The POSTGRES95 User Manual

Beta Release (5/1/95)

Andrew Yu and Jolly Chen
(with the POSTGRES Group)
Computer Science Div., Dept. of EECS
University of California at Berkeley

1. INTRODUCTION

This document is the user manual for the POSTGRES95 database management system developed at the University of California at Berkeley. POSTGRES95 is based on POSTGRES release 4.2. 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, each attribute of which has a specific type. In current commercial systems, possible types including 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 succeeded in replacing previous models in part because of its simplicity. However, as mentioned, the "Spartan simplicity" of the relational model often makes the implementation of certain applications very difficult. The POSTGRES data model offers substantial additional power by incorporating the following four additional basic constructs:

classes
inheritance
types
functions

in such a way that users can easily extend the system. In addition, POSTGRES supports a powerful production rule system.

1.2. A Short History of the 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 "demo-ware" 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 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 informatics database and several geographic information systems. POSTGRES has also been used as an educational tool at several universities. Finally, Illustra Information Technologies has picked up the prototype code and commercialized it.

POSTGRES became the primary data manager for the Sequoia 2000 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.

1.3. What is POSTGRES95?

POSTGRES95 is a derivative of the last official release of POSTGRES (version 4.2). The code has been ansi-fied and trimmed by 25%. There are a lot of internal changes that improve performance and code maintainability. POSTGRES95 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 support subqueries (which can be imitated with user defined SQL functions) at the moment. 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 four 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.
- gmake (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).

1.4. About This Release

POSTGRES95 is available free of charge. This manual describes the beta release of POSTGRES95. The authors have compiled and tested POSTGRES95 on the following platforms:

architecture	processor	operating system
DECstation 3000	Alpha AXP	OSF/1 2.0, 3.0
DECstation 5000	MIPS	ULTRIX 4.4
Sun4	SPARC	SunOS 4.1.3, 4.1.3_U1; Solaris 2.4
H-P 9000/700 and 800	PA-RISC	HP-UX 9.00, 9.01, 9.03

1.5. Outline of This Manual

From now on, We will use POSTGRES to mean POSTGRES95. The first part of this manual goes over some basic system concepts and procedures for starting the POSTGRES system. We then turn to a tutorial overview of the POSTQUEL data model and query language, introducing a few of its advanced features. Next, we explain 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, the manual concludes with a detailed appendix that discusses some of the more involved and operating system-specific procedures involved in extending the system.

We assume proficiency with UNIX and C programming.

UNIX is a trademark of X/Open, Ltd. Sun4, SPARC, SunOS and Solaris are trademarks of Sun Microsystems, Inc. DEC, DECstation, Alpha AXP and ULTRIX are trademarks of Digital Equipment Corp. PA-RISC and HP-UX are trademarks of Hewlett-Packard Co. OSF/1 is a trademark of the Open Systems Foundation.

2. POSTGRES ARCHITECTURE 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 three cooperating UNIX processes (programs):

- A supervisory daemon process (the `postmaster`),
- the user’s frontend application (e.g., the `monitor` program), and

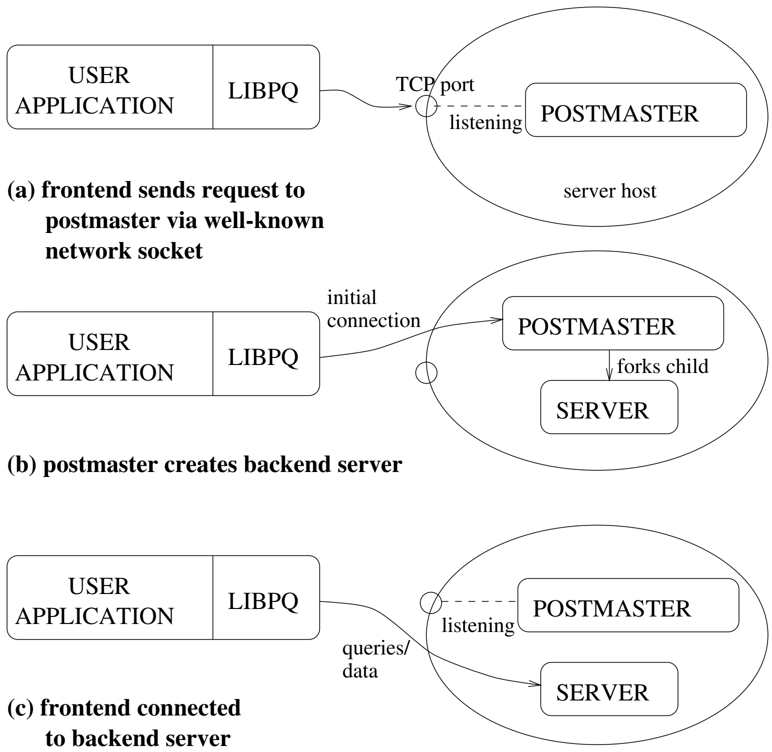

Figure 1. How a connection is established.

• the backend database server (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 LIBPQ library. The library forwards the user requests over the network to the `postmaster` (Figure 1(a)), which in turn starts a new backend server process (Figure 1(b)) and connects the frontend process to its server (Figure 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 the frontend and backend processes come and go.

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 or may not be running on a separate machine (e.g., a client workstation). You should keep this in mind, because this means that the files that you can access on your 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 the `postgres` server 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.

3. GETTING STARTED WITH POSTGRES

This section discusses how to start POSTGRES and set up your own environment so that you can use frontend applications. We assume POSTGRES has already been successfully installed. (Refer to the installation notes for how to install POSTGRES.)

Some of the steps listed in this section will apply to all POSTGRES users, and some will apply primarily to 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.

In this section, items for end users are labelled “User” and items intended for the site administrator are labelled “Admin.”

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. Admin/User: Setting Up Your Environment

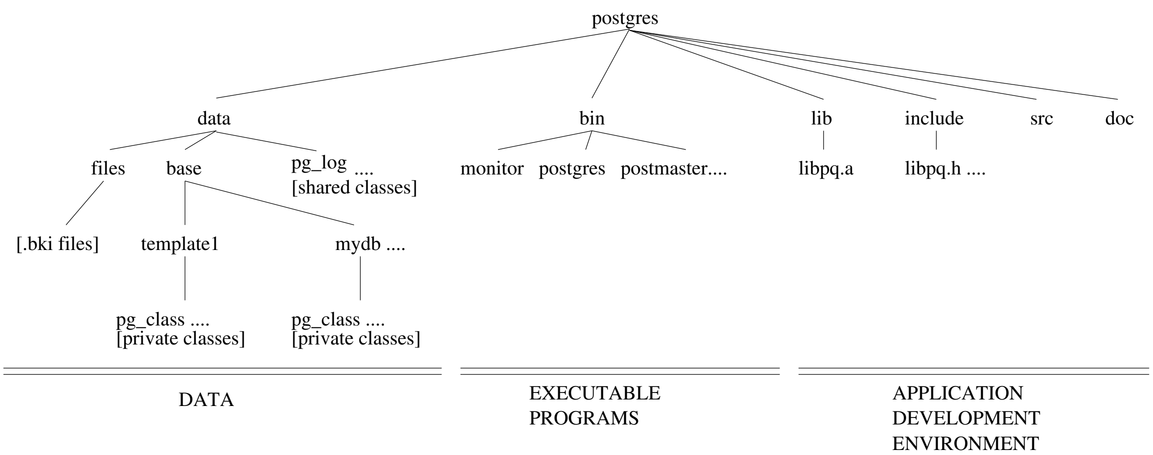

Figure 2. POSTGRES file layout.

Figure 2 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/postgres95. Therefore, wherever you see the directory /usr/local/postgres95 you should substitute the name of the directory where POSTGRES is actually installed.

All POSTGRES commands are installed in the directory /usr/local/postgres95/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 put

```
% set path = ( /usr/local/postgres95/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 put

```
% PATH=/usr/local/postgres95/bin:$PATH
% export PATH
```

in the .profile file in your home directory.

From now on, we will assume that you have put the POSTGRES bin directory in 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 user shell before going any further.

3.2. Admin: Starting the Postmaster

It should be clear from the preceding discussion that 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 to the shell that started it. This is often helpful during troubleshooting. 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.

3.3. Admin: Adding Users

The `createuser` command enables specific users to access POSTGRES.

3.4. User: Starting Applications

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/postgres95/bin` to your shell search path. In most cases, this is all you should have to do in terms of preparation.[1]

If you get the following error message from a POSTGRES command (such as `monitor` or `createdb`):

```
FATAL: StreamOpen: connect() failed: errno=61
FATAL: Failed to connect to postmaster (host=xxx, port=4321)
        Is the postmaster running?
```

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.

3.5. User: Managing a Database

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

3.5.1. Creating a Database

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

```
% createdb foo
```

POSTGRES allows you to create any number of databases at a given site and you automatically become the *database administrator* of the database just created. Database names must have an alphabetic first character and are limited to 16 characters in length.

---

[1] 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.

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.

3.5.2. 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 commands in the SQL.
- 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 the terminal monitor, `monitor`, to try out the examples in this manual. It can be activated for the `foo` database by typing the command:

```
% monitor foo
```

You will be greeted with the following message:

```
Welcome to the POSTGRES95 terminal monitor

Go
*
```

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 `monitor` program responds to escape codes that begin with the backslash character, "\". For example, you print the current contents of the workspace by typing:

```
* \p
```

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

```
* \g
```

This tells the server to `go`. If you make a typing mistake, you can invoke your favorite text editor (eg. `vi`) by typing:

```
* \e
```

The workspace will be passed to the editor, and once you exit it, your edited query will placed in the terminal monitor workspace. You can then submit the contents of the workspace to POSTGRES by using the `\g` command as described above.

To get out of the monitor and return to UNIX, type

```
* \q
```

and `monitor` 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. Comments are denoted by `--`. Everything (except escape codes) after the dashes up to the end of the line is ignored. Unfortunately, the `monitor` implementation does not allow you to comment out an escape code. In other words, this doesn't work as you might expect:

```
-- \g this should be invisible
```

3.5.3. Destroying a Database

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

```
% destroydb foo
```

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.

4. THE QUERY LANGUAGE

The POSTGRES query language is a variant of SQL-3. 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] or [DATE93]. You should also be aware that some features are not part of the ANSI standard.

In the examples that follow, we assume that you have created the `foo` database as described in the previous subsection and have started the terminal monitor.

Examples in this manual can also be found in `/usr/local/postgres95/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/postgres95/src/tutorial
% monitor -s
Welcome to the POSTGRES95 terminal monitor

Go
* \i basics.sql
```

The `i` command read in queries from the specified files. The `-s` option puts you in tutorial mode which pauses at appropriate places. Queries in this section are in `basics.sql`.

4.1. 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.

As previously discussed, classes are grouped into databases, and a collection of databases managed by a single `postmaster` process constitutes an installation or site.

4.2. 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            text,
```

```
        temp_lo         int4,           -- low temperature
        temp_hi         int4,           -- high temperature
        prcp            float8,         -- precipitation
        date            abstime
);
```

For those who have experience with SQL from other vendors, you might not recognize the types. These are some of the base types of POSTGRES SQL (eg. a variable-length array of printable characters (`text`) and a 4-byte signed integer (`int4`)

As we will see later, POSTGRES can be customized with an arbitrary number of user-defined data types.

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, so we use a different word to describe them.

4.3. Populating a Class with Instances

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

```
* INSERT INTO weather
      VALUES ('San Francisco', 46, 50, 0.25, 'Nov 27 1994');
```

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

4.4. Querying a Class

The `weather` class can be queried with normal relational selection and projection queries. A SQL **select** statement is use to do that. 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	"Sun Nov 27 00:00:00 1994 PST"
San Francisco	43	57	0	"Sun Nov 29 00:00:00 1994 PST"
Hayward	37	54		"Sun Nov 29 00:00:00 1994 PST"

You may specify any aribitrary 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	"Sun Nov 27 00:00:00 1994 PST"

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;
```

4.5. 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.[2] 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

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.[3]

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

[3] 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 recom-

4.6. 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 > 'Nov 28 1994';
```

4.7. 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, the **delete** command will simply delete all instances of the given class, leaving it empty. The system **will not request confirmation** before doing this.

---

putes 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. See Appendix ?? under **select** for more details.

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 the introduction of the previous chapter 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      float8,
        altitude        int4            -- (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`. Furthermore, state capitals have an extra attribute, `state`, that shows their state. In POSTGRES, a class can inherit from zero or more other classes,[4] and a query can reference either all instances of a class or all instances of a class plus all of its descendants. For example, the following query finds all the cities that are situated at an attitude of 500 'ft or higher:

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

[4] I.e., the inheritance hierarchy is a directed acyclic graph.

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 500 'ft, 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** command.

5.2. Time Travel

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

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

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 "`[,]`."

5.3. 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.

5.3.1. 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"},{""}}

6. 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

6.1. 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 hard-coded 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.

6.2. 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.

6.3. 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.)

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

The Reference Manual gives a more detailed explanation of these catalogs and their attributes. However, Figure 3 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:

(1) 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.

(2) 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.

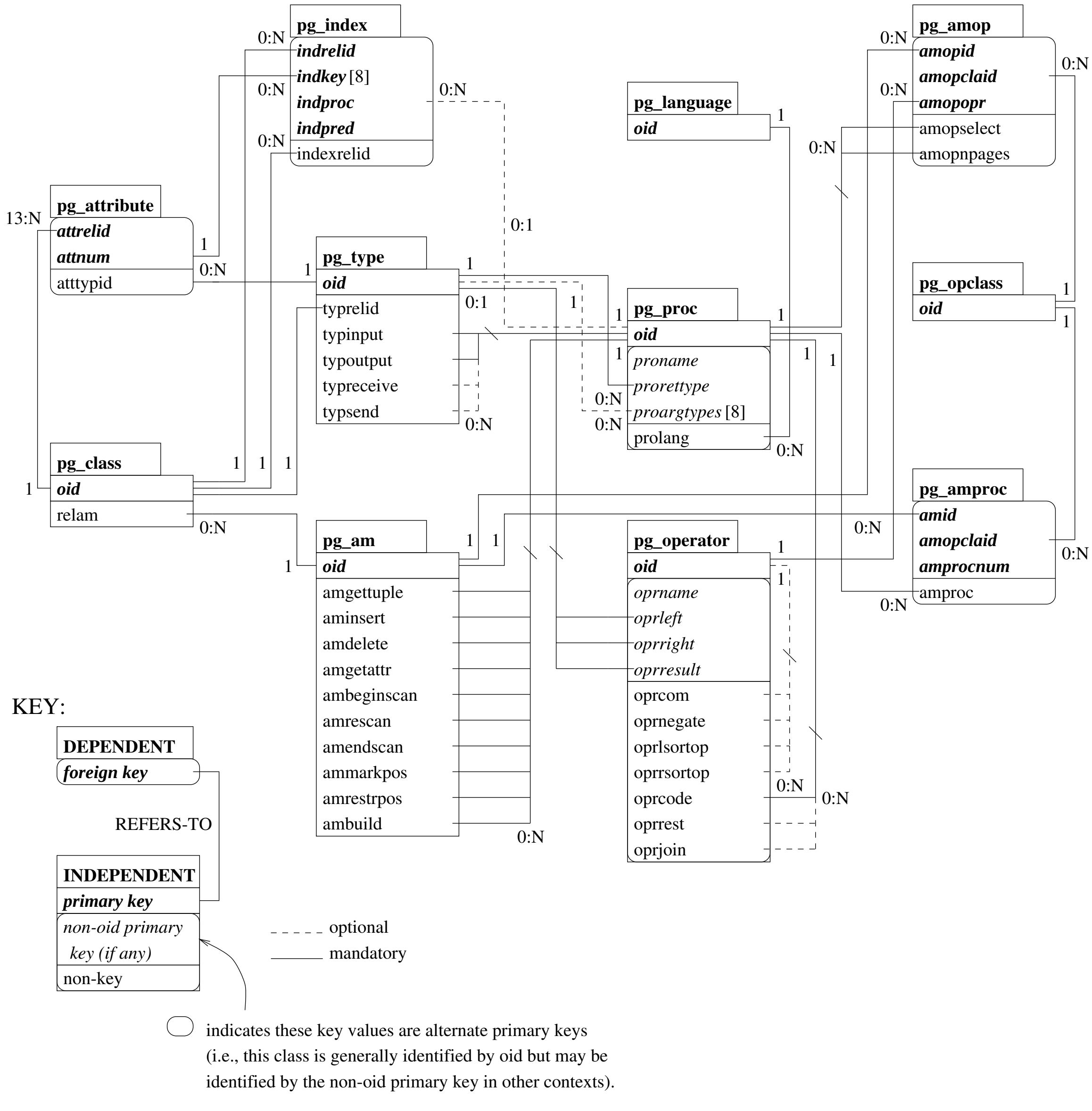

Figure 3. The major POSTGRES system catalogs.

(3) Types and procedures[6] are central to the schema. 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

[6] We use the words *procedure* and *function* more or less interchangably.

unique instances of other catalogs.

(4) 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.

7. 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`.

7.1. Query Language (SQL) Functions

7.1.1. 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

7.1.2. 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 be careful to typecast the expressions (using `::`) very carefully or you will see the following error:

```
WARN:Mar  3 03:06:18: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:Mar  3 03:09:28: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

7.2. Programming Language Functions

7.2.1. 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(sizeof(int4) + 40);
destination->length = sizeof(int4) + 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 '/usr/local/postgres95/tutorial/obj/funcs.so' LANGUAGE 'c';

* CREATE FUNCTION concat16(char16, char16) RETURNS char16
     AS '/usr/local/postgres95/tutorial/obj/funcs.so' LANGUAGE 'c';

* CREATE FUNCTION copytext(text) RETURNS text
     AS '/usr/local/postgres95/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).

7.2.2. 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 '/usr/local/postgres95/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.

7.2.3. Caveats

We now turn to the more difficult task of defining 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:

(1) Most of the header (include) files for POSTGRES should already be installed in /usr/local/postgres95/include (see Figure 2). You should always include

```
-I/usr/local/postgres95/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/usr/local/postgres95/src/backend
-I/usr/local/postgres95/src/backend/include
-I/usr/local/postgres95/src/backend/port/<PORTNAME>
-I/usr/local/postgres95/src/backend/obj
```

(where `<PORTNAME>` is the name of the port, e.g., `alpha` or `sparc).`

(2) 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.

(3) 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.

(4) Most of the internal POSTGRES types are declared in `postgres.h`, so it's usually a good idea to include that file as well.

(5) 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.

8. 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`.

8.1. User-Defined Types

8.1.1. 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.

(1) 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);
}
```

(2) 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 '/usr/local/postgres95/tutorial/obj/complex.so'
   LANGUAGE 'c';

CREATE FUNCTION complex_out(opaque)
   RETURNS opaque
   AS '/usr/local/postgres95/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.

8.1.2. Large Objects

The types discussed to this point are all "small" objects — that is, they are smaller than 8KB[7] in size. 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.

[7] 8 * 1024 == 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.

9. 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)

10. 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 */
                             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

11. INTERFACING EXTENSIONS TO INDICES

The procedures described thus far let you define a new type, new functions and new operators. However, we cannot yet define a secondary index (such as a B-tree, R-tree or hash access method) over a new type or its operators.

Look back at Figure 3. The 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

amname	name of the access method
amowner	object id of the owner's instance in pg_user
amkind	not used at present, but set to 'o' as a place holder
amstrategies	number of strategies for this access method (see below)
amsupport	number of support routines for this access method (see below)
amgettuple aminsert ...	procedure identifiers for interface routines to the access method. For example, regproc ids for opening, closing, and getting instances from the access method appear here.

The object ID of the instance in pg_am is used as a foreign key in lots of other classes. You don't need to add a new instance to this class; all you're interested in is the object ID of the access method instance you want to extend:

```
* SELECT oid FROM pg_am WHERE amname = 'btree'
```

oid
403

The amstrategies attribute exists to standardize comparisons across data types. For example, B-trees impose a strict ordering on keys, lesser to greater. Since POSTGRES allows the user to define operators, POSTGRES cannot look at the **name** of an operator (eg, > or <) and tell what kind of comparison it is. In fact, some access methods don't impose any ordering at all. For example, R-trees express a rectangle-containment relationship, whereas a hashed data structure expresses only bitwise similarity based on the value of a hash function. POSTGRES needs some consistent way of taking a qualification in your query, looking at the operator and then deciding if a usable index exists. This implies that POSTGRES needs to know, for example, that the <= and > operators partition a B-tree. POSTGRES uses strategies to express these relationships between operators and the way they can be used to scan indices.

Defining a new set of strategies is beyond the scope of this discussion, but we'll explain how B-tree strategies work because you'll need to know that to add a new operator class. In the `pg_am` class, the `amstrategies` attribute is the number of strategies defined for this access method. For B-trees, this number is 5. These strategies correspond to

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.[8]

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

[8] Strictly speaking, this routine can return a negative number (< 0), 0, or a non-zero positive number (> 0).

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 B-trees, 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

```
/usr/local/postgres95/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 '/usr/local/postgres95/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:

amopid	the oid of the pg_am instance for B-tree (== 403, see above)
amopclaid	the oid of the pg_opclass instance for int4_abs_ops (== whatever you got instead of 17314, see above)
amopopr	the oids of the operators for the opclass (which we'll get in just a minute)
amopselect, amopnpages	cost functions.

The cost functions are used by the query optimizer to decide whether or not to use a given index in a scan. Fortunately, these already exist. The two functions we'll use are btreesel, which estimates the selectivity of the B-tree, and btreenpage, which estimates the number of pages a search will touch in the tree.

So we need the oids of the operators we just defined. We'll look up the names of all the operators that take two int4s, 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';
```

which returns:

oid	oprname
17321	<
17322	<=
17323	=
17324	>=
17325	>

(Again, some of your oid numbers will almost certainly be different.) The operators we are interested in are those with oids 17321 through 17325. The values you get will probably be different, and you should substitute them for the values below. We can look at the operator names and pick out the ones we just added.

Now we're ready to update pg_amop with our new operator class. The most important thing in this entire discussion is that the operators are ordered, from less equal through greater equal, in pg_amop. We add the instances we need:

```
* INSERT INTO pg_amop (amopid, amopclaid, amopopr, amopstrategy,
                       amopselect, amopnpages)
     SELECT am.oid, opcl.oid, c.opoid, 3,
        'btreesel'::regproc, 'btreenpage'::regproc
```

```
        FROM pg_am am, pg_opclass opcl, complex_ops_tmp c
        WHERE amname = 'btree' and opcname = 'complex_abs_ops'
           and c.oprname = '=';
```

Note the order: "less than" is 1, "less than or equal" is 2, "equal" is 3, "greater than or equal" is 4, and "greater than" is 5.

The last step (finally!) is registration of the "support routine" previously described in our discussion of `pg_am`. The `oid` of this support routine is stored in the `pg_amproc` class, keyed by the access method `oid` and the operator class `oid`. First, we need to register the function in POSTGRES (recall that we put the C code that implements this routine in the bottom of the file in which we implemented the operator routines):

```
* CREATE FUNCTION int4_abs_cmp(int4, int4)
       RETURNS int4
       AS '/usr/local/postgres95/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);
```

12. 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.

Originally, POSTGRES 4.2 supports 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 POSTGRES95. Even though is is slower to access, it provides stricter data integrity and time travel. For historical reasons, they are called Inversion large objects. (We will use Inversion and large objects interchangerably to mean the same thing in this section.)

12.1. 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.

12.2. 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, POSTGRES95 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 .

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.

12.2.1. Creating a Large Object

The routine

```
Oid
lo_creat(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

```
/usr/local/postgres95/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. At Berkeley, storage manager zero is magnetic disk, storage manager one is a Sony optical disk jukebox, and storage manager two is main memory.

The commands below create an (Inversion) large object located on magnetic disk:

```
inv_oid = lo_creat(INV_MD|INV_ARCHIVE);
```

12.2.2. Importing a Large Object

To import a UNIX file as a large object, call

```
Oid
lo_import(text *filename)
```

The *filename* argument specifies the UNIX pathname of the file to be imported as a large object.

12.2.3. Opening an Existing Large Object

To open an existing large object, call

```
int
lo_open(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`.

12.2.4. Writing Data to a Large Object

The routine

```
int
lo_write(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.

12.2.5. Seeking on a Large Object

To change the current read or write location on a large object, call

```
int
lo_lseek(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*. For this release of POSTGRES, only L_SET is a legal value for *whence*.

12.2.6. Closing a Large Object Descriptor

A large object may be closed by calling

```
int
lo_close(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.

12.3. Example

Let's look at an example of a schema involving large objects.

```
CREATE TABLE image (
        name            text,
        raster          oid
);
\g

INSERT INTO image (name, raster)
   VALUES ("beautiful image", lo_import("/etc/motd"));
\g
```

The following shows how to use the other functions:

```
/*
 * exportFile -
 *    export large object "lobjOid" to file "out_filename"
 *
 */
void exportFile(Oid lobjId, char *filename)
{
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
```

```
        int fd;

        /*
         * create an inversion "object"
         */
        lobj_fd = lo_open(lobjId, INV_READ);
        if (lobj_fd < 0) {
          elog(WARN, "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 */
          elog(WARN, "can't open unix file
               filename);
        }

        /*
         * read in from the Unix file and write to the inversion file
         */
        while ((nbytes = lo_read(lobj_fd, buf, BUFSIZE)) > 0) {
          tmp = write(fd, buf, nbytes);
            if (tmp < nbytes) {
              elog(WARN, "error while writing
                 filename);
          }
        }

        (void) lo_close(lobj_fd);
        (void) close(fd);

        return;
}
```

12.4. Comments

There is no external large object. Use text to store the path name!

13. 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 [STON90b] 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 [ONG90] as well as [STON90b].

14. ADMINISTERING POSTGRES

In this section, we will discuss aspects of POSTGRES that are of interest to those who make extensive use of POSTGRES, or who are the site administrator for a group of POSTGRES users.

14.1. Frequent Tasks

Here we will briefly discuss some procedures that you should be familiar with in managing any POSTGRES installation.

14.1.1. Starting the Postmaster

If you did not install POSTGRES exactly as described in the installation instructions, you may have to perform some additional steps before starting the `postmaster` process.

- Even if you were not the person who installed POSTGRES, you should understand the installation instructions. The installation instructions explain some important issues with respect to where POSTGRES places some important files, proper settings for environment variables, etc. that may vary from one version of POSTGRES to another.
- You *must* start the `postmaster` process with the user-id that owns the installed database files. In most cases, if you have followed the installation instructions, this will be the user "postgres". If you do not start the `postmaster` with the right user-id, the backend servers that are started by the `postmaster` will not be able to read the data.
- Make sure that `/usr/local/postgres95/bin` is in your shell command path, because the `postmaster` will use your `PATH` to locate POSTGRES commands.
- Remember to set the environment variable `PGDATA` to the directory where the POSTGRES databases are installed. (This variable is more fully explained in the POSTGRES installation instructions and the Reference Manual.)
- If you do start the `postmaster` using non-standard options, such as a different TCP port number, remember to tell all users so that they can set their `PGPORT` environment variable correctly.

14.1.2. Shutting Down the Postmaster

If you need to halt the `postmaster` process, you can use the UNIX `kill(1)` command. Some people habitually use the `-9` or `-KILL` option; this should never be necessary and we do not recommend that you do this, as the `postmaster` will be unable to free its various shared resources, its child processes will be unable to exit gracefully, etc.

14.1.3. Adding and Removing Users

The `createuser` and `destroyuser` commands enable and disable access to POSTGRES by specific users on the host system. Please read the descriptions of these commands in the Reference Manual for specific instructions on their use.

14.1.4. Periodic Upkeep

The `vacuum` command should be run on each database periodically. This command processes deleted instances[9] and, more importantly, updates the system *statistics* concerning the size of each class. If these statistics are permitted to become out-of-date and inaccurate, the POSTGRES query optimizer may make extremely poor decisions with respect to query evaluation strategies. Therefore, we recommend running `vacuum` every night or so (perhaps in a script that is executed by the UNIX `cron`(1) or `at`(1) commands).

Do frequent backups. That is, you should either back up your database directories using the POSTGRES **copy** command and/or the UNIX `dump`(1) or `tar`(1) commands. You may think, “Why am I backing up my database? What about crash recovery?” One side effect of the POSTGRES “no overwrite” storage manager is that it is also a “no log” storage manager. That is, the database log stores only abort/commit data, and this is not enough information to recover the database if the storage medium (disk) or the database files are corrupted! In other words, if a disk block goes bad or POSTGRES happens to corrupt a database file, **you cannot recover that file**. This can be disastrous if the file is one of the shared catalogs, such as `pg_database`.

14.1.5. Tuning

Once your users start to load a significant amount of data, you will typically run into performance problems. POSTGRES is not the fastest DBMS in the world, but many of the worst problems encountered by users are due to their lack of experience with any DBMS. Some general tips include:

(1) Define indices over attributes that are commonly used for qualifications. For example, if you often execute queries of the form

```
SELECT * from EMP where salary < 5000
```

then a B-tree index on the `salary` attribute will probably be useful. If scans involving equality are more common, as in

```
SELECT * from EMP where salary = 5000
```

then you should consider defining a hash index on `salary`. You can define both, though it will use more disk space and may slow down updates a bit. Scans using indices are **much** faster than sequential scans of the entire class.

---

[9] This may mean different things depending on the *archive mode* with which each class has been created. See the Reference Manual under the heading **create** for more details. However, the current implementation of the `vacuum` command does *not* perform any compaction or clustering of data. Therefore, the UNIX files which store each POSTGRES class never shrink and the space “reclaimed” by `vacuum` is never actually reused.

(2) Run the `vacuum` command a lot. This command updates the statistics that the query optimizer uses to make intelligent decisions; if the statistics are inaccurate, the system will make inordinately stupid decisions with respect to the way it joins and scans classes.

(3) When specifying query qualfications (i.e., the `where` part of the query), try to ensure that a clause involving a constant can be turned into one of the form *range_variable operator constant*, e.g.,

```
EMP.salary = 5000
```

The POSTGRES query optimizer will only use an index with a constant qualification of this form. It doesn't hurt to write the clause as

```
5000 = EMP.salary
```

if the operator (in this case, =) has a *commutator* operator defined so that POSTGRES can rewrite the query into the desired form. However, if such an operator does not exist, POSTGRES will never consider the use of an index.

(4) When joining several classes together in one query, try to write the join clauses in a "chained" form, e.g.,

```
where A.a = B.b and B.b = C.c and ...
```

Notice that relatively few clauses refer to a given class and attribute; the clauses form a linear sequence connecting the attributes, like links in a chain. This is preferable to a query written in a "star" form, such as

```
where A.a = B.b and A.a = C.c and ...
```

Here, many clauses refer to the same class and attribute (in this case, `A.a`). When presented with a query of this form, the POSTGRES query optimizer will tend to consider far more choices than it should and may run out of memory.

(5) If you are really desperate to see what query plans look like, you can run the `postmaster` with the `-d` option and then run `monitor` with the `-t` option. The format in which query plans will be printed is hard to read but you should be able to tell whether any index scans are being performed. See the Reference Manual under **postgres** and **postmaster**.

14.2. Infrequent Tasks

At some time or another, every POSTGRES site administrator has to perform all of the following actions.

14.2.1. Cleaning Up After Crashes

The `postgres` server and the `postmaster` run as two different processes. They may crash separately or together. The housekeeping procedures required to fix one kind of crash are different from those required to fix the other.

The message you will usually see when the backend server crashes is:

```
FATAL: no response from backend: detected in ...
```

This generally means one of two things: there is a bug in the POSTGRES server, or there is a bug in some user code that has been dynamically loaded into POSTGRES. You should be able to restart your application and resume processing, but there are some considerations:

(1) POSTGRES usually dumps a core file (a snapshot of process memory used for debugging) in the database directory

```
/usr/local/postgres95/data/base/<database>/core
```

on the server machine. If you don't want to try to debug the problem or produce a stack trace to report the bug to someone else, you can delete this file (which is probably around 10MB).

(2) When one backend crashes in an uncontrolled way (i.e., without calling its built-in cleanup routines), the `postmaster` will detect this situation, kill all running servers and reinitialize the state shared among all backends (e.g., the shared buffer pool and locks). If your server crashed, you will get the "no response" message shown above. If your server was killed because someone else's server crashed, you will see the following message:

```
I have been signalled by the postmaster.
Some backend process has died unexpectedly and possibly
corrupted shared memory.  The current transaction was
aborted, and I am going to exit.  Please resend the
last query. -- The postgres backend
```

(3) Sometimes shared state is not completely cleaned up. Frontend applications may see errors of the form:

```
WARN:Mar 11 14:41:29: cannot write block 34 of myclass [mydb] blind
```

In this case, you should kill the `postmaster` and restart it.

(4) When the system crashes while updating the system catalogs (e.g., when you are creating a class, defining an index, retrieving into a class, etc.) the B-tree indices defined on the catalogs are sometimes corrupted. The general (and non-unique) symptom is that **all** queries stop working. If you have tried all of the above steps and nothing else seems to work, try using the `reindexdb` command. If `reindexdb` succeeds but things still don't work, you have another problem; if it fails, the system catalogs themselves were almost certainly corrupted and you will have to go back to your backups.

The `postmaster` does not usually crash (it doesn't do very much except start servers) but it does happen on occasion. In addition, there are a few cases where it encounters problems during the reinitialization of shared resources. Specifically, there are race conditions where the operating system lets the `postmaster` free shared resources but then will not permit it to reallocate the same amount of shared resources (even when there is no contention).

You will typically have to run the `ipcclean` command if system errors cause the `postmaster` to crash. If this happens, you may find (using the UNIX `ipcs`(1) command) that the "postgres" user has shared memory and/or semaphores allocated even though no `postmaster` process is running. In this case, you should run `ipcclean` as the "postgres" user in order to deallocate these resources. Be warned that *all* such resources owned by the "postgres" user will be deallocated. If you have multiple `postmaster` processes running on the same machine, you should kill all of them before running `ipcclean` (otherwise, they will crash on their own when their shared resources are suddenly deallocated).

14.2.2. Moving Database Directories

By default, all POSTGRES databases are stored in separate subdirectories under `/usr/local/postgres95/data/base`.[10] At some point, you may find that you wish to move one or more databases to another location (e.g., to a filesystem with more free space).

If you wish to move *all* of your databases to the new location, you can simply:

- Kill the `postmaster`.
- Copy the entire `data` directory to the new location (making sure that the new files are owned by user "postgres").

```
% cp -rp /usr/local/postgres95/data /new/place/data
```

- Reset your `PGDATA` environment variable (as described earlier in this manual and in the installation instructions).

```
# using csh or tcsh...
% setenv PGDATA /new/place/data

# using sh, ksh or bash...
% PGDATA=/new/place/data; export PGDATA
```

- Restart the `postmaster`.

```
% postmaster &
```

- After you run some queries and are sure that the newly-moved database works, you can remove the old `data` directory.

```
% rm -rf /usr/local/postgres95/data
```

---

[10] Data for certain classes may stored elsewhere if a non-standard storage manager was specified when they were created. Use of non-standard storage managers is an experimental feature that is not supported outside of Berkeley.

To install a *single* database in an alternate directory while leaving all other databases in place, do the following:

- Create the database (if it doesn't already exist) using the **createdb** command. In the following steps we will assume the database is named `foo`.
- Kill the `postmaster`.
- Copy the directory `/usr/local/postgres95/data/base/foo` and its contents to its ultimate destination. It should still be owned by the "postgres" user.

```
% cp -rp /usr/local/postgres95/data/base/foo /new/place/foo
```

- Remove the directory `/usr/local/postgres95/data/base/foo`:

```
% rm -rf /usr/local/postgres95/data/base/foo
```

- Make a symbolic link from `/usr/local/postgres95/data/base` to the new directory:

```
% ln -s /new/place/foo /usr/local/postgres95/data/base/foo
```

- Restart the `postmaster`.

14.2.3. Updating Databases

POSTGRES is a research system. In general, POSTGRES may not retain the same binary format for the storage of databases from release to release. Therefore, when you update your POSTGRES software, you will probably have to modify your databases as well. This is a common occurrence with commercial database systems as well; unfortunately, unlike commercial systems, POSTGRES does not come with user-friendly utilities to make your life easier when these updates occur.

In general, you must do the following to update your databases to a new software release:

- *Extensions* (such as user-defined types, functions, aggregates, etc.) must be reloaded by re-executing the SQL **CREATE** commands. See Appendix A for more details.
- *Data* must be dumped from the old classes into ASCII files (using the **COPY** command), the new classes created in the new database (using the **CREATE**TABLE command), and the data reloaded from the ASCII files.
- *Rules* and *views* must also be reloaded by re-executing the various **CREATE** commands.

You should give any new release a "trial period"; in particular, do not delete the old database until you are satisfied that there are no compatibility problems with the new software. For example, you do not want to discover that a bug in a type's "input" (conversion from ASCII) and "output" (conversion to ASCII) routines prevents you from reloading your data after you have destroyed your old databases! (This should be standard procedure when updating any software package, but some people try to economize on disk space without applying enough foresight.)

14.3. Database Security

Most sites that use POSTGRES are educational or research institutions and do not pay much attention to security in their POSTGRES installations. If desired, one can install POSTGRES with additional security features. Naturally, such features come with additional administrative overhead that must be dealt with.

14.3.1. Kerberos

POSTGRES can be configured to use the MIT Kerberos network authentication system. This prevents outside users from connecting to your databases over the network without the correct authentication information. For more information on Kerberos, see the file `src/doc/kerberos.faq` and the **UNIX** section of the Reference Manual.

14.3.2. Access Control

Access control lists (ACLs) can be defined on a per-class basis. These work rather like a more flexible version of the UNIX `chmod`(1) command. See the Reference Manual under the heading **change acl** for more details.

14.4. Querying the System Catalogs

As an administrator (or sometimes as a plain user), you want to find out what extensions have been added to a given database. The queries listed below are "canned" queries that you can run on any database to get simple answers. Before executing any of the queries below, be sure to execute the POSTGRES `vacuum` command. (The queries will run much more quickly that way.) Also, note that these queries are also listed in

```
/usr/local/postgres95/tutorial/syscat.sql
```

so use cut-and-paste (or the `\i` command) instead of doing a lot of typing.

This query prints the names of all database adminstrators and the name of their database(s).

```
* SELECT usename, datname
    FROM pg_user, pg_database
    WHERE usesysid = int2in(int4out(datdba))
    ORDER BY usename, datname;
```

This query lists all user-defined classes in the database.

```
* SELECT relname
    FROM pg_class
    WHERE relkind = 'r'           -- not indices
      and relname !~ '^pg_'       -- not catalogs
      and relname !~ '^Inv'       -- not large objects
    ORDER BY relname;
```

This query lists all simple indices (i.e., those that are not defined over a function of several attributes).

```
* SELECT bc.relname AS class_name,
         ic.relname AS index_name,
         a.attname
    FROM pg_class bc,             -- base class
         pg_class ic,             -- index class
         pg_index i,
         pg_attribute a           -- att in base
    WHERE i.indrelid = bc.oid
       and i.indexrelid = ic.oid
       and i.indkey[0] = a.attnum
       and a.attrelid = bc.oid
       and i.indproc = '0'::oid   -- no functional indices
    ORDER BY class_name, index_name, attname;
```

This query prints a report of the user-defined attributes and their types for all user-defined classes in the database.

```
* SELECT c.relname, a.attname, t.typname
    FROM pg_class c, pg_attribute a, pg_type t
    WHERE c.relkind = 'r'     -- no indices
      and c.relname !~ '^pg_' -- no catalogs
      and c.relname !~ '^Inv' -- no large objects
      and a.attnum > 0        -- no system att's
      and a.attrelid = c.oid
      and a.atttypid = t.oid
    ORDER BY relname, attname;
```

This query lists all user-defined base types (not including array types).

```
* SELECT u.usename, t.typname
    FROM pg_type t, pg_user u
    WHERE u.usesysid = int2in(int4out(t.typowner))
      and t.typrelid = '0'::oid   -- no complex types
      and t.typelem = '0'::oid    -- no arrays
      and u.usename <> 'postgres'
    ORDER BY usename, typname;
```

This query lists all left-unary (post-fix) operators.

```
* SELECT o.oprname AS left_unary,
         right.typname AS operand,
         result.typname AS return_type
    FROM pg_operator o, pg_type right, pg_type result
    WHERE o.oprkind = 'l'           -- left unary
      and o.oprright = right.oid
      and o.oprresult = result.oid
    ORDER BY operand;
```

This query lists all right-unary (pre-fix) operators.

```
* SELECT o.oprname AS right_unary,
```

```
          left.typname AS operand,
          result.typname AS return_type
     FROM pg_operator o, pg_type left, pg_type result
     WHERE o.oprkind = 'r'          -- right unary
       and o.oprleft = left.oid
       and o.oprresult = result.oid
     ORDER BY operand;
```

This query lists all binary operators.

```
* SELECT o.oprname AS binary_op,
          left.typname AS left_opr,
          right.typname AS right_opr,
          result.typname AS return_type
     FROM pg_operator o, pg_type left, pg_type right, pg_type result
     WHERE o.oprkind = 'b'         -- binary
       and o.oprleft = left.oid
       and o.oprright = right.oid
       and o.oprresult = result.oid
     ORDER BY left_opr, right_opr;
```

This query returns the name, number of arguments (parameters) and return type of all user-defined C functions. The same query can be used to find all built-in C functions if you change the "C" to "internal", or all SQL functions if you change the "C" to "postquel".

```
* SELECT p.proname, p.pronargs, t.typname
     FROM pg_proc p, pg_language l, pg_type t
     WHERE p.prolang = l.oid
       and p.prorettype = t.oid
       and l.lanname = 'c'
     ORDER BY proname;
```

This query lists all of the aggregate functions that have been installed and the types to which they can be applied. `count` is not included because it can take any type as its argument.

```
* SELECT a.aggname, t.typname
     FROM pg_aggregate a, pg_type t
     WHERE a.aggbasetype = t.oid
     ORDER BY aggname, typname;
```

This query lists all of the operator classes that can be used with each access method as well as the operators that can be used with the respective operator classes.

```
* SELECT am.amname, opc.opcname, opr.oprname
     FROM pg_am am, pg_amop amop, pg_opclass opc, pg_operator opr
     WHERE amop.amopid = am.oid
       and amop.amopclaid = opc.oid
       and amop.amopopr = opr.oid
     ORDER BY amname, opcname, oprname;
```

15. REFERENCES

[DATE93] Date, C. J. and Darwen, Hugh, A Guide to The SQL Standard, 3rd Edition, Reading, MA, June 1993.

[MELT93] Melton, J. Understanding the New SQL, 1994.

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

[ROWE87] Rowe, L. and Stonebraker, M., "The POSTGRES Data Model," Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[STON86] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, DC, May 1986.

[STON87a] Stonebraker, M., Hanson, E. and Hong, C.-H., "The Design of the POSTGRES Rules System," Proc. 1987 IEEE Conference on Data Engineering, Los Angeles, CA, Feb. 1987.

[STON87b] Stonebraker, M., "The POSTGRES Storage System," Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

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

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

[STON90b] Stonebraker, M. et al., "On Rules, Procedures, Caching and Views in Database Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., June 1990.

Appendix A: 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 run-time, as required. In order to allow your code to be dynamically loaded, you may have to compile and link-edit it in a special way. This section briefly describes how to perform the compilation and link-editing required before you can load your user-defined functions into a running POSTGRES server. Note that *this process has changed as of Version 4.2.*[11] 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 `/usr/local/postgres95/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.[12]
- 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.

[11] 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 post-processed 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.

[12] 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!

(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.

ULTRIX

It is very easy to build dynamically-loaded object files under ULTRIX. ULTRIX does not have any shared-library mechanism and hence does not place any restrictions on the dynamic loader interface. On the other hand, we had to (re)write a non-portable dynamic loader ourselves and could not use true shared libraries.

Under ULTRIX, the only restriction is that you must produce each object file with the option `-G 0`. (Notice that that's the numeral "`0`" and not the letter "`O`"). For example,

```
# simple ULTRIX example
% cc -G 0 -c foo.c
```

produces an object file called `foo.o` that can then be dynamically loaded into POSTGRES. No additional loading or link-editing must be performed.

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.[13] If the file you specify is not a shared object, the backend will hang!

SunOS 4.x, Solaris 2.x and HP-UX

Under both 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.

---

[13] 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`!

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.