /* The keyword of the option */
  char   *environ;   /* Fallback environment variable name */
  char   *compiled;  /* Fallback compiled in default value */
  char   *val;    /* Options 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 options and their current values.

- PQdb Returns the database name of the connection.

```
char *PQdb(PGconn *conn)
```

- PQhost Returns the host name of the connection.

```
char *PQhost(PGconn *conn)
```

- PQoptions Returns the pgoptions used in the connection.

```
char *PQoptions(PGconn *conn)
```

- PQport Returns the pgport of the connection.

```
char *PQport(PGconn *conn)
```

- PQtty Returns the pgtty of the connection.

```
char *PQtty(PGconn *conn)
```

- `PQstatus` Returns the status of the connection. The status can be CONNECTION_OK or CONNECTION_BAD.

  ```
  ConnStatusType *PQstatus(PGconn *conn)
  ```

- `PQerrorMessage` Returns the error message associated with the connection

  ```
  char *PQerrorMessage(PGconn* conn);
  ```

- `PQfinish` Close the connection to the backend. Also frees memory used by the PGconn structure. The PGconn pointer should not be used after PQfinish has been called.

  ```
  void PQfinish(PGconn *conn)
  ```

- `PQreset` Reset the communication port with the backend. This function will close the IPC socket connection to the backend and attempt to reestablish a new connection to the same backend.

  ```
  void PQreset(PGconn *conn)
  ```

- `PQtrace` Enables tracing of messages passed between the frontend and the backend. The messages are echoed to the debug_port file stream.

  ```
  void PQtrace(PGconn *conn,
               FILE* debug_port);
  ```

- `PQuntrace` Disables tracing of messages passed between the frontend and the backend.

  ```
  void PQuntrace(PGconn *conn);
  ```

41.3. Query Execution Functions

- `PQexec` Submit a query to Postgres. Returns a PGresult pointer if the query was successful or a NULL otherwise. If a NULL is returned, PQerrorMessage can be used to get more information about the error.

  ```
  PGresult *PQexec(PGconn *conn,
                   char *query);
  ```

  The `PGresult` structure encapsulates the query result returned by the backend. `libpq` programmers should be careful to maintain the PGresult abstraction. Use the accessor functions described below to retrieve the results of the query. Avoid directly referencing the fields of the PGresult structure as they are subject to change in the future.

- `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 */
  PGRES_TUPLES_OK,  /* the query successfully returned tuples */
  PGRES_COPY_OUT,
  PGRES_COPY_IN,
  PGRES_BAD_RESPONSE, /* an unexpected response was received */
  PGRES_NONFATAL_ERROR,
  PGRES_FATAL_ERROR
  ```

  If the result status is PGRES_TUPLES_OK, then the following routines can be used to retrieve the tuples returned by the query.

- `PQntuples` returns the number of tuples (instances) in the query result.

```
int PQntuples(PGresult *res);
```

- `PQnfields` Returns the number of fields (attributes) in the query result.

```
int PQnfields(PGresult *res);
```

- `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);
```

- `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. If the size returned is -1, the field is a variable length field. Field indices start at 0.

```
int2 PQfsize(PGresult *res,
             int field_index);
```

- `PQgetvalue` Returns the field (attribute) value. For most queries, the value returned by PQgetvalue is a null-terminated ASCII string representation of the attribute value. If the query was a result of a BINARY cursor, then the value returned by PQgetvalue is the binary representation of the type in the internal format of the backend server. It is the programmer's responsibility to cast and convert the data to the correct C type. The value returned by PQgetvalue points to storage that is part of the PGresult structure. One must explicitly copy the value into other storage if it is to be used past the lifetime of the PGresult structure itself.

```
char* PQgetvalue(PGresult *res,
                 int tup_num,
                 int field_num);
```

- `PQgetisnull` Tests a field for a NULL entry.

```
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 known value..

- `PQgetlength` Returns the length of a field (attribute) in bytes. If the field is a struct varlena, the length returned here does not include the size field of the varlena, i.e., it is 4 bytes less.

```
int PQgetlength(PGresult *res,
                int tup_num,
                int field_num);
```

- `PQcmdStatus` Returns the command status associated with the last query command.

```
char *PQcmdStatus(PGresult *res);
```

- `PQcmdTuples` Returns the number of rows affected by the last command.

  ```
  const char *PQcmdTuples(PGresult *res);
  ```

  If the last command was INSERT, UPDATE or DELETE, this returns a string containing the number of rows affected. If the last command was anything else, it returns the empty string.

- `PQoidStatus` Returns a string with the object id of the tuple inserted if the last query is an INSERT command. 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 funtion is intended to replace PQprintTuples(), which is now obsolete.

- `PQprintTuples` Prints out all the tuples and, optionally, the attribute names to the specified output stream. The programs psql and monitor both use PQprintTuples for output.

  ```
  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 properly freed when it is no longer used. Failure to do this will result in memory leaks in the frontend application.

  ```
  void PQclear(PQresult *res);
  ```

41.4. 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.

  ```
  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 load the return value. The caller must have allocated sufficient space to store the return value. The result length will be returned in the storage pointed to by result_len. If the result is to be an integer value, than result_is_int should be set to 1; otherwise it should be set to 0. args and nargs specify the arguments 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 not longer needed.

41.5. Asynchronous Notification

Postgres supports asynchronous notification via the LISTEN and NOTIFY commands. A backend registers its interest in a particular relation with the LISTEN command. All backends listening on a particular relation will be notified asynchronously when a NOTIFY of that relation name is executed by another 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 the relation. `libpq` applications are notified whenever a connected backend has received an asynchronous notification. However, the communication from the backend to the frontend is not asynchronous. Notification comes piggy-backed on other query

results. Thus, an application must submit queries, even empty ones, in order to receive notice of backend notification. In effect, the libpq application must poll the backend to see if there is any pending notification information. After the execution of a query, a frontend may call PQNotifies to see if any notification data is available from the backend.

- PQNotifies returns the notification from a list of unhandled notifications from the backend. Returns NULL if there are no pending notifications from the backend. PQNotifies behaves like the popping of a stack. Once a notification is returned from PQnotifies, it is considered handled and will be removed from the list of notifications.

  ```
  PGnotify* PQNotifies(PGconn *conn);
  ```

  The second sample program gives an example of the use of asynchronous notification.

41.6. 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 full advantage of this capability.

- PQgetline Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer string of size 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 single character ".", which indicates that the backend server has finished sending the results of the copy command. Therefore, if the application ever expects to receive lines that are more than length-1 characters long, the application must be sure to check the return value of PQgetline very carefully. The code in ../src/bin/psql/psql.c contains routines that correctly handle the copy protocol.

  ```
  int PQgetline(PGconn *conn,
                char *string,
                int length)
  ```

- PQputline Sends a null-terminated string to the backend server. The application must explicitly send the single character "." to indicate to the backend that it has finished sending its data.

  ```
  void PQputline(PGconn *conn,
                 char *string);
  ```

- 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);
  ```

  ```
  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);
```

41.7. 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)
  ```

41.8. User Authentication Functions

If the user has generated the appropriate authentication credentials (e.g., obtaining Kerberos tickets), the frontend/backend authentication process is handled by PQexec without any further intervention. The following routines may be called by libpq programs to tailor the behavior of the authentication process.

- 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.

41.9. BUGS

The query buffer is 8192 bytes long, and queries over that length will be silently truncated.

41.10. Sample Programs

41.10.1. 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.0, db
Name);
            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 failed0);
```

```
            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 myportal CURSOR FOR select *
from pg_database");
          if (PQresultStatus(res) != PGRES_COMMAND_OK) {
            fprintf(stderr,"DECLARE CURSOR command failed0);
            PQclear(res);
            exit_nicely(conn);
          }
          PQclear(res);

          res = PQexec(conn,"FETCH ALL in myportal");
          if (PQresultStatus(res) != PGRES_TUPLES_OK) {
            fprintf(stderr,"FETCH ALL command didn't return tuples
properly0);
            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("0);

          /* 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("0);
          }

          PQclear(res);

          /* close the portal */
          res = PQexec(conn, "CLOSE myportal");
          PQclear(res);

          /* end the transaction */
          res = PQexec(conn, "END");
          PQclear(res);

          /* close the connection to the database and cleanup */
          PQfinish(conn);
```

```
        /*   fclose(debug); */
        }
```

41.10.2. 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.0, db
Name);
            fprintf(stderr,"%s",PQerrorMessage(conn));
            exit_nicely(conn);
          }

          res = PQexec(conn, "LISTEN TBL2");
          if (PQresultStatus(res) != PGRES_COMMAND_OK) {
            fprintf(stderr,"LISTEN command failed0);
            PQclear(res);
            exit_nicely(conn);
          }
          /* should PQclear PGresult whenever it is no longer needed
 to avoid
             memory leaks */
          PQclear(res);

          while (1) {
              /* async notification only come back as a result of a
 query*/
              /* we can send empty queries */
              res = PQexec(conn, " ");
        /*      printf("res->status = %s0, pgresStatus[PQresult
Status(res)]); */
              /* check for asynchronous returns */
              notify = PQnotifies(conn);
              if (notify) {
               fprintf(stderr,
                    "ASYNC NOTIFY of '%s' from backend pid '%d'
 received0,
                    notify->relname, notify->be_pid);
               free(notify);
               break;
              }
              PQclear(res);
          }

          /* close the connection to the database and cleanup */
          PQfinish(conn);
```

```
}
```

41.10.3. 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.0, db
Name);
            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 failed0);
            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 failed0);
            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
properly0);
            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] = %d0,
                  i, PQftype(res,i),
                  i, PQfsize(res,i));
          }
          for (i=0; i < PQntuples(res); i++) {
            int *ival;
            float *dval;
            int plen;
            POLYGON* pval;
            /*/
            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: got0, i);
            printf(" i = (%d bytes) %d,0,
                PQgetlength(res,i,i_fnum), *ival);
            printf(" d = (%d bytes) %f,0,
                PQgetlength(res,i,d_fnum), *dval);
            printf(" p = (%d bytes) %d points boundbox = (hi=%f/%f,
lo = %f,%f)0,
                PQgetlength(res,i,d_fnum),
                pval->npts,
                pval->boundbox.xh,
                pval->boundbox.yh,
                pval->boundbox.xl,
                pval->boundbox.yl);
          }

          PQclear(res);

          /* close the portal */
```

```
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

}
```

Part V. Reference

User and programmer interfaces.

Table of Contents

Chapter 42. Functions

Reference information for user-callable functions.

> **Note**
>
> This section needs to be written. Volunteers?

Chapter 43. 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.

43.1. 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.)

43.2. 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.

43.3. 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.

43.3.1. 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);
```

43.3.2. 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.

43.3.3. 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.

43.3.4. 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.

43.3.5. 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.

43.3.6. 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.

43.3.7. 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.

43.4. 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';
```

43.5. 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.

43.6. 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
            }

            /*
             * create the large object
             */
            lobjId = lo_creat(conn, INV_READ|INV_WRITE);
            if (lobjId == 0) {
             fprintf(stderr, "can't create large object");
            }

            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
             }
            }

            (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",
                  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,"0);
            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",
                  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,"0);
            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",
          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
          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
              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_filename0,
                    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.0,
database);
             fprintf(stderr,"%s",PQerrorMessage(conn));
             exit_nicely(conn);
            }

            res = PQexec(conn, "begin");
            PQclear(res);

            printf("importing file
        /*  lobjOid = importFile(conn, in_filename); */
            lobjOid = lo_import(conn, in_filename);
        /*
            printf("as large object %d.0, lobjOid);

            printf("picking out bytes 1000-2000 of the large object0)
;
            pickout(conn, lobjOid, 1000, 1000);

            printf("overwriting bytes 1000-2000 of the large object
with X's0);
            overwrite(conn, lobjOid, 1000, 1000);
        */

            printf("exporting large object to file
        /*    exportFile(conn, lobjOid, out_filename); */
            lo_export(conn, lobjOid,out_filename);
```

```
    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Chapter 44. 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.

44.1. 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. Most embedded SQL preprocessors I have seen and heard of make extensions so it is difficult to obtain portability between them anyway. I have not read the standard but I hope that my implementation does not deviate too much and that it would be possible to port programs with embedded SQL written for other RDBMS packages to Postgres and thus promoting the spirit of free software.

44.2. 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.

[1] mailto:linus@epact.se
[2] mailto:meskes@debian.org

44.3. How To Use egpc

This section describes how to use the egpc tool.

44.3.1. Preprocessor

The preprocessor is called ecpg. After installation it resides in the Postgres `bin/` directory.

44.3.2. 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, FILE *stream)

If this is called, with the first argument non-zero, then debuglogging is turned on. Debuglogging 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.

44.3.3. 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.

44.4. 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.

44.5. Porting From Other RDBMS Packages

To be written by persons that knows the different RDBMS packages and that actually does port something...

44.6. Installation

Since version 0.5 ecpg is distributed together with Postgres. So you should get your precompiler, libraries and header files compiled and installed on the fly.

44.7. For the Developer

This section is for those that wants 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.

44.7.1. ToDo List

This version the preprocessor has some flaws:

Preprocessor output

The variables should be static.

Preprocessor cannot do syntax checking on your SQL statements

Whatever you write is copied more or less exactly to the Postgres and you will not be able to locate your errors until run-time.

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

Possibility to define records or structures in the declare section in a way that the record can be filled from one row in the database.

This is a simpler way to handle an entire row at a time.

array operations

Oracle has array operations that enhances speed. When implementing it in ecpg it is done for compatibility reasons only. For them to improve speed would require a lot more insight in the Postgres internal mechanisms than I possess.

indicator variables

Oracle has indicator variables that tell if a value is `null` or if it is empty. This largely simplifies array operations and provides for a way to hack around some design flaws in the handling of `VARCHAR2` (like that an empty string isn't distinguishable from a `null` value). I am not sure if this is an Oracle extension or part of the ANSI standard.

typedefs

As well as complex types like records and arrays, typedefs would be a good thing to take care of.

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.

44.7.2. 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.

44.7.3. 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.)

44.7.4. 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 45. libpq

libpq is the application programming interface to Postgres. libpq is a set of library routines which allows client programs to pass queries to the Postgres backend server and to receive the results of these queries. This version of the documentation describes the C interface library. Three short programs are included at the end of this section to show how to write programs that use libpq. There are several 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.

45.1. Control and Initialization

The following environment variables can be used to set up default environment values to avoid hard-coding database names into an application program:

- PGHOST sets the default server name.
- PGOPTIONS sets additional runtime options for the Postgres backend.
- PGPORT sets the default port for communicating with the Postgres backend.
- PGTTY sets the file or tty on which debugging messages from the backend server are displayed.
- PGDATABASE sets the default Postgres database name.
- 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 enabled.

45.2. Database Connection Functions

The following routines deal with making a connection to a backend from a C program.

- 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 is checked. If the environment variable is also not set, then hardwired defaults are used. PQsetdbLogin always returns a valid PGconn pointer. The PQstatus (see below) command should be called to ensure that a connection was properly

made before queries are sent via the connection. libpq 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 as they are subject to change in the future.

- 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.

- PQconndefaults Returns the database name of the connection.

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
 {
  char   *keyword;   /* The keyword of the option */
  char   *environ;   /* Fallback environment variable name */
  char   *compiled;  /* Fallback compiled in default value */
  char   *val;    /* Options 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 options and their current values.

- PQdb Returns the database name of the connection.

```
char *PQdb(PGconn *conn)
```

- PQhost Returns the host name of the connection.

```
char *PQhost(PGconn *conn)
```

- PQoptions Returns the pgoptions used in the connection.

```
char *PQoptions(PGconn *conn)
```

- PQport Returns the pgport of the connection.

```
char *PQport(PGconn *conn)
```

- PQtty Returns the pgtty of the connection.

```
char *PQtty(PGconn *conn)
```

- `PQstatus` Returns the status of the connection. The status can be CONNECTION_OK or CONNECTION_BAD.

  ```
  ConnStatusType *PQstatus(PGconn *conn)
  ```

- `PQerrorMessage` Returns the error message associated with the connection

  ```
  char *PQerrorMessage(PGconn* conn);
  ```

- `PQfinish` Close the connection to the backend. Also frees memory used by the PGconn structure. The PGconn pointer should not be used after PQfinish has been called.

  ```
  void PQfinish(PGconn *conn)
  ```

- `PQreset` Reset the communication port with the backend. This function will close the IPC socket connection to the backend and attempt to reestablish a new connection to the same backend.

  ```
  void PQreset(PGconn *conn)
  ```

- `PQtrace` Enables tracing of messages passed between the frontend and the backend. The messages are echoed to the debug_port file stream.

  ```
  void PQtrace(PGconn *conn,
               FILE* debug_port);
  ```

- `PQuntrace` Disables tracing of messages passed between the frontend and the backend.

  ```
  void PQuntrace(PGconn *conn);
  ```

45.3. Query Execution Functions

- `PQexec` Submit a query to Postgres. Returns a PGresult pointer if the query was successful or a NULL otherwise. If a NULL is returned, PQerrorMessage can be used to get more information about the error.

  ```
  PGresult *PQexec(PGconn *conn,
                   char *query);
  ```

  The `PGresult` structure encapsulates the query result returned by the backend. `libpq` programmers should be careful to maintain the PGresult abstraction. Use the accessor functions described below to retrieve the results of the query. Avoid directly referencing the fields of the PGresult structure as they are subject to change in the future.

- `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 */
  PGRES_TUPLES_OK,  /* the query successfully returned tuples */
  PGRES_COPY_OUT,
  PGRES_COPY_IN,
  PGRES_BAD_RESPONSE, /* an unexpected response was received */
  PGRES_NONFATAL_ERROR,
  PGRES_FATAL_ERROR
  ```

  If the result status is PGRES_TUPLES_OK, then the following routines can be used to retrieve the tuples returned by the query.

- `PQntuples` returns the number of tuples (instances) in the query result.

```
int PQntuples(PGresult *res);
```

- `PQnfields` Returns the number of fields (attributes) in the query result.

```
int PQnfields(PGresult *res);
```

- `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);
```

- `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. If the size returned is -1, the field is a variable length field. Field indices start at 0.

```
int2 PQfsize(PGresult *res,
             int field_index);
```

- `PQgetvalue` Returns the field (attribute) value. For most queries, the value returned by PQgetvalue is a null-terminated ASCII string representation of the attribute value. If the query was a result of a BINARY cursor, then the value returned by PQgetvalue is the binary representation of the type in the internal format of the backend server. It is the programmer's responsibility to cast and convert the data to the correct C type. The value returned by PQgetvalue points to storage that is part of the PGresult structure. One must explicitly copy the value into other storage if it is to be used past the lifetime of the PGresult structure itself.

```
char* PQgetvalue(PGresult *res,
                 int tup_num,
                 int field_num);
```

- `PQgetisnull` Tests a field for a NULL entry.

```
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 known value..

- `PQgetlength` Returns the length of a field (attribute) in bytes. If the field is a struct varlena, the length returned here does not include the size field of the varlena, i.e., it is 4 bytes less.

```
int PQgetlength(PGresult *res,
                int tup_num,
                int field_num);
```

- `PQcmdStatus` Returns the command status associated with the last query command.

```
char *PQcmdStatus(PGresult *res);
```

- `PQcmdTuples` Returns the number of rows affected by the last command.

  ```
  const char *PQcmdTuples(PGresult *res);
  ```

  If the last command was INSERT, UPDATE or DELETE, this returns a string containing the number of rows affected. If the last command was anything else, it returns the empty string.

- `PQoidStatus` Returns a string with the object id of the tuple inserted if the last query is an INSERT command. 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 funtion is intended to replace PQprintTuples(), which is now obsolete.

- `PQprintTuples` Prints out all the tuples and, optionally, the attribute names to the specified output stream. The programs psql and monitor both use PQprintTuples for output.

  ```
  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 properly freed when it is no longer used. Failure to do this will result in memory leaks in the frontend application.

  ```
  void PQclear(PQresult *res);
  ```

45.4. 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.

  ```
  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 load the return value. The caller must have allocated sufficient space to store the return value. The result length will be returned in the storage pointed to by result_len. If the result is to be an integer value, than result_is_int should be set to 1; otherwise it should be set to 0. args and nargs specify the arguments 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 not longer needed.

45.5. Asynchronous Notification

Postgres supports asynchronous notification via the LISTEN and NOTIFY commands. A backend registers its interest in a particular relation with the LISTEN command. All backends listening on a particular relation will be notified asynchronously when a NOTIFY of that relation name is executed by another 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 the relation. `libpq` applications are notified whenever a connected backend has received an asynchronous notification. However, the communication from the backend to the frontend is not asynchronous. Notification comes piggy-backed on other query

results. Thus, an application must submit queries, even empty ones, in order to receive notice of backend notification. In effect, the libpq application must poll the backend to see if there is any pending notification information. After the execution of a query, a frontend may call PQNotifies to see if any notification data is available from the backend.

- PQNotifies returns the notification from a list of unhandled notifications from the backend. Returns NULL if there are no pending notifications from the backend. PQNotifies behaves like the popping of a stack. Once a notification is returned from PQnotifies, it is considered handled and will be removed from the list of notifications.

  ```
  PGnotify* PQNotifies(PGconn *conn);
  ```

  The second sample program gives an example of the use of asynchronous notification.

45.6. 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 full advantage of this capability.

- PQgetline Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer string of size 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 single character ".", which indicates that the backend server has finished sending the results of the copy command. Therefore, if the application ever expects to receive lines that are more than length-1 characters long, the application must be sure to check the return value of PQgetline very carefully. The code in ../src/bin/psql/psql.c contains routines that correctly handle the copy protocol.

  ```
  int PQgetline(PGconn *conn,
                char *string,
                int length)
  ```

- PQputline Sends a null-terminated string to the backend server. The application must explicitly send the single character "." to indicate to the backend that it has finished sending its data.

  ```
  void PQputline(PGconn *conn,
                 char *string);
  ```

- 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);
  ```

  ```
  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);
```

45.7. 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)
  ```

45.8. User Authentication Functions

If the user has generated the appropriate authentication credentials (e.g., obtaining Kerberos tickets), the frontend/backend authentication process is handled by PQexec without any further intervention. The following routines may be called by libpq programs to tailor the behavior of the authentication process.

- 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.

45.9. BUGS

The query buffer is 8192 bytes long, and queries over that length will be silently truncated.

45.10. Sample Programs

45.10.1. 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.0, db
Name);
            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 failed0);
```

```
            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 myportal CURSOR FOR select *
from pg_database");
          if (PQresultStatus(res) != PGRES_COMMAND_OK) {
            fprintf(stderr,"DECLARE CURSOR command failed0);
            PQclear(res);
            exit_nicely(conn);
          }
          PQclear(res);

          res = PQexec(conn,"FETCH ALL in myportal");
          if (PQresultStatus(res) != PGRES_TUPLES_OK) {
            fprintf(stderr,"FETCH ALL command didn't return tuples
properly0);
            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("0);

          /* 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("0);
          }

          PQclear(res);

          /* close the portal */
          res = PQexec(conn, "CLOSE myportal");
          PQclear(res);

          /* end the transaction */
          res = PQexec(conn, "END");
          PQclear(res);

          /* close the connection to the database and cleanup */
          PQfinish(conn);
```

```
        /*   fclose(debug); */
        }
```

45.10.2. 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.0, db
Name);
            fprintf(stderr,"%s",PQerrorMessage(conn));
            exit_nicely(conn);
          }

          res = PQexec(conn, "LISTEN TBL2");
          if (PQresultStatus(res) != PGRES_COMMAND_OK) {
            fprintf(stderr,"LISTEN command failed0);
            PQclear(res);
            exit_nicely(conn);
          }
          /* should PQclear PGresult whenever it is no longer needed
 to avoid
             memory leaks */
          PQclear(res);

          while (1) {
              /* async notification only come back as a result of a
 query*/
              /* we can send empty queries */
              res = PQexec(conn, " ");
        /*      printf("res->status = %s0, pgresStatus[PQresult
Status(res)]); */
              /* check for asynchronous returns */
              notify = PQnotifies(conn);
              if (notify) {
               fprintf(stderr,
                    "ASYNC NOTIFY of '%s' from backend pid '%d'
 received0,
                    notify->relname, notify->be_pid);
               free(notify);
               break;
              }
              PQclear(res);
          }

          /* close the connection to the database and cleanup */
          PQfinish(conn);
```

```
}
```

45.10.3. 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.0, db
  Name);
              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 failed0);
              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 failed0);
              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
properly0);
            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] = %d0,
                  i, PQftype(res,i),
                  i, PQfsize(res,i));
          }
          for (i=0; i < PQntuples(res); i++) {
            int *ival;
            float *dval;
            int plen;
            POLYGON* pval;
            /*/
            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: got0, i);
            printf(" i = (%d bytes) %d,0,
                PQgetlength(res,i,i_fnum), *ival);
            printf(" d = (%d bytes) %f,0,
                PQgetlength(res,i,d_fnum), *dval);
            printf(" p = (%d bytes) %d points boundbox = (hi=%f/%f,
lo = %f,%f)0,
                PQgetlength(res,i,d_fnum),
                pval->npts,
                pval->boundbox.xh,
                pval->boundbox.yh,
                pval->boundbox.xl,
                pval->boundbox.yl);
          }

          PQclear(res);

          /* close the portal */
```

```
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

}
```

Chapter 46. pgtcl

pgtcl is a tcl package for front-end programs to interface with Postgres backends. pgtcl does not use the libpq library but communicates to the backend directly via the frontend-backend protocol. Thus, it is more efficient than previous postgres->tcl bindings which are layered on top of libpq. In addition, pgtcl can handle multiple backend connections from a single frontend application.

This package was originally written by Jolly Chen.

46.1. Commands

The pg_lo* routines are interfaces to the Inversion Large Objects in Postgres. The functions are designed to mimic the analogous file system functions in the standard Unix file system interface.

Table 46.1. PGTCL Commands

Command	Description
pg_connect	opens a connection to the backend server
pg_disconnect	closes a connection
pg_exec	send a query to the backend
pg_select	loop over the result of a select statement
pg_result	manipulate the results of a query
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 on 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

Some commands equivalent to libpq commands are provided for connection and query operations.

The pg_lo* routines should typically 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.

46.2. 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_disconnect $conn
    return $datnames
}
```

46.3. Reference Information

pg_connect

pg_connect — opens a connection to the backend server

Synopsis

```
pg_connect dbName [-host hostName]
  [-port portNumber] [-tty pqtty] [-options optionalBackendArgs]
```

Inputs

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*]

(need information thomas 1997-12-24)

[-options *optionalBackendArgs*]

Specifies options for the backend server for *dbName*.

Outputs

dbHandle

The return result is either an error message or a handle for a database connection. Handles start with the prefix "pgp"

Description

`pg_connect` opens a connection to the Postgres backend.

Usage

XXX thomas 1997-12-24

pg_disconnect

pg_disconnect — closes a connection to the backend server

Synopsis

```
pg_disconnect dbHandle
```

Inputs

dbHandle

Specifies a valid database handle.

Outputs

None

Description

`pg_disconnect` closes a connection to the Postgres backend.

pg_exec

pg_exec — send a query string to the backend

Synopsis

```
pg_exec dbHandle queryString
```

Inputs

dbHandle

Specifies a valid database handle.

queryString

Specifies a valid SQL query.

Outputs

queryHandle

the return result is either an error message or a handle for a query result.

Description

`pg_exec` submits a query to the Postgres backend and returns a result. Handles start with the prefix "pgp".

pg_select

pg_select — loop over the result of a select statement

Synopsis

```
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

queryHandle

the return result is either an error message or a handle for a query result.

Description

`pg_select` submits a query to the Postgres backend. and returns the results. The *`queryString`* must be a select statement. Anything else returns an error. The *`arrayVar`* variable is an array name used in the loop. It is filled out with the result of the query for each tuple using the field names as the associative indices.

Usage

```
set DB "mydb"
set conn [pg_connect $DB]
pg_select $conn "SELECT * from table" array {
 puts [format "%5d %s" array(control) array(name)]
}
pg_disconnect $conn
```

pg_result

pg_result — get information about a query result

Synopsis

```
pg_result queryHandle resultOption
```

Inputs

queryHandle

The handle for a query result.

resultOption

Specifies one of several possible options.

Options

-status

the status of the result.

-oid

if the last query was an insert, returns the oid of the inserted tuple

-conn

the connection that produced the result

-assign arrayName

assign the results to an array

-numTuples

the number of tuples in the query

-attributes

returns a list of the name/type pairs of the tuple attributes

-getTuple tupleNumber

returns the values of the tuple in a list

-clear

clear the result buffer. Do not reuse after this

Outputs

queryHandle

the return result is either an error message or a handle for a query result.

Description

`pg_result` returns information about a query.

pg_lo_creat

pg_lo_creat — create a large object

Synopsis

```
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"]
```

pg_lo_open

pg_lo_open — open a large object

Synopsis

```
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".

pg_lo_close

pg_lo_close — close a large object

Synopsis

```
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

pg_lo_read

pg_lo_read — read a large object

Synopsis

```
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.

pg_lo_write

pg_lo_write — write a large object

Synopsis

```
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.

pg_lo_lseek

pg_lo_lseek — seek to a position on a large object

Synopsis

`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".

pg_lo_tell

pg_lo_tell — return the current seek position of a large object

Synopsis

`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

pg_lo_unlink

pg_lo_unlink — delete a large object

Synopsis

```
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

pg_lo_import

pg_lo_import — import a large object from a Unix file

Synopsis

```
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.

pg_lo_export

pg_lo_export — export a large object to a Unix file

Synopsis

```
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 47. ODBC Interface

Tim Goeke

> **Note**
>
> Contributed by Tim Goeke[1]

ODBC is an abstract API which allows you to write standard "ODBC" code, using the ODBC API.

47.1. 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 RDBMS.

The backend access come from ODBC drivers, or vendor specifc drivers that allow data access. PostODBC 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 PostgreSQL servers which have exactly the same data. Using ODBC, your Windows app would make exactly the same calls and the back end data source would look the same (to the windows app).

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 and MFC 4.2 class. This is the easiest route if you are doing Windows C++ development under Windows NT.

If I write an app for PostgreSQL can I write it using ODBC calls to the PostgreSQL server, or is that only when another database program like MS SQL Server or Access needs to access the data?

Again, the ODBC API set is the way to go. You can find out more at Microsoft's web site or in your Visual C++ docs (if that's what you are using.)

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.

[1] mailto:tgoeke@xpressway.com

Tip

The PostgreSQL datetime type will break MS Access.

Chapter 48. JDBC Interface

There is a JDBC interface available for Postgres. It is documented elsewhere using the accepted tool for Java-language code.

Part VI. Developer's Guide

The Developer's Guide includes discussion of design decisions and suggestions for future development.

Table of Contents

Chapter 49. Architecture

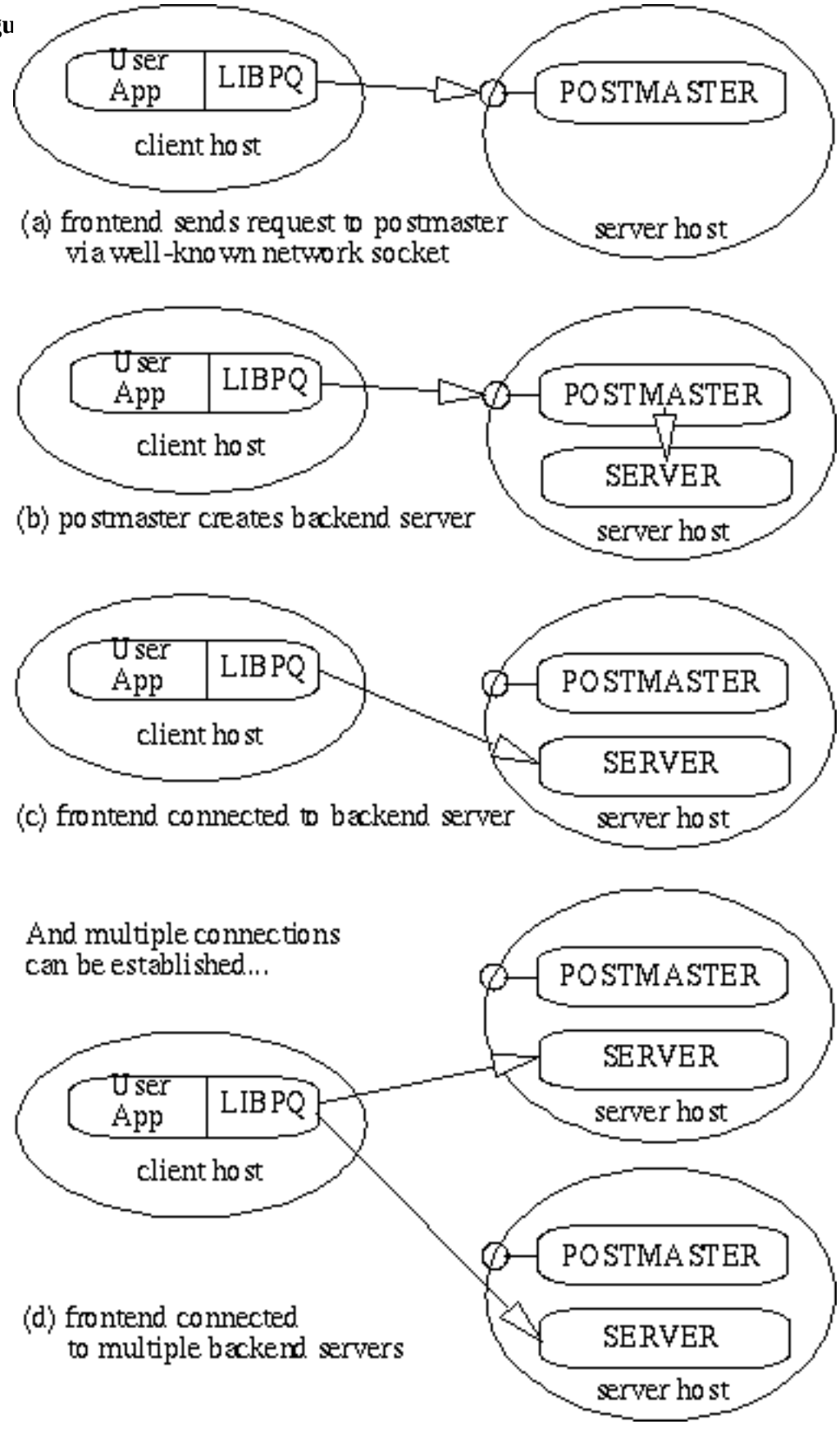
49.1. Postgres Architectural Concepts

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 (Figure 49.1(a)), which in turn starts a new backend server process (Figure 49.1(b))

Figu


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

and connects the frontend process to the new server (Figure 49.1(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 50. 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.

50.1. 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 `join`s 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 `join`s 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.

50.2. 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                          |
+===+=====================================+
```

50.3. 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).

50.4. Future Implementation Tasks for Postgres GEQO

50.4.1. Basic Improvements

50.4.1.1. 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.

50.4.1.2. 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

50.4.1.3. 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.

50.4.1.4. 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 :-(

50.4.2. 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.

FAQ in comp.ai.genetic[1] is available at Encore[2].

The Design and Implementation of the Postgres Query Optimizer. Z. Fong. University of California, Berkeley Computer Science Department.

File `planner/Report.ps` in the 'postgres-papers' distribution.

Fundamentals of Database Systems. R. Elmasri and S. Navathe. The Benjamin/Cummings Pub., Inc..

[1] news://comp.ai.genetic
[2] ftp://ftp.Germany.EU.net/pub/research/softcomp/EC/Welcome.html

Chapter 51. Frontend/Backend Protocol

Phil Thompson

> **Note**
>
> Written by Phil Thompson[1]

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 the initial version-numbered protocol, designated v1.0. Higher level features built on this protocol (for example, how `libpq` passes certain environment variables after the connection is established) are covered elsewhere.

51.1. 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 to a backend.

Subsequent communications are query and result packets exchanged between the frontend and the backend. The postmaster takes no further part in the communication.

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 send from a frontend to the postmaster, which comprise a packet length then the packet itself. The difference is historical.

51.2. Protocol

This section describes the message flow. There are four different types of flows depending on the state of the connection: authentication, query, function call, and termination.

[1] mailto:phil@river-bank.demon.co.uk

51.2.1. Authentication

The frontend sends a StartupPacket. The postmaster uses this 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.

51.2.2. Query

The frontend sends a Query message to the backend. The response sent by the backend depends on the contents of the query. The possible responses are as follows.

CompletedResponse

The query 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 if 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

The query was empty.

ErrorResponse

An error has occurred.

NoticeResponse

A warning message has been issued in relation to the query. Notices are in addition to other responses, ie. the backend will send another response message immediately afterwards.

NotificationResponse

A notify(l) command has been executed for a relation for which a previous listen(l) command was executed. Notifications are in addition to other responses, ie. the backend will send another response message immediately afterwards.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message.

51.2.3. Function Call

The frontend sends a FunctionCall message to the backend. The response sent by the backend depends on the result of the function call. The possible responses are as follows.

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.

NoticeResponse

A warning message has been issued in relation to the function call. Notices are in addition to other responses, ie. the backend will send another response message immediately afterwards.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message.

51.2.4. Termination

The frontend sends a Terminate message and immediately closes the connection. On receipt of the message, the backend immediately closes the connection and terminates.

51.3. 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 it's 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').

51.4. 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, in the context in which it is sent (see CopyInResponse), as an ASCII row.

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 of the 1st byte, the 9th field corresponds to bit 8 of the 2nd byte, and so on. The bit is set if the value of the corresponding field is not NULL.

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.

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.

BinaryRow (B)

Byte1('B')

Identifies the message, in the context in which it is sent (see CopyOutResponse), as a binary row.

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 of the 1st byte, the 9th field corresponds to bit 8 of the 2nd byte, and so on. The bit is set if the value of the corresponding field is not NULL.

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.

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('D')

Identifies the message, in the context in which it is sent (see AsciiRow), as a copy in started response.

CopyOutResponse (B)

Byte1('B')

Identifies the message, in the context in which it is sent (see BinaryRow), as a copy out started response.

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 an empty query response.

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 an actual 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 no actual 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 backend process.

String

The name of the relation that the notify has been raised on.

Query (F)

Byte1('Q')

Identifies the message as query.

String

The query itself.

RowDescription (B)

Byte1('T')

Identifies the message as a row description.

Int16

Specifies the number of fields in a row (and 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.

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 omitted.

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 52. 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

Part VII. Appendices

Additional related information.

Table of Contents

Appendix A. Documentation

Thomas Lockhart

Postgres documentation is written using the *Standard Generalized Markup Language* (SGML) DocBook[1] *Document Type Definition* (DTD).

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.

A.1. Introduction

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 to 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.2. 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. Building Documentation

HTML documentation packages can be generated from the SGML source by typing

[1] http://www.ora.com/davenport/
[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/

```
% 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.4. Toolsets

A.4.1. jade

The current stable release of jade is version 1.0.1.

A.4.1.1. Installation for Linux

Install RPMs[7] for jade and related packages.

A.4.1.2. Installation for non-Linux Platforms

There are some other packaged distributions for jade. FreeBSD seems to have one available. Please report package status to the docs mailing list and we will include that information here.

For other platforms, install sources[8] for jade and related packages and build from scratch.

A.4.2. Modular Style Sheets

The current stable release of the Modular Style Sheets is version 1.0.7.

A.5. Hardcopy Generation for v6.3

The hardcopy Postscript documentation is generated by converting the SGML source code to RTF, then importing into Applixware. 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 -v -geometry 400x400'>' figure03.gif con.gif
% convert -v -crop 400x380 con.gif connections.gif
```

A.5.1. RTF Cleanup Procedure

Several items must be addressed in generating Postscript hardcopy:

[7] ftp://ftp.cygnus.com/pub/home/rosalia/
[8] ftp://ftp.cygnus.com/pub/eichin/docware/SOURCES/

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. Alternate Toolsets

The current stable release of sgml-tools is version 1.0.4. The v1.0 release includes some restructuring of the directory tree to more easily support additional document styles, possibly including DocBook. The only version of sgml-tools evaluated for Postgres was v0.99.0.

A.6.1. sgml-tools

Install sgml-tools-0.99.0

Apply sgml-tools-patches[9] to the linuxdoc styles. These patches fix small problems with table formatting and with figure file names on conversion to postscript or html.

[9] http://alumni.caltech.edu/~lockhart/postgres/linuxdoc/sgml-tools-patches-0.99.0.tar.gz

A.6.2. sgml2latex

The current stable release of sgml2latex is version 1.4. I have misplaced the original reference for this package, so will temporarily post it with this example.

Install sgml2latex[10].

A.6.3. latex

Get and install texmf, teTeX, or another package providing full tex/latex functionality.

Add the required styles[11] linuxdoc-sgml.sty, linuxdoc-sgml-a4.sty isolatin.sty, qwertz.sty, and null.sty to texmf/tex/latex/tools/ or the appropriate area.

```
% cat latex-styles-0.99.0.tar.gz | (cd texmf/tex/latex/tools/; tar
 zxvf -)
```

Run texhash to update the tex database.

[10] http://alumni.caltech.edu/~lockhart/postgres/linuxdoc/sgml2latex-format.1.4.tar.gz
[11] http://alumni.caltech.edu/~lockhart/postgres/linuxdoc/latex-styles-0.99.0.tar.gz

Appendix B. Contacts

B.1. Introduction

B.2. People

- 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.

[bowman93] *The Practical SQL Handbook*. Using Structured Query Language. 3. Judity Bowman, Sandra Emerson, and Marcy Damovsky. ISBN 0-201-44787-8. 1996. Addison-Wesley. Copyright © 1997 Addison-Wesley Longman, Inc..

[date97] *A Guide to The SQL Standard*. The SQL Standard. A user's guide to the standard database language SQL. 4. C. J. Date and Hugh Darwen. ISBN 0-201-96426-0. 1997. Addison-Wesley. Copyright © 1997 Addison-Wesley Longman, Inc..

[melt93] *Understanding the New SQL*. A complete guide. Jim Melton and Alan R. Simon. ISBN 1-55860-245-3. 1993. Morgan Kaufmann. Copyright © 1993 Morgan Kaufmann Publishers, Inc..

Abstract

Accessible reference for SQL features.

PostgreSQL-Specific Documentation

This section is for related documentation.

[admin-guide] *The PostgreSQL. Administrator's Guide*. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[programmers-guide] *The PostgreSQL. Programmer's Guide*. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[tutorial] *The PostgreSQL. Reference Manual*. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[users-guide] *The PostgreSQL. Tutorial Introduction*. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[reference] *The PostgreSQL. User's Guide*. Thomas Lockhart. 1998-03-01. The PostgreSQL Global Development Group.

[yu95] *The Postgres95. User Manual*. YU95. A. Yu and J. Chen. . Sept. 5, 1995. University of California, Berkeley CA.

Proceedings and Articles

This section is for articles and newsletters.

[ong90] *A Unified Framework for Version Modeling Using Production Rules in a Database System*. ONG90. L. Ong and J. Goh. April, 1990. ERL Technical Memorandum M90/33. University of California, Berkeley CA.

[rowe87] *The Postgres. Data Model*. ROWE87. L. Rowe and M. Stonebraker. VLDB Conference, Brighton, England, Sept. 1987.

[ston86] *The Design of Postgres.* . STON86. M. Stonebraker and L. Rowe. Conference on Management of Data, Washington DC, May 1986.

[ston87a] *The Design of the Postgres. Rules System*. STON87a. M. Stonebraker, E. Hanson, and C. H. Hong. Conference on Data Engineering, Los Angeles, CA, Feb. 1987.

[ston87b] *The Postgres. Storage System*. STON87b. M. Stonebraker. VLDB Conference, Brighton, England, Sept. 1987.

[ston89] *A Commentary on the Postgres. Rules System*. STON89. M. Stonebraker, M. Hearst, and S. Potamianos. Record 18(3), Sept. 1989.

[ston90a] *The Implementation of Postgres.* . STON90a. M. Stonebraker, L. A. Rowe, and M. Hirohama. Transactions on Knowledge and Data Engineering 2(1), March 1990.

[ston90b] *On Rules, Procedures, Caching and Views in Database Systems*. STON90b. M. Stonebraker and et. al.. Conference on Management of Data, June 1990.

Index

P

S