The POSTGRES User Manual

Edited by the POSTGRES Group
Computer Science Div., Dept. of EECS
University of California at Berkeley

1. INTRODUCTION

This document is the user manual for the POSTGRES database management system developed at the University of California at Berkeley. This project, led by Professor Michael Stonebraker, has been sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

The first part of this manual 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.

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, at least two companies (Multimedia Information Systems and Montage Software) have 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. About This Release

Version 4.2, the current version of POSTGRES, is about 200,000 lines of code in the C programming language. POSTGRES is available free of charge, and (as of this writing) has been installed by approximately 600 sites around the world.

This manual describes Version 4.2 of POSTGRES. The POSTGRES group has compiled and tested Version 4.2 on the following platforms:

architecture	processor	operating system
DECstation 3000	Alpha AXP	OSF/1 1.3, 2.0
DECstation 3100 and 5000	MIPS	ULTRIX 4.2A, 4.3A
Sun4	SPARC	SunOS 4.1.3, 4.1.3_U1; Solaris 2.3
H-P 9000/700 and 800	PA-RISC	HP-UX 9.00, 9.01, 9.03
IBM RS/6000	POWER	AIX 3.2.5

Previous versions of POSTGRES ran on Sun Microsystems Sun3 and Sequent Symmetry machines. POSTGRES no longer runs on these systems. Outside users have ported previous releases of POSTGRES to many platforms, including NeXTSTEP, IRIX 5.2, Intel System V Release 4, Linux, FreeBSD and NetBSD.

Version 4.2 has been tuned modestly. On the Wisconsin benchmark, one should expect performance about twice that of the public domain, University of California version of INGRES, a relational prototype from the late 1970s.

2. WHAT YOU SHOULD READ

This manual is primarily intended to provide a broad overview of the system, as well as to illustrate how C programmers can tie their own code into the POSTGRES database server (commonly referred to as the *backend server*, or simply ‘‘backend’’).

In addition to this manual, there is another document, the POSTGRES Reference Manual. The Reference Manual gives full descriptions of the syntax and options for each command in a format not unlike UNIX ‘‘man pages.’’ (In fact, the contents of the Reference Manual should be available on-line as actual man pages.) However, the Reference Manual is designed as a complete reference for the experienced POSTGRES user and contains few tutorial examples. This User Manual does not attempt to provide all of the information that the Reference Manual provides. Instead, it describes the major *concepts* of the system, gives *examples* of the use of the major constructs, and then provides *pointers* to the appropriate place in the Reference Manual in which you can find more information if you so desire.

If you are new to POSTGRES, you should probably read this manual first, followed by the parts of the POSTGRES Reference Manual necessary to build your application. In particular, you should read the Reference Manual section on LIBPQ if you intend to build a client application around POSTGRES, as that library is not discussed in this manual.

If you are not already familiar with relational databases, you should probably find a good introductory text on the subject. This manual assumes that you already have some knowledge of the relational model, and it doesn't hurt to know a query language such as QUEL or SQL.

3. 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

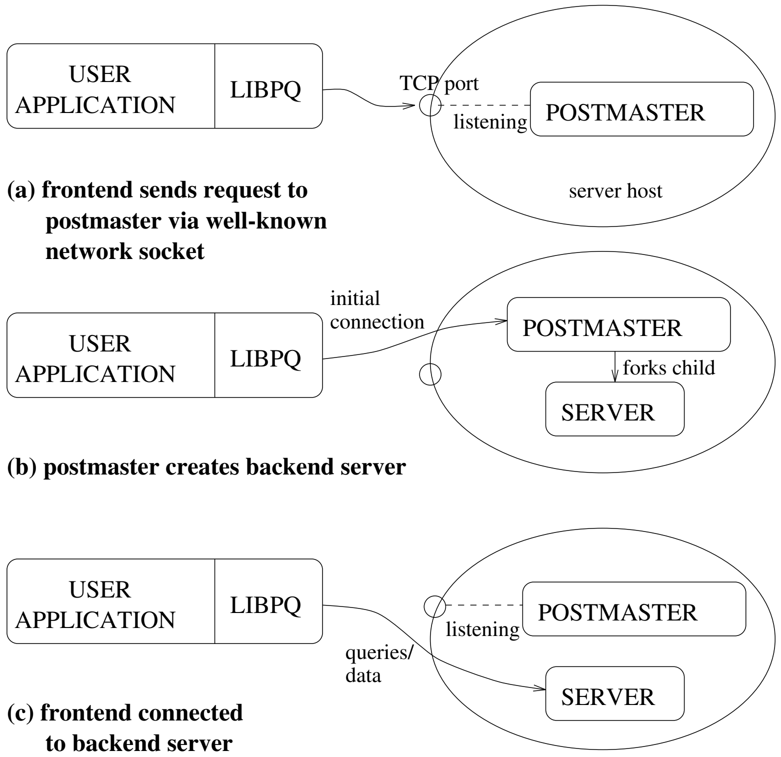

Figure 1. How a connection is established.

- 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.

4. GETTING STARTED WITH POSTGRES

Before you can start learning the POSTQUEL query language, you need to have a working POSTGRES system. This section discusses how to start POSTGRES and set up your own environment so that you can use frontend applications.

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, POSTQUEL.

4.1. Admin: Installing POSTGRES

Detailed installation instructions can be found in the POSTGRES source code distribution. The `troff` source is located in the file `src/doc/postgres-setup.me` and a formatted version is located at the top of the distribution directory tree. Those instructions vary from release to release and will not be duplicated here. However, if you are installing POSTGRES now, you must read these instructions and carry them out before going any further.

A reminder: don't run the regression tests as the "postgres" user. Part of the test is a check of the POSTGRES security mechanisms that turns off superuser permissions. If you run the test as "postgres," you may not be able to add users later.

4.2. Admin/User: Setting Up Your Environment

Figure 2 shows how the POSTGRES distribution is laid out when installed in the default way. The system can be installed such that the various top-level directories can be scattered around your disks, but for the sake of simplifying this manual we will assume that this is not the case. In the examples that follow, we will assume that POSTGRES has been installed in the directory `/usr/local/postgres`. Therefore, wherever you see the directory `/usr/local/postgres` you should substitute the name of the directory where POSTGRES is actually installed.

All POSTGRES commands are installed in the directory `/usr/local/postgres/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

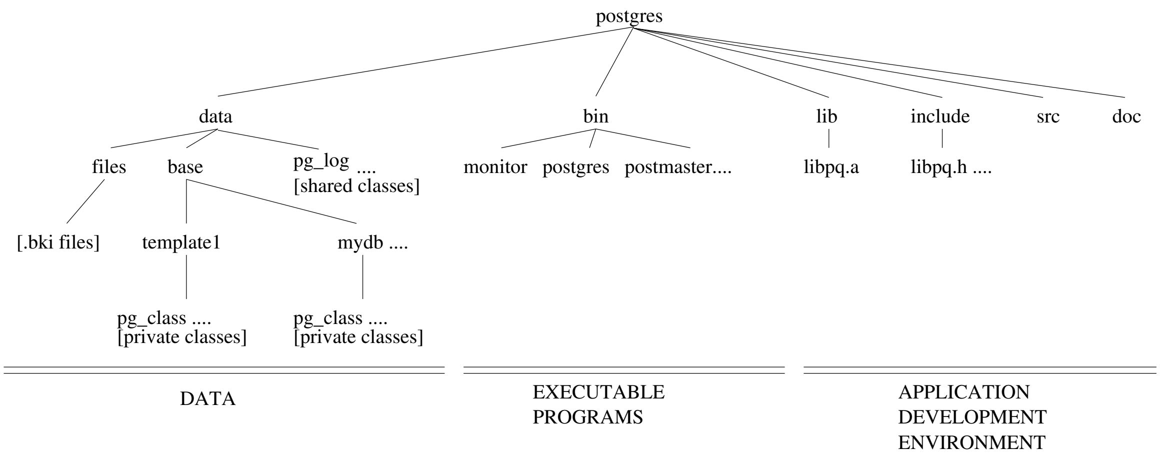

Figure 2. POSTGRES file layout.

```
% set path = ( /usr/local/postgres/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/postgres/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.

4.3. 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 &
```

If the `postmaster` does not start, but instead prints a series of cryptic error messages, you should consult the Reference Manual under the heading **postmaster**. This manual page contains troubleshooting tips.

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.

4.4. Admin: Adding Users

The `createuser` command enables specific users to access POSTGRES. Please read the descriptions of these commands in the Reference Manual for specific instructions on their use.

4.5. 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/postgres/bin` to your shell search path. In most cases, this is all you should have to do in terms of preparation.. rr *

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.

---

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.

4.6. 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.

4.6.1. 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.

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.

4.6.2. Accessing a Database

Once you have constructed a database, there are three ways to access it:

- You can run the POSTGRES terminal monitor (the `monitor` program) which allows you to interactively enter, edit, and execute commands in the POSTQUEL query language.
- You can interact with POSTGRES from a C program by using the LIBPQ subroutine library. This allows you to submit POSTQUEL commands from C and get answers and status messages back to your program. This interface is discussed further in the LIBPQ section of the Reference Manual.
- You can use the *fast path* facility, which allows you to execute functions within the server program itself. This facility is (minimally) described in the Reference Manual under "Fast Path."

This manual will only discuss access through the terminal monitor.

The terminal monitor can be activated for the `foo` database by typing the command:

```
% monitor foo
```

You will be greeted with the following message:

```
Welcome to the POSTGRES terminal monitor

Go
*
```

This prompt indicates that the terminal monitor is listening to you and that you can type POSTQUEL 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 the `vi` text editor by typing:

```
* \e
```

The workspace will be passed to the editor, and once you exit `vi`, 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.

There are two other things that `monitor` understands that make it easier to write nice-looking scripts. First, white space (i.e., spaces, tabs and newlines) may be used freely in POSTQUEL queries. Second, comments that look like those used in the C programming language, e.g.,

```
/* This is a comment. */
```

may also be used in your queries. Beware: you cannot comment out an escape code. In other words, this doesn't work as you might expect:

```
/* I don't want to send this!\g  */
retrieve (message = "but I want to send this!") \g
```

For a complete description of the `monitor` commands and its options, see the Reference Manual under the heading **monitor**.

4.6.3. 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.

5. THE POSTQUEL QUERY LANGUAGE

POSTQUEL is the POSTGRES query language. POSTQUEL was derived from the QUEL language developed by the University of California INGRES project, but the two languages are different in many ways. This section provides an overview of how to use the more QUEL-like features of POSTQUEL to perform simple operations.

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.

Before you start reading, take a look at the directory `/usr/local/postgres/src/examples`. This directory contains all of the POSTQUEL queries listed in this manual (the ones that aren't examples of things that don't work, that is) broken down by chapter. Instead of typing the queries below into the `monitor` program, you can just cut and paste out of the appropriate file or use the `\i` command at the terminal monitor.

5.1. 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.

5.2. Creating a New Class

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

```
* create EMP (name = text, salary = int4,
              age = int4, dept = char16) \g

* create DEPT (dname = char16, floor = int4,
               manager = text) \g
```

The POSTQUEL base types used above are a variable-length array of printable characters (`text`), a 4-byte signed integer (`int4`), and a fixed-length array of 16 characters (`char16`.)

So far, the POSTGRES **create** command looks exactly like the command used to create a table in a traditional relational system. This exact syntax was used in QUEL, the original INGRES query language. 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.

5.3. Populating a Class with Instances

The **append** command is used to populate a class with instances:

```
* append EMP (name = "Claire", salary = 2000,
              age = 40, dept = "shoe") \g

* append EMP (name = "Joe", salary = 1400,
              age = 40, dept = "shoe") \g

* append EMP (name = "Sam", salary = 1200,
              age = 29, dept = "toy") \g

* append EMP (name = "Bill", salary = 1600,
              age = 36, dept = "candy") \g

* append DEPT (dname = "shoe", floor = 5,
               manager = "Claire") \g

* append DEPT (dname = "toy", floor = 3,
               manager = "Sam") \g

* append DEPT (dname = "candy", floor = 4,
               manager = "(None)") \g
```

This adds four instances to the `EMP` class, one for each **append** command.

You can also use the **copy** command to perform load large amounts of data from flat (ASCII) files. See the Reference Manual under **copy** for details.

5.4. Querying a Class

The `EMP` class can be queried with normal relational selection and projection queries. The POSTQUEL equivalent of the SQL **select** statement is **retrieve**. As in SQL, 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 find the employees under 35 years of age, type:

```
* retrieve (EMP.name) where EMP.age < 35 \g
```

and the output should be:

name
Sam

Note that, unlike SQL, parentheses are required around the target list, `EMP.name`.

POSTQUEL allows you to return arbitrary computations in the target list as long as they are given some kind of name:

```
* retrieve (result = EMP.salary / EMP.age)
      where EMP.name = "Bill" \g
```

result
44

In this case, we divided Bill's salary by his age and called the result `result`. (Of course, the answer is really 44 $\frac{4}{9}$, but division of two integers produces another integer so the fraction is lost.).

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

```
* retrieve (EMP.all)
      where EMP.age < 30
         or not EMP.name = "Joe" \g
```

name	salary	age	dept
Claire	2000	36	shoe
Sam	1200	29	toy
Bill	1600	36	candy

As a final note, you can specify that the results of a **retrieve** can be returned in a sorted order or with duplicate instances removed. See the Reference Manual under **retrieve** for more information.

5.5. Redirecting Retrieve Queries

Any **retrieve** query can be redirected to a new class in the database:

```
* retrieve into temp (EMP.name)
      where EMP.age < 35 and EMP.salary > 1000 \g
```

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

```
* retrieve (temp.all) \g
```

name
Sam

5.6. Joins Between Classes

Thus far, our queries have only accessed one class at a time. Queries can access multiple classes at once, or access the same class in such a way that multiple instances of the class are being processed at the same time. A query that accesses multiple instances of the same or different classes at one time is called a *join query*.

As an example, say we wish to find the names of employees which are the same age. In effect, we need to compare the age attribute of each EMP instance to the age attribute of all other EMP instances.. rr * We can do this with the following query:

```
* retrieve (E1.name, E2.name)
      from E1 in EMP, E2 in EMP
      where E1.age = E2.age and E1.name != E2.name \g
```

name	name
Bill	Claire
Claire	Bill

In this case, both E1 and E2 are *surrogates* for an instance of the class EMP, and both range over all instances of the class. (In the terminology of most database systems, E1 and E2 are known as "range variables.") A POSTQUEL query can contain an arbitrary number of class names and surrogates.. rr *

5.7. Updates

You can update existing instances using the **replace** command:

```
* replace EMP (salary = E.salary)
      from E in EMP
      where EMP.name = "Joe" and E.name = "Sam" \g
```

This command replaces the salary of Joe by that of Sam.

Notice that this example is actually another join query. Here, we are using the actual class name ("EMP") as one range variable and a surrogate name for EMP ("E") as another range variable.

5.8. Deletions

Deletions are performed using the **delete** command:

```
* delete EMP where EMP.salary > 0 \g
```

Since all employees have positive salaries, this command will leave the EMP class empty.

One should be wary of queries of the form

in a more efficient manner, but this is invisible to the user.

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.

POSTQUEL does not assign any meaning to duplicate values in such expressions. This means that POSTGRES sometimes recomputes the same target list several times — this frequently happens when Boolean expressions are connected with an **or**. To remove such duplicates, you must use the **retrieve unique** statement. See the Reference Manual under **retrieve** for more details.

```
delete 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.

Before going on, repopulate your EMP database using the **append** commands listed above.

5.9. Using Functions

POSTQUEL queries can contain function calls as well as operators. If we wanted to express our very first **retrieve** query as:

```
* retrieve (EMP.name) where int4lt(EMP.age, 35) \g
```

name
Sam

we could do so. Obviously, if we need to compute some function of more than two arguments, we *must* use the function syntax instead of the operator syntax.

5.10. Using Aggregate Functions

Like most other query languages, POSTGRES supports aggregate functions. However, the current implementation of POSTGRES aggregate functions is very limited. Specifically, while there are aggregates to compute such functions as the count, sum, average, maximum and minimum over a set of instances, aggregates can only appear in the target list of a query and not in the qualification (`where` clause). As an example,

```
* retrieve (how_many = count{EMP.name}) \g
```

how_many
4

counts all employees, and

```
* retrieve (avg_salary =
      int4ave{EMP.salary
              where EMP.dept = "toy"}) \g
```

avg_salary
1200

computes the average salary of all employees in the toy department. However, the following query (to find out who makes more money than any of the toy department employees) will **not** work:

```
* retrieve (EMP.name) where
      EMP.salary > int4max{EMP.salary
                           where EMP.dept = "toy"} \g

WARN:Mar  3 00:40:54:parser: syntax error at or near "{"
```

because the aggregate is not in the target list. In addition, if the qualification of the aggregate expression contains any join clauses (references to other classes), the aggregate may or may not return the right result. (In other words, aggregates with join clauses are neither disallowed nor are they correctly supported.) See the Reference Manual under **postquel** for more details.

5.11. Help! What Are the Valid Types, Operators and Functions?

So far, we have been rather cavalier in our use of types (such as `char16`), operators (such as <), and aggregate functions (such as `count`). A large number of pre-defined types, operators and aggregates are available by default in POSTGRES, and these are listed in the section of the Reference Manual labelled **built-in**. This would be a good time to go ahead and take a peek at that section.

In a later section of this manual, we will describe how to query the system to find out the current list of *all* valid types, operators, functions, etc. known to the system.

6. ADVANCED POSTQUEL FEATURES

Having covered the basics of using POSTQUEL 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).

6.1. Inheritance

First, if you haven't done so already, re-populate the EMP class by repeating the **append** commands in section 5.3. Then, create a second class STUD_EMP, and populate it as follows:

```
* create STUD_EMP (location = point) inherits (EMP) \g

* append STUD_EMP (name = "Sunita", salary = 4000,
                   age = 23, dept = "electronics",
                   location = "(3, 5)") \g
```

In this case, an instance of STUD_EMP *inherits* all data fields (name, salary, age, and dept) from its parent, EMP. Furthermore, student employees have an extra field, location, that shows their address as a coordinate pair. In POSTGRES, a class can inherit from zero or more other classes,. rr * 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 the employees over 20:

```
* retrieve (E.name) from E in EMP where E.age > 20 \g
```

name
Claire
Joe
Sam
Bill

On the other hand, to find the names of all employees, including student employees, over age 20, the query is:

```
* retrieve (E.name) from E in EMP* where E.age > 20 \g
```

which returns:

name
Claire
Joe
Sam
Bill
Sunita

Here the * after EMP indicates that the query should be run over EMP and all classes below EMP in the inheritance hierarchy. Many of the commands that we have already discussed — **retrieve**, **replace** and **delete** — support this * notation, as do others, such as the **rename** and **addattr** commands. See the Reference Manual entries for these commands for additional details.

Note that location in STUD_EMP is not a traditional relational data type. As we will see later, POSTGRES can be customized with an arbitrary number of user-defined data types.

6.2. Time Travel

POSTGRES supports the notion of *time travel*. This feature allows a user to run historical queries. For example, to find Sam's current salary, one would query:

```
* retrieve (E.salary) from E in EMP["now"]
      where E.name = "Sam" \g
```

salary
1200

POSTGRES will automatically find the version of Sam's record valid at the correct time and get the appropriate salary.

One can also give a time *range*. For example to see all the salaries that Sam has ever earned, one would query:

```
* retrieve (E.salary)
      from E in EMP["epoch", "now"]
      where E.name = "Sam" \g
```

where "epoch" indicates the beginning of the system clock.. rr * If you have executed all of the examples so far, then the above query returns:

salary
1200
1200

Notice that there are two salaries for Sam because he was deleted from and then re-appended to the EMP class.

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 "[,]." See Section 3 of the Reference Manual, **Built-Ins**, and the introduction to Section 4, **POSTQUEL**, for a full description of the time types (absolute time, relative time and time ranges).

6.3. 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 or *sets* of any type.

6.3.1. 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 SAL_EMP (name = text,
                  pay_by_quarter = int4[],
                  schedule = char16[][]) \g
```

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 appends; 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.

```
* append SAL_EMP (name = "Bill",
     pay_by_quarter[4] = "{10000, 10000, 10000, 10000}",
     schedule[7][2]    = "{{"meeting", "lunch"}, {}}") \g

* append SAL_EMP (name = "Carol",
        pay_by_quarter = "{20000, 25000, 25000, 25000}",
        schedule[5][2] = "{{"talk", "consult"}, {"meeting"}}") \g
```

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*]. Note that the elements of an array do not have to be completely specified. For example, you may have noticed that we did not initialize all of the elements of the attribute schedule above. The value of an uninitialized element is undefined, but it can be updated later using the replace command.

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:

```
* retrieve (SAL_EMP.name)
      where SAL_EMP.pay_by_quarter[1] !=
            SAL_EMP.pay_by_quarter[2] \g
```

name
Carol

This query retrieves the third quarter pay of all employees:

```
* retrieve (SAL_EMP.pay_by_quarter[3]) \g
```

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 three days of the week. ill.

```
* retrieve (SAL_EMP.schedule[1:3][1:1])
      where SAL_EMP.name = "Bill" \g
```

schedule
{{"meeting"},{""},{""}}

Similarly, the replace command can be used to update a single array element or an arbitrary subarray. This query updates Carol's schedule for the second and third day of the week.

```
* replace SAL_EMP (schedule[2:3][1:2] =
                    "{{"debugging", "shopping"}, {"meeting", "present"}}")
                    where SAL_EMP.name = "Carol" \g
```

This query gives a $1000 raise in the first quarter to all members whose first item on schedule for the first working day is debugging:

```
* replace SAL_EMP (pay_by_quarter[1] =
                    SAL_EMP.pay_by_quarter[1] + 1000)
      where SAL_EMP.schedule[1][1] = "debugging" \g
```

6.3.2. Sets

Class attributes can also be *sets* that are defined in an *intentional*, or declarative, manner. For example, let's say that we want to create a new kind of department class. A department consists of a department name as well as a **query** that lists all members of the department.

```
* create NEW_DEPT (deptname = char16,
                   members = setof EMP) \g

* append NEW_DEPT (deptname = "shoe",
                   members = "retrieve (EMP.all)
                              where EMP.age >= 40") \g

* append NEW_DEPT (deptname = "toy",
                   members = "retrieve (EMP.all)
                              where EMP.name = \\"Sam\\"") \g

* append NEW_DEPT (deptname = "candy",
                   members = "retrieve (EMP.all)
                              where EMP.name != \\"Sam\\"
                              and EMP.age < 40") \g
```

These amount to our business rules: all people over 40 work in the shoe department, Sam works alone in the toy department, and everyone else works in the candy department.

We can retrieve (but not update) individual attributes of each member of a set-valued attribute. We do with the *nested-dot* notation.

```
* retrieve (NEW_DEPT.deptname,
            NEW_DEPT.members.name) \g
```

deptname	name
shoe	Claire
shoe	Joe
toy	Sam
candy	Bill

That is, we project attributes from our set-valued attribute, `NEW_DEPT.members`, by adding the reference to the `EMP` attribute `.name`. There are two caveats: the shorthand `.all` doesn't work for set-valued attributes, and retrieval of more than one attribute from a set-valued attribute may produce unexpected results.

The main advantage of representing sets in a declarative way (instead of storing the actual values, or `EMP`s, in this example) is that the set declarations automatically maintain their consistency. If we hire someone new, they will be assigned to the proper `NEW_DEPT` whether we explicitly give them a department or not.

```
/* whoops, we forgot to put Ginger in a department... */
```

```
* append EMP (name = "Ginger", salary = 2000,
              age = 34) \g

/* ...but it's ok */
* retrieve (NEW_DEPT.deptname,
            NEW_DEPT.members.name) \g
```

deptname	name
shoe	Claire
shoe	Joe
toy	Sam
candy	Bill
candy	Ginger

Notice that POSTGRES returns several results for each of the departments that have more than one employee. This is because POSTGRES "flattens" the result when a set attribute contains multiple instances. In other words, an instance is returned for each of the set elements and the contents of the other attributes (in this case, `deptname`) is just duplicated in each of those instances.

7. EXTENDING POSTQUEL: AN OVERVIEW

In the sections that follow, we will discuss how you can extend the POSTQUEL query language by adding:

- functions
- types
- operators
- aggregates

We will then give some integrated examples of their use.

7.1. 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 tables, 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 tables can be modified by the user, and since POSTGRES bases its internal operation on these tables, 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 the POSTQUEL query language 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.

7.2. 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.

7.3. 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 POSTQUEL)
`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 **define** command may modify

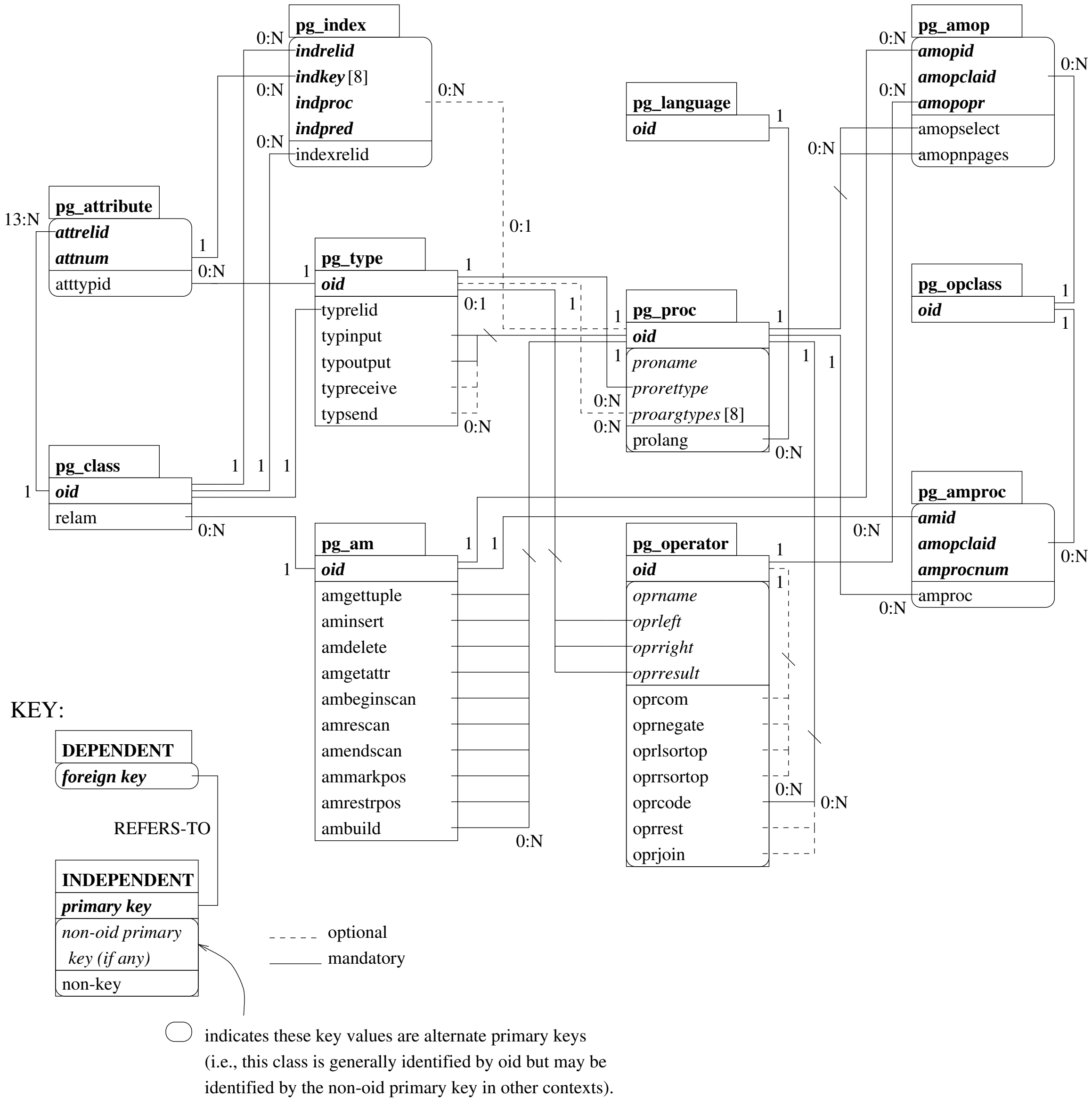

Figure 3. The major POSTGRES system catalogs.

many of these catalogs.

(3) Types and procedures. rr * are central to the schema. Nearly every catalog contains some reference to instances in one or both of these classes. For example,

procedure and *function* more or less interchangably.

POSTGRES frequently uses type signatures (e.g., of functions and operators) to identify 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.

8. EXTENDING POSTQUEL: 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.

POSTQUEL provides two types of functions: *query language functions* (functions written in POSTQUEL) 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 POSTQUEL functions, so we'll start with those.

8.1. Query Language (POSTQUEL) Functions

8.1.1. POSTQUEL Functions on Base Types

The simplest possible POSTQUEL function has no arguments and simply returns a base type, such as `int4`:

```
* define function one
      (language = "postquel", returntype = int4)
      as "retrieve (one = 1)" \g


* retrieve (answer = one()) \g
```

answer
1

Notice that we defined a target list for the function (with the name `one`), 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 POSTQUEL 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` and specify their types using the `arg is` clause.

```
* define function add_pq
      (language = "postquel", returntype = int4)
      arg is (int4, int4)
      as "retrieve (sum = $1 + $2)" \g


* retrieve (answer = add_pq(1, 2)) \g
```

answer
3

8.1.2. POSTQUEL 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 we must also specify the attributes of that argument. For example, take the function `double_salary` that computes what your salary would be if it were doubled.

```
* define function double_salary
      (language = "postquel", returntype = int4)
      arg is (EMP)
      as "retrieve (salary = $1.salary * 2)" \g

* retrieve (EMP.name, dream = double_salary(EMP))
      where EMP.dept = "toy" \g
```

name	dream
Sam	2400

This is pretty straightforward. 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:
 * retrieve (youngster = EMP.name))
 *    where EMP.age < 30
 */
* retrieve (youngster = name(EMP))
      where age(EMP) < 30 \g
```

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:

```
* define function new_emp
      (language = "postquel", returntype = EMP)
      as "retrieve (name = \\"None\\"::text,
                    salary = 1000,
                    age = 25,
                    dept = \\"none\\"::char16)"
```

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 fields appear in the **create** statement (or when you execute a `.all` query).
- You must be careful to typecast the fields (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.all)
```

  See the Reference Manual under **postquel** for a discussion of typecasting.
- 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.

```
* retrieve (nobody = name(new_emp())) \g
```

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.

```
* retrieve (nobody = new_emp().name) \g
WARN:Mar  3 03:09:28:parser: syntax error at or near "."
```

Any collection of commands in the POSTQUEL query language can be packaged together and defined as a function. The commands can include updates (i.e., **append**, **replace** and **delete**) as well as **retrieve** queries. However, the final command must be a **retrieve** that returns whatever is specified as the function's `returntype`.

```
* define function clean_EMP (language = "postquel",
                             returntype = int4)
      as "delete EMP where EMP.salary <= 0
          retrieve (ignore_this = 1)" \g

* retrieve (x = clean_EMP()) \g
```

x
1

8.1.3. POSTQUEL Functions on Sets

Unfortunately, POSTGRES does not really distinguish between functions that return single instances and those that return sets of instances. In all cases, instances are returned one-by-one. Similarly, functions can only take single instances as their arguments and cannot have sets as an argument. For example, the following function `high_pay` returns the set of all employees in class `EMP` whose salaries exceed 1500:

```
* define function high_pay
      (language = "postquel", returntype = setof EMP)
      as "retrieve (EMP.all) where EMP.salary > 1500" \g

* retrieve (overpaid = name(high_pay())) \g
```

overpaid
Claire
Bill
Ginger

However, this function could be defined with

```
returntype = EMP
```

with exactly the same results.

8.2. Programming Language Functions

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/postgres/include` (see Figure 2). You should always include

```
-I/usr/local/postgres/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/postgres/src/backend
-I/usr/local/postgres/src/backend/port/<PORTNAME>
-I/usr/local/postgres/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 `tmp/c.h` and `tmp/postgres.h`, so it's usually a good idea to include those files 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.

8.2.1. 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 "tmp/c.h"
```

```
#include "tmp/postgres.h"
#include "utils/palloc.h"

...

void *buffer; /* our source data */

...

text *destination = (text *) palloc(sizeof(int4) + 40);
destination->length = sizeof(int4) + 40;
bcopy(buffer, destination->data, 40);

...
```

Now that we've gone over all of the possible structures for base types, we can show some examples of real functions.

```
#include <string.h>

#include "tmp/c.h"
#include "tmp/postgres.h" /* for char16, etc. */
#include "utils/palloc.h" /* for palloc */

int
add_one(arg)
    int arg;
{
    return(arg + 1);
}

char16 *
concat16(arg1, arg2)
    char16 *arg1, *arg2;
{
    char16 *new_c16 = (char16 *) palloc(sizeof(char16));

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

text *
copytext(t)
    text *t;
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
```

```
    bzero((char *) new_t, VARSIZE(t));

    /*
     * VARDATA is a pointer to the data region of the struct.
     * VARLEN is the size of VARDATA in bytes (so it's always
     * VARSIZE - sizeof(int4)).
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARLEN(t));              /* how many bytes */
    return(new_t);
}
```

On ULTRIX we would type:

```
* define function add_one
      (language = "C", returntype = int4)
      arg is (int4)
      as "/usr/local/postgres/src/examples/chapter8.o" \g

* define function concat16
      (language = "C", returntype = char16)
      arg is (char16, char16)
      as "/usr/local/postgres/src/examples/chapter8.o" \g

* define function copytext
      (language = "C", returntype = text)
      arg is (text)
      as "/usr/local/postgres/src/examples/chapter8.o" \g
```

On other systems, we might have to make the filename end in `.so` or `.sl` (to indicate that it's a shared library).

8.2.2. 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

```
* retrieve (EMP.all) where c_overpaid(EMP) \g
```

In the query above, we can define `c_overpaid` as:

```
#include <tmp/c.h>
#include <tmp/postgres.h>
#include <tmp/libpq-fe.h> /* for TUPLE */
```

```
bool
c_overpaid(t, limit)
    TUPLE t;   /* the current instance of EMP */
    int4 limit;
{
    bool isnull = false;
    int4 salary;

    salary = (int4) GetAttributeByName(t, "salary", &isnull);

    if (isnull == true)
         return((bool) false);
    return((bool) (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 POSTQUEL 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:

```
* define function c_overpaid
      (language = "c", returntype = bool)
      arg is (EMP, int4)
      as "/usr/local/postgres/src/examples/overpaid.o" \g
```

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. See the document

```
/usr/local/postgres/src/doc/implementation/am.me
```

for details.

8.2.3. Programming Language Functions on Sets

No interface has been defined for passing a set of instances into a function as an argument to a C function, nor is there such an interface for returning a set of instances from a C function.

9. EXTENDING POSTQUEL: TYPES

As previously mentioned, there are two kinds of types in POSTGRES: *base* types (defined in a programming language) and *composite* types (instances).

9.1. User-Defined Types

9.1.1. 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 representation of the type. The output function takes the internal representation of the type and returns a null-delimited character string.

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! This is easy in some cases, or if we are lazy. For example, an input function for `int4` can be as simple as:

```
int4
int4_input(s)
    char *s;
{
    return(atoi(s));
}
```

if we cheat and use the C library function `atoi` (and don't do any checks for such errors as out-of-range integers). The output function can be almost as simple:

```
char *
int4_output(i)
    int4 i;
{
    /* the largest 32-bit number is 10 digits long */
    char *buf = palloc(11);

    (void) sprintf(buf, "%d", i);
    return(buf);
}
```

(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.

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.

9.1.2. Large Objects

The types discussed to this point are all ‘‘small’’ objects — that is, they are smaller than 8KB. rr * 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. The interface to large objects is quite similar to the UNIX file system interface. The particulars are detailed in Section 7 of the POSTGRES Reference Manual.

---

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.

9.2. Composite Types

Instances of a composite type are just instances of a class. Here, we discuss how to create attributes of one class that are composed of one or more instances of a composite type (another class). We can do this using *set-valued attributes* or by using functions to create *virtual attributes*.

We have already discussed how to define a set-valued attribute using the `setof` keyword in the `create` command. This produces an attribute whose value is procedurally defined using a query.

Since POSTQUEL functions return instances or sets of instances, they can also be used to create "attributes" of composite types. For example, consider extending the `EMP` class with a `manager` field. That is, for each instance of `EMP`, we want to associate another instance of `EMP` corresponding to the manager of the first instance. Specifically, we will define a POSTQUEL function `manager`:

```
* define function manager
      (language = "postquel", returntype = EMP)
      arg is (EMP)
      as "retrieve (E.all) from E in EMP
              where E.name = DEPT.manager
              and DEPT.dname = $1.dept" \g
```

When a function takes a single composite type argument, POSTQUEL allows us to use the same *nested-dot* notation we used for sets to refer into an instance returned by the function. Here, the function `manager` takes an `EMP` instance as its only argument, we can write the query

```
* retrieve (EMP.name)
      where name(manager(EMP)) = "Claire" \g
```

as

```
* retrieve (EMP.name)
      where EMP.manager.name = "Claire" \g
```

In either case, we get

name
Claire
Joe

We have essentially added an attribute to the `EMP` class which is of type `EMP`, i.e., it has a value which is an instance of the class `EMP`. The limitations discussed for set-valued attributes generally apply to virtual attributes as well. For example, one cannot do direct updates to such columns. That is,

```
* append EMP (manager.name = "Smith") \g
WARN:Mar 10 22:48:42:manager: no such class
```

won't work. Non-projected retrieves don't work either. For example, queries that attempt to retrieve the entire `manager` attribute, such as

```
* retrieve (EMP.manager) \g
```

don't return anything useful.

Note that `manager` is defined as returning a single instance of `EMP`. We can also write a POSTQUEL function that returns sets of instances. For example, consider the function

```
* define function same_dept
      (language = "postquel", returntype = setof EMP)
      arg is (EMP)
      as "retrieve (E.all) from E in EMP
              where $1.dept = E.dept" \g
```

The `same_dept` function is defined as returning a set of instances, rather than a single instance. Given the query:

```
* retrieve (EMP.name, EMP.same_dept.name) \g
```

name	name
Claire	Claire
Claire	Joe
Joe	Claire
Joe	Joe
Sam	Sam
Bill	Bill
Ginger	(null)

the query in the body of the `same_dept` function returns many instances and the **retrieve** query will return all of them in a ‘‘flattened’’ form.

10. EXTENDING POSTQUEL: OPERATORS

POSTQUEL 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. (For a discussion of typecasting, see the Reference Manual under **postquel**).

In this example, we will use some functions that are already built into POSTGRES to define a set of operators that all have the same name, `##`. First, we define left unary operators on both `int4` and `int2` that have very different meanings. To do this, we will use some mathematical functions that already happen to be built into POSTGRES. `int4fac`, `int2um/int4um` and `int4pl` are functions that calculate integer factorial, unary minus and addition, respectively.

```
/* n! (factorial) for int4 */
* define operator ## (arg2 = int4,
                      associativity = right,
                      procedure = int4fac)
 \g

/* -n (negation) for int2 */
* define operator ## (arg2 = int2,
                      associativity = right,
                      procedure = int2um)
 \g
```

Next, we define a right unary operator:

```
/* -n (negation) for int4 */
* define operator ## (arg1 = int4,
                      associativity = left,
                      procedure = int4um)
 \g
```

Finally, we define a binary operator:

```
/* a+b (addition) for int4 */
* define operator ## (arg1 = int4,
                      arg2 = int4,
                      procedure = int4pl,
                      commutator = ## )
 \g
```

If we give the system enough type information, it can automatically figure out which operators to use. In this case, we can take advantage of the fact that plain "numbers" default to the `int4` type to get the following behavior:

```
* retrieve (four_factorial = ## 4,
            minus_five = ## 5::int2,
            minus_four = 4 ##,
            four_plus_four = 4 ## 4)
 \g
```

four_factorial	minus_five	minus_four	four_plus_four
24	-5	-4	8

11. EXTENDING POSTQUEL: AGGREGATES

Creation of user-defined aggregates is explained in the Reference Manual under **define aggregate**. The key observation to be made, however, is that any aggregate can be 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 **define 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.

```
* define aggregate my_sum (sfunc1 = int4pl, /* addition */
                           basetype = int4,
                           stype1 = int4,
                           initcond1 = "0") \g

* retrieve (salary_sum = my_sum{EMP.salary}) \g
```

salary_sum
8200

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.

```
* define aggregate my_count (sfunc2 = int4inc, /* add one */
                             stype2 = int4,
                             initcond2 = "0") \g

* retrieve (emp_count = my_count{EMP.oid}) \g
```

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.

```
* define aggregate my_average (sfunc1 = int4pl, /* sum */
                               basetype = int4,
                               stype1 = int4,
                               sfunc2 = int4inc, /* count */
                               stype2 = int4,
                               finalfunc = int4div, /* division */
                               initcond1 = "0",
                               initcond2 = "0") \g

* retrieve (emp_average = my_average{EMP.salary}) \g
```

emp_average
1640

12. EXTENDING POSTQUEL: AN EXAMPLE

In this discussion, we will be defining a `circle` type, using functions written in the C programming language.

For additional examples of how to create new types, functions and operators, you should look in the directories

```
/usr/local/postgres/src/regress/demo
/usr/local/postgres/src/regress/regress
/usr/local/postgres/src/regress/video
```

These directories contain several C and POSTQUEL files that should how to perform various extensions to the system, and the routines we use in our regression tests should always work.

12.1. C Data Structures

Before we do anything, we have to decide on what a circle looks like, both in string format and internally in memory. Circles have a center and a radius, so a reasonable string representation of a circle would be an ordered triple:

(center_x, center_y, radius)

where each element is a real number with arbitrary units, e.g.:

```
(5.0, 10.3, 3)
```

This is what the input to the circle input function looks like, and what the output from the circle output function looks like.

Now we have to come up with an internal representation for a circle in memory. The following declarations are legal and reasonable given the format we chose above:

```
typedef struct {
    double x, y;
} POINT;

typedef struct {
    POINT center;
    double r;
} CIRCLE;
```

Memory containing values of type `CIRCLE` will be written to disk and read from disk, so `CIRCLE` must be both *complete* and *contiguous*; that is, it cannot contain any pointers.

The type definition

```
typedef struct {
    POINT *center   /* NO! */
    double r;
} CIRCLE;
```

will *NOT* work, because the virtual memory *address* stored in `center` would be written to disk instead of the contents of the `POINT` structure to which `center` presumably points. POSTGRES cannot detect this kind of coding error; you must guard against it yourself.

12.2. Defining the Input and Output Functions

Suppose in defining our type "circle," we have a C source file called `circle.c`, and a corresponding object code file `/usr/local/postgres/src/examples/circle.o`. (All functions related to our `circle` type must be in the same object file.) For the purposes of this discussion, suppose our platform is a MIPS DECstation, where `sizeof(double)` is 8 bytes. This assumption will be important later.

We will create source file `circle.c`, containing C source code for the functions that support our `CIRCLE` type. `circle.c` contains three functions:

- `circle_in`, which is the input function for circles. It takes a C string as an argument and returns a pointer to a `CIRCLE`.
- `circle_out`, which is the output function for circles. It is takes a pointer to a `CIRCLE` as input and returns a C string. The return value of `circle_in` must be a legal argument to `circle_out`, and vice versa.
- `eq_area_circle`, which is the equality function for circles. For the purposes of this discussion, circles are equal if their areas are equal.

The contents of `circle.c` are:

```
#include <math.h>
#include <stdio.h>
#include <string.h>

#include "tmp/c.h"            /* (always)                  */
#include "utils/geo-decls.h" /* for POINT declaration    */
#include "utils/palloc.h"    /* for palloc() declaration */

typedef struct {
    POINT  center;
    double radius;
} CIRCLE;

#define LDELIM '('
#define RDELIM ')'
#define NARGS  3

CIRCLE *
circle_in(str)
```

```
    char   *str;
{
    char   *p, *coord[NARGS];
    int    i;
    CIRCLE *result;

    if (str == (char *) NULL)
            return((CIRCLE *) NULL);

    for (i = 0, p = str;
         *p && i < NARGS && *p != RDELIM;
         p++)
    {
        if (*p == ',' || (*p == LDELIM && !i))
            coord[i++] = p + 1;
    }

    if (i < NARGS - 1)
            return((CIRCLE *) NULL);

    result = (CIRCLE *) palloc(sizeof(CIRCLE));

    result->center.x = atof(coord[0]);
    result->center.y = atof(coord[1]);
    result->radius = atof(coord[2]);

    return(result);
}

char *
circle_out(circle)
    CIRCLE *circle;
{
    char   *result;

    if (circle == (CIRCLE *) NULL)
            return((char *) NULL);

    result = (char *) palloc(60);

    sprintf(result, "(%g, %g, %g)",
            circle->center.x, circle->center.y,
            circle->radius);

    return(result);
}

int
eq_area_circle(circle1, circle2)
    CIRCLE *circle1, *circle2;
{
    if (circle1 == (CIRCLE *) NULL)
```

```
                return(circle2 == (CIRCLE *) NULL);
        if (circle2 == (CIRCLE *) NULL)
                return(0);
        return(circle1->radius == circle2->radius);
}
```

Now that we have written these functions and compiled the source file, we have to let POSTGRES know that they exist. First, we run the following queries to define the input and output functions. These functions must be defined *before* we define the type. POSTGRES will notify you that return type circle is not defined yet, but this is OK. Notice that we use the keyword **any** to indicate that the input and/or output of the function is not a POSTGRES type (e.g., a simple C string).

```
* define function circle_in
      (language = "c", returntype = circle)
      arg is (any)
      as "/usr/local/postgres/src/examples/circle.o" \g

* define function circle_out
      (language = "c", returntype = any)
      arg is (any)
      as "/usr/local/postgres/src/examples/circle.o" \g
```

Note that the full pathname of the object code file must be specified, so you would change `/usr/local/postgres` to whatever is appropriate for your installation.

Now we can define the `circle` type:

```
* define type circle
      (internallength = 24,
       input = circle_in, output = circle_out) \g
```

where `internallength` is the size of the `CIRCLE` structure in bytes. For circles, the type members are three `doubles`, which on most platforms are 8 bytes each, with no additional alignment constraints. However, when defining your own types, you should *not* make assumptions about structure sizes, but instead write a test program that does something like

```
printf("size is %d\n", sizeof(MYTYPE));
```

on your type.

If `internallength` is defined incorrectly, you will encounter strange errors and may crash the server. If this were to happen with our `CIRCLE` type, we would have to do a

```
* remove type circle \g
```

and then redefine the `circle` type correctly. Note that we would *not* have to redefine our functions, since their behavior would not have changed.

12.2.1.1. Defining Operators

Now that we have finished defining the `circle` type, we can **create** classes with circles in them, **append** records to them with circles defined, and **retrieve** the values of the entire list of records. However, we can't do anything terribly useful with them until we have some operators and/or functions. To do this, we make use of the concept of *operator overloading*, and in this case we will set the POSTGRES equality operator "=" to work for circles. First we have to tell POSTGRES that our circle equality function exists:

```
* define function eq_area_circle
      (language = "c", returntype = bool)
      arg is (circle, circle)
      as "/usr/local/postgres/src/examples/circle.o" \g
```

We will now bind this function to the equality symbol with the following query:

```
* define operator =
      (arg1 = circle, arg2 = circle,
       procedure = eq_area_circle) \g
```

12.2.1.2. Using a New Type

Let's create a class `tutorial` that contains a `circle` attribute, and run some queries against it:

```
* create tutorial (a = circle) \g

* append tutorial (a = "(1.0, 1.0, 10.0)"::circle) \g

* append tutorial (a = "(2.0, 2.0, 5.0)"::circle) \g

* append tutorial (a = "(0.0, 1.8, 10.0)"::circle) \g

* retrieve (tutorial.all)
      where tutorial.a = "(0.0, 0.0, 10.0)"::circle \g
```

which returns:

a
(1.0, 1.0, 10.0)
(0.0, 1.8, 10.0)

Recall that we defined circles as being equal if their areas were equal.

Other operators (less than, greater than, etc.) can be defined in a similar way. Note that the = symbol will still work for other types — it has merely had a new type added to the list of types it works on.

13. 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:

```
* retrieve (pg_am.oid) where pg_am.amname = "btree" \g
```

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 POSTQUEL 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 a field called `amsupport`. This field 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.. rr *

The `amstrategies` entry in `pg_am` is just the *number* of strategies defined for the access method in question. The procedures for less than, less equal, and so on don't appear in `pg_am`. Similarly, `amsupport` is just the number of support routines required by the access method. The actual routines are listed elsewhere.

The next class of interest is `pg_opclass.` This class exists only to associate a name with an `oid`. In `pg_amop`, every B-tree operator class has a set of procedures, one through five, above. Some existing opclasses are `int2_ops`, `int4_ops`, and `oid_ops`. You need to add an instance with your opclass name (for example,

0), 0, or a non-zero positive number (> 0).

int4_abs_ops) to pg_opclass. The oid of this instance is a foreign key in other classes.

```
* append pg_opclass (opcname = "int4_abs_ops") \g

* retrieve (cl.oid, cl.opcname) from cl in pg_opclass
      where cl.opcname = "int4_abs_ops" \g
```

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 int4_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/postgres/src/examples/int4_abs.c
```

The code is

```
/*
 * int4_abs.c -- absolute value comparison functions
 *               for int4 data
 */

#include "tmp/c.h"

#define ABS(a) ((a < 0) ? -a : a)

/* routines to implement operators */

bool int4_abs_lt(a, b) int32 a, b;
     { return(ABS(a) < ABS(b)); }

bool int4_abs_le(a, b) int32 a, b;
     { return(ABS(a) <= ABS(b)); }

bool int4_abs_eq(a, b) int32 a, b;
     { return(ABS(a) == ABS(b)); }
```

```
bool int4_abs_ge(a, b) int32 a, b;
     { return(ABS(a) >= ABS(b)); }

bool int4_abs_gt(a, b) int32 a, b;
     { return(ABS(a) > ABS(b)); }

/* support (signed comparison) routine */

int int4_abs_cmp(a, b) int32 a, b;
     { return(ABS(a) - ABS(b)); }
```

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.

```
* define function int4_abs_lt
      (language = "c", returntype = bool)
      arg is (int4, int4)
      as "/usr/local/postgres/src/examples/int4_abs.o" \g

* define function int4_abs_le
      (language = "c", returntype = bool)
      arg is (int4, int4)
      as "/usr/local/postgres/src/examples/int4_abs.o" \g

* define function int4_abs_eq
      (language = "c", returntype = bool)
      arg is (int4, int4)
      as "/usr/local/postgres/src/examples/int4_abs.o" \g

* define function int4_abs_ge
      (language = "c", returntype = bool)
      arg is (int4, int4)
      as "/usr/local/postgres/src/examples/int4_abs.o" \g

* define function int4_abs_gt
      (language = "c", returntype = bool)
```

```
        arg is (int4, int4)
        as "/usr/local/postgres/src/examples/int4_abs.o" \g
```

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 &
 */
* retrieve (o.all)
    from o in pg_operator
    where o.oprname ~ "^..&$"::text \g
```

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.

```
* define operator <<&
      (arg1 = int4, arg2 = int4, procedure=int4_abs_lt,
       associativity = left, restrict = intltsel,
       join = intltjoinsel) \g

* define operator <=&
      (arg1 = int4, arg2 = int4, procedure = int4_abs_le,
       associativity = left, restrict = intltsel,
       join = intltjoinsel) \g

* define operator ==&
      (arg1 = int4, arg2 = int4, procedure = int4_abs_eq,
       associativity = left, restrict = eqsel,
       join = eqjoinsel) \g

* define operator >=&
      (arg1 = int4, arg2 = int4, procedure = int4_abs_ge,
       associativity = left, restrict = intgtsel,
       join = intgtjoinsel) \g

* define operator >>&
      (arg1 = int4, arg2 = int4, procedure = int4_abs_gt,
       associativity = left, restrict = intgtsel,
       join = intgtjoinsel) \g
```

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 `oid`s of the operators for the opclass (which we'll get in just a minute)
`amopselect`, `amopnpages`	cost functions.

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

So we need the `oid`s of the operators we just defined. We'll look up the names of all the operators that take two `int4`s, and pick ours out:

```
* retrieve (o.oid, o.oprname)
      from o in pg_operator, t in pg_type
      where o.oprleft = t.oid and o.oprright = t.oid
          and t.typname = "int4" \g
```

which returns:

oid	oprname
96	\=
97	<
514	*
518	!=
521	>
523	<=
525	>=
528	/
530	%
551	+
555	-
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. Recall that the B-tree instance's oid is 403 and int4_abs_ops is oid 17314. Then we add the instances we need:

```
* append pg_amop
      (amopid = "403"::oid,          /* btree oid         */
       amopclaid = "17314"::oid,     /* pg_opclass tuple */
       amopopr = "17321"::oid,       /* <<& tup oid      */
       amopstrategy = "1"::int2,     /* 1 is <<&         */
       amopselect = "btreesel"::regproc,
       amopnpages = "btreenpage"::regproc) \g

* append pg_amop (amopid = "403"::oid,
                  amopclaid = "17314"::oid,
                  amopopr = "17322"::oid,
                  amopstrategy = "2"::int2,
                  amopselect = "btreesel"::regproc,
                  amopnpages = "btreenpage"::regproc) \g

* append pg_amop (amopid = "403"::oid,
                  amopclaid = "17314"::oid,
                  amopopr = "17323"::oid,
                  amopstrategy = "3"::int2,
                  amopselect = "btreesel"::regproc,
```

```
                     amopnpages = "btreenpage"::regproc) \g

    * append pg_amop (amopid = "403"::oid,
                     amopclaid = "17314"::oid,
                     amopopr = "17324"::oid,
                     amopstrategy = "4"::int2,
                     amopselect = "btreesel"::regproc,
                     amopnpages = "btreenpage"::regproc) \g

    * append pg_amop (amopid = "403"::oid,
                     amopclaid = "17314"::oid,
                     amopopr = "17325"::oid,
                     amopstrategy = "5"::int2,
                     amopselect = "btreesel"::regproc,
                     amopnpages = "btreenpage"::regproc) \g
```

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.

In the file

```
/usr/local/postgres/src/examples/chapter13
```

we show the POSTQUEL that performs the four-way join between `pg_amop`, `pg_opclass`, `pg_operator` and `pg_type`. Doing the join obviates the need to write down any `oid`s but the query is considerably more complicated-looking.

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):

```
* define function int4_abs_cmp
      (language = "c", returntype = int4)
      arg is (int4, int4)
      as "/usr/local/postgres/src/examples/int4_abs.o" \g

* retrieve (p.oid, p.proname)
      from p in pg_proc
      where p.proname = "int4_abs_cmp" \g
```

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:

```
* append pg_amproc
      (amid = "403"::oid,         /* btree oid         */
       amopclaid = "17314"::oid, /* pg_opclass tuple */
       amproc = "17328"::oid,    /* new pg_proc oid */
       amprocnum = "1"::int2) \g
```

Okay, now it's time to test the new operator class. First we'll create and populate a class.   rr *:

```
* create pairs (name = char16, number = int4) \g

* append pairs (name = "mike", number = -10000) \g

* append pairs (name = "greg", number = 3000) \g

* append pairs (name = "lay peng", number = 5000) \g

* append pairs (name = "jeff", number = -2000) \g

* append pairs (name = "mao", number = 7000) \g

* append pairs (name = "cimarron", number = -3000) \g

* retrieve (pairs.all) \g
```

name	number
mike	-10000
greg	3000
lay peng	5000
jeff	-2000
mao	7000
cimarron	-3000

Okay, looks pretty random. Define an index using the new opclass:

```
* define index pairsind on pairs
      using btree (number int4_abs_ops) \g
```

Now run a query that doesn't use one of our new operators. What we're trying to do here is to run a query that *won't* use our index, so that we can tell the difference when we see a query that *does* use the index. This query won't use the index because the operator we use in the qualification isn't one that appears in the list of strategies for our index.

In fact, POSTGRES uses a "cost-based" query optimizer that makes the decision whether or not to use an index based on how much data is touched. Since this example creates a very small amount of data, the example will likely not work as advertised — one would have to insert a fair amount of data before using an index would actually be cheaper than just scanning the underlying heap data structure. "A fair amount" typically means on the order of several kilobytes.

```
* retrieve (pairs.all) where pairs.number < 9000 \g
```

name	number
mike	-10000
greg	3000
lay peng	5000
jeff	-2000
mao	7000
cimarron	-3000

Yup, just as random; that didn't use the index. Okay, let's run a query that *does* use the index:

```
* retrieve (pairs.all) where pairs.number <<& 9000 \g
```

name	number
jeff	-2000
cimarron	-3000
greg	3000
lay peng	5000
mao	7000

Note that the `number` values are in order of increasing absolute value (as they should be, since the index was used for this scan) and that we got the right answer — the instance for `mike` doesn't appear, because −10000 `>=&` 9000.

14. 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 main point you should understand is that POSTGRES actually has two rule systems, the *instance-level* rule system and the *query rewrite* rule system, and that there are tradeoffs in the employment of each.

The *instance-level* rule system uses markers placed in each instance in a class to "trigger" rules. Examples of the instance-level rule system are explained and illustrated in `/usr/local/postgres/src/regress/demo`, which is included with the POSTGRES distribution. Additional discussion of the instance-level rule system can be found in the Reference Manual under **define rule**.

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. Examples can be found in `/usr/local/postgres/src/regress/video`, and further discussion is in the Reference Manual under **define rule**. The power of this rule system is discussed in [ONG90] as well as [STON90b].

Since each rule system is implemented quite differently, they work best in different situations. The query rewrite system is best when rules affect *most* of the instances in a class, while the instance-level system is best when a rule affects only a *few* instances.

15. 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.

15.1. Frequent Tasks

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

15.1.1. 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 should look at the Reference Manual under the heading **postmaster** if you wish to use non-default options (e.g., increased security options, a non-standard installation directory, etc.).
- 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/postgres/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.

15.1.2. 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.

15.1.3. 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.

15.1.4. Periodic Upkeep

The `vacuum` command should be run on each database periodically. This command processes deleted instances. rr * 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`.

15.1.5. 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

```
retrieve (EMP.all) where EMP.salary < 5000
```

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

```
retrieve (EMP.all) where EMP.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.

(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

---

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.

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**.

15.2. Infrequent Tasks

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

15.2.1. 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/postgres/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 table, 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).

15.2.2. Moving Database Directories

By default, all POSTGRES databases are stored in separate subdirectories under `*/usr/local/postgres/data/base`.. rr 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/postgres/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
```

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.

```
# 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/postgres/data
```

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/postgres/data/base/foo` and its contents to its ultimate destination. It should still be owned by the "postgres" user.

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

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

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

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

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

- Restart the `postmaster`.

15.2.3. 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 POSTQUEL **define** commands. Notice that as of Version 4.2, the method by which you generate object code for user-defined functions has changed, so you may have to modify your old `.o` files. See Appendix A for more details.
- *Data* must be dumped from the old classes into ASCII files (using the POSTQUEL **copy** command), the new classes created in the new database (using the POSTQUEL

create command), and the data reloaded from the ASCII files.

- *Rules* and *views* must also be reloaded by re-executing the various POSTQUEL **define** 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.)

15.3. 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.

15.3.1. 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.

15.3.2. 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.

15.4. 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/postgres/src/examples/chapter15
```

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).

```
* retrieve (user_name = u.usename,
            database = d.datname)
      from u in pg_user,
           d in pg_database
      where u.usesysid = int2in(int4out(d.datdba))
      sort by user_name, database
 \g
```

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

```
* retrieve (class_name = c.relname)
      from c in pg_class
      where c.relkind = ’r’      /* no indices */
        and c.relname !˜ "^pg_"  /* no catalogs */
      sort by class_name
 \g
```

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

```
* retrieve (class_name = bc.relname,
            index_name = ic.relname,
            attr_name = a.attname)
      from bc in pg_class,         /* base class */
           ic in pg_class,         /* index class */
           i in pg_index,
           a in pg_attribute       /* 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 */
```

```
        sort by class_name, index_name,
                attr_name
```

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

```
* retrieve (class_name = c.relname,
            attr_name = a.attname,
            attr_type = t.typname)
      from c in pg_class,
           a in pg_attribute,
           t in pg_type
      where c.relkind = 'r'      /* no indices */
        and c.relname !~ "^pg_" /* no catalogs */
        and a.attnum > 0        /* no system att's */
        and a.attrelid = c.oid
        and a.atttypid = t.oid
      sort by class_name, attr_name
 \g
```

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

```
* retrieve (owner_name = u.usename,
            type_name = t.typname)
      from t in pg_type,
           u in pg_user
      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"
      sort by owner_name, type_name
 \g
```

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

```
* retrieve (left_unary = o.oprname,
            operand = right.typname,
            return_type = result.typname)
      from o in pg_operator,
           right in pg_type,
           result in pg_type
      where o.oprkind = 'l'           /* left unary */
        and o.oprright = right.oid
        and o.oprresult = result.oid
      sort by operand
 \g
```

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

```
* retrieve (right_unary = o.oprname,
            operand = left.typname,
```

```
             return_type = result.typname)
       from o in pg_operator,
            left in pg_type,
            result in pg_type
       where o.oprkind = 'r'          /* right unary */
         and o.oprleft = left.oid
         and o.oprresult = result.oid
       sort by operand
  \g
```

This query lists all binary operators.

```
* retrieve (binary_op = o.oprname,
             left_opr = left.typname,
             right_opr = right.typname,
             return_type = result.typname)
       from o in pg_operator,
            left in pg_type,
            right in pg_type,
            result in pg_type
       where o.oprkind = 'b'         /* binary */
         and o.oprleft = left.oid
         and o.oprright = right.oid
         and o.oprresult = result.oid
       sort by left_opr, right_opr
  \g
```

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 POSTQUEL functions if you change the "C" to "postquel".

```
* retrieve (p.proname,
             arguments = p.pronargs,
             returntype = t.typname)
       from p in pg_proc,
            l in pg_language,
            t in pg_type
       where p.prolang = l.oid
         and p.prorettype = t.oid
         and l.lanname = "C"
       sort by proname
  \g
```

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.

```
* retrieve (aggregate_name = a.aggname,
             type_name = t.typname)
       from a in pg_aggregate,
```

```
            t in pg_type
       where a.aggbasetype = t.oid
       sort by aggregate_name, type_name
  \g
```

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.

```
 * retrieve (access_method = am.amname,
             operator_class = opc.opcname,
             operator_name = opr.oprname)
       from am in pg_am,
            amop in pg_amop,
            opc in pg_opclass,
            opr in pg_operator
       where amop.amopid = am.oid
         and amop.amopclaid = opc.oid
         and amop.amopopr = opr.oid
       sort by access_method, operator_class,
               operator_name
  \g
```

16. REFERENCES

[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.*. rr 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/postgres/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 **define 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.. rr *
- The POSTGRES user must be able to traverse the path given to the **define function** command and be able to read the object file. This is because the POSTGRES server runs as the POSTGRES user, not as the user who starts up the frontend process. (Making the file or a higher-level directory unreadable and/or unexecutable by the

---

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.

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!

"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 **define function** command, one must give it the name of the shared object file (ending in `.so`) rather than the simple object file.. rr *

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.

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

---

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 **define 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`!

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 **define 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.

AIX

AIX, like SunOS, OSF/1 and HP-UX, requires users to build shared object files in order to use its built-in dynamic loading mechanism. No special compiler options must be given to build the simple object file. However, AIX provides a very general, flexible and complicated interface for producing shared object files. As a result, it is (relatively) difficult to produce dynamically-loaded object files. Bear in mind that this only means that it is difficult when compared to the mechanisms just discussed; it's really not that hard to do.

AIX allows the user to tell it which program symbols (e.g., function and global variable names) should be visible to other pieces of code. This can be convenient in certain cases. Unfortunately, AIX also *requires* the user to tell it which symbols should be visible (i.e., the default behavior is to disallow sharing). AIX controls this behavior by using *export files* and *import files*.

> A symbol may be *exported* from the shared object file to the program into which the shared object file is being loaded. In other words, the export file specifies which symbols defined within the shared object file can be accessed by POSTGRES. We usually want all symbols to be visible to POSTGRES.
>
> A symbol may be *imported* by the shared object file from the program into which the shared object file is being loaded. In other words, the import file specifies which symbols defined with the POSTGRES server can be called by routines defined within the shared object file. Again, we usually want all POSTGRES symbols to be

visible to the user code.

Hence, in order to load a shared object file, one must have an export file for the shared object file as well as an import file for the POSTGRES backend server. This turns out to be easy to do, since export and import files have the same basic format and may be produced from the simple object file(s) by running the `mkldexport` command that comes with POSTGRES. The following three steps should work for most cases:

```
# simple AIX example, using Bourne shell
% cc -c foo.c
% mkldexport foo.o `pwd` > foo.exp
% ld -H512 -T512 -o foo.so -e _nostart \
      -bI:/usr/local/postgres/lib/postgres.exp \
      -bE:foo.exp foo.o -lm -lc 2>/dev/null
```

The values given for the `-H`, `-T` and `-e` flags to `ld` should simply be taken as voodoo. The file specified by the `-bI:` flag is produced when the POSTGRES server is compiled and installed. (The library directory `/usr/local/postgres/lib` given in the example may differ if you have installed POSTGRES in a different place, of course.) The file specified by the `-bE:` flag must be produced by hand (using the `mkldexport` command, as shown) before the `.so` shared object file can be produced.. rr * You are probably asking, "If it's so easy, why not do it all for me?!" In fact, the magic command lines given above do work in most cases and so could be embedded within POSTGRES and hidden from the user. However, there are circumstances in which it will fail. In these cases, the user must be able to control the loader flags with which the shared object file is constructed. In addition, since the file system *locations* of the various object files are hardcoded into the export/import files (and hence into the shared object file), this fact should also be visible to the user. Finally, by putting the export/import files under user control, the user can do as the designers of AIX intended and actually edit the files (i.e., control link-editing) as desired.

If you want an actual understanding of how the AIX loader actually works, you should take a look at the tutorials written by Gary Hook at the IBM AIX Systems Center. These tutorials are located in

```
/usr/local/postgres/src/doc/useful/aix-linking.ps
/usr/local/postgres/src/doc/useful/aix-advlink.ps
```

---

functions (see the Reference Manual under the heading **define function**, you must use the `pg_ufp.exp` exports file instead of the `postgres.exp` exports file.