The POSTGRES User Manual

Edited by Jon Rhein, Greg Kemnitz and The POSTGRES Group
EECS Dept.
University of California, Berkeley

1. OVERVIEW

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

1.1. DISTRIBUTION

This manual describes Version 4.1 of POSTGRES. The POSTGRES Group has compiled and tested Version 4.1 on the following platforms:

architecture	operating system
DECstation (MIPS)	ULTRIX V4.2
SPARC	SunOS 4.1.2

1.2. PERFORMANCE

Version 4.1 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 1970's.

1.3. ACKNOWLEDGEMENTS

POSTGRES has been constructed by a team of undergraduate, graduate, and staff programmers. The contributors (in alphabetical order) consisted of Paul Aoki, James Bell, Jennifer Caetta, Jolly Chen, Ron Choi, Adam Glass, Jeffrey Goh, Joey Hellerstein, Wei Hong, Anant Jhingran, Greg Kemnitz, Case Larsen, Jeff Meredith, Michael Olson, Nels Olson, Lay-Peng Ong, Spyros Potamianos, Sunita Sarawagi and Cimarron Taylor.

For version 4.1 Marc Teitelbaum served as chief programmer and was responsible for overall coordination of the project.

The above implementation team contributed significantly to this manual, as did Claire Mosher, Chandra Ghosh, and Jim Frew.

2. INTRODUCTION

Traditional relational 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 are 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. The POSTGRES data model offers substantial additional power by incorporating the following four additional basic constructs:

classes
inheritance
types
functions

The POSTGRES DBMS has been under construction since 1986. The initial concepts for the system were presented in [STON86] and the initial data model appeared in [ROWE87]. The first rule system that was implemented is discussed in [STON88] and the storage manager concepts are detailed in [STON87]. The first "demo-ware" was operational in 1987, and we released Version 1 of POSTGRES to a few external users in June 1989. A critique of version 1 of POSTGRES appears in [STON90]. Version 2 followed in June 1990, and it included a new rule system documented in [STON90B]. Version 4.1, the current version of POSTGRES, is about 200,000 lines of code in the C programming language. POSTGRES is available free of charge, and is being used by approximately 200 sites around the world at this writing.

3. ORGANIZATION

This manual discusses the POSTQUEL query language, including extensions such as user-defined types, operators, and both query language and programming language functions. Arrays of types and functions of an instance are discussed, as well as the POSTGRES rule system. This manual concludes with a discussion on adding an operator class to POSTGRES for use in access methods.

This manual describes the major concepts of the system and attempts to provide an accessible path into using the system. As such, it tries to give examples of the use of the major constructs, so a beginning user does not need to delve immediately into the Reference Manual.

4. WHAT YOU SHOULD READ

This manual is primarily intended to provide a broad overview of the system, as well as to illustrate how programmers would use functions to interact with the POSTGRES "backend." The POSTGRES Reference Manual discusses additional aspects of the system, and provides full syntactic descriptions of every POSTGRES and POSTQUEL command in a format similar to that used in UNIX "man pages."

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 section on LIBPQ if you intend to build a client application around POSTGRES, as this is not discussed at all in this manual.

5. Installing and running POSTGRES

For purposes of examples in this manual it is assumed that POSTGRES has been installed in the directory "/usr/local/postgres" with all the default settings. In practice, however, POSTGRES can be installed anywhere. Also, in all the following examples it is required that the POSTGRES `postmaster` be running. This is because all the commands, such as `createdb`, `monitor`, and others all try to connect to the `postmaster`. If you get the following error message from any of these commands

```
StreamOpen: connect: errno=61
Error: Failed to connect to backend (host=xxx, port=4321)
```

it is because the postmaster is not running. To start the postmaster, type the following command.

```
% postmaster &
```

6. The POSTQUEL Query Language

POSTQUEL is the query language used for interacting with POSTGRES. Here, we give an overview of how to use POSTQUEL to access data. In other sections, user extensions to POSTQUEL will be discussed.

6.1. Creating a database

Once POSTGRES has been installed at your site by following the directions in the release notes, you can create a database named `foo` using 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.

Once you have constructed a database, there are four ways to interact with it:

- You can run the POSTGRES terminal monitor which allows you to interactively enter, edit, and execute commands in the query language POSTQUEL.
- You can interact with POSTGRES from a C program by using the LIBPQ library of subroutine and call facilities. 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 directly execute functions stored in the database. This facility is described in the Reference Manual under "Fast Path."

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

```
% monitor foo
```

(the "`%`" is your UNIX shell prompt.) You will be greeted by the following message:

```
Welcome to the C POSTGRES terminal monitor

Go
*
```

The `Go` indicates the terminal monitor is listening to you and that you can type POSTQUEL commands into a workspace maintained by the monitor. The monitor indicates it is listening by typing `*` as a prompt. Printing the workspace can be performed by typing:

```
* \p
```

and it can be passed to POSTGRES for execution by typing:

```
* \g
```

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 you have the full power of `vi` to make any necessary changes. For more info on using `vi`, type

```
% man vi
```

Once you exit `vi`, your edited query will be in the monitor's query buffer and you can submit it to POSTGRES by using the `\g` command described above.

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

```
* \q
```

and the monitor will respond:

```
I live to serve you.
%
```

For a complete collection of monitor commands, see the manual page on `monitor` in the UNIX section of the Reference Manual.

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

```
% destroydb foo
```

Other DBA commands include `createuser` and `destroyuser`, which are discussed further in the UNIX section of the Reference Manual.

6.2. Classes and the Query Language POSTQUEL

6.2.1. Basic Capabilities

The fundamental notion in POSTGRES is that of a **class,** which is a named collection of instances of objects. Each instance has the same collection of named attributes, and each attribute is of a specific type. Furthermore, each instance has an installation-wide unique (never-changing) **object identifier** or **oid**.

6.2.2. Creating a New Class

(In order to try out the following POSTQUEL examples, create the `foo` database as described in the previous section, and start the terminal monitor.)

A user 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`.)[1] Spaces, tabs and newlines may be used freely in POSTQUEL queries.

So far, the `create` command looks exactly like the create statement in a traditional relational system. However, we will presently see that classes have properties that are extensions of the relational model, so we use a different word to describe them.

6.2.3. Populating a Class with Instances

To populate a class with instances, one can use the `append` command:

```
* 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
```

This will add 3 instances to `EMP`, one for each `append` command.

6.2.4. Querying a Class

The `EMP` class can be queried with normal selection and projection queries. For example, to find the employees under 35 years of age, one would type:

[1] See "Built-In Types" in the Reference Manual.

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

and the output would be:

name
Sam

Notice that parentheses are required around the **target list** of returned attributes (e.g., `EMP.name`.)

POSTQUEL allows you to return computations in the target list as long as they are given a name (e.g., `result`):

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

6.2.5. Redirecting retrieve queries

Any retrieve query can be redirected to a new class in the database, and arbitrary boolean operators (`and`, `or`, `not`) are allowed in the qualification of any query:

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

6.2.6. Joins

To find the names of employees which are the same age, one could write:

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

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. A POSTQUEL query can contain an arbitrary number of class names and surrogates.[2]

6.2.7. Updates

Updates are accomplished in POSTQUEL 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.

[2]The semantics of such a join are that the qualification is a truth expression defined for the Cartesian product of the classes indicated in the query. For those instances in the Cartesian product for which the qualification is true, POSTGRES must compute and return the target list.

6.2.8. Deletions

Deletions are done 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.

6.2.9. Arrays

POSTGRES supports both fixed-length and variable-length one-dimensional arrays. To illustrate their use, we first create a class with an array type.

```
* create SAL_EMP (name = char[],
                  pay_by_quarter = int4[4]) \g
```

The above query will create a class named `SAL_EMP` with a variable-length array of `text` strings (`name`), and an array of 4 `int4` integers (`pay_by_quarter`), which represents the employee's salary by quarter. Now we do some `append`s; note that when appending to a non-character array, we enclose the values within braces and separate them by commas.

```
* append SAL_EMP (name = "bill",
      pay_by_quarter = "{10000, 10000, 10000, 10000}") \g

* append SAL_EMP (name = "jack",
      pay_by_quarter = "{10000, 15000, 15000, 15000}") \g

* append SAL_EMP (name = "joe",
      pay_by_quarter = "{20000, 25000, 25000, 25000}") \g
```

POSTGRES uses the FORTRAN numbering convention for arrays—that is, POSTGRES arrays start with array[1] and end with array[n].

Now, we can run some queries on `SAL_EMP`. 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
```

This query retrieves the third quarter pay of all employees:

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

This query deletes everyone from `SAL_EMP` whose name begins with the letter "j." `SAL_EMP` should now contain only the employee named "bill":

```
* delete SAL_EMP where SAL_EMP.name[1] = 'j' \g
```

Let's make sure (note that the attribute `all` may be used as a shorthand for all attributes of a class):

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

name	pay_by_quarter
bill	{10000,10000,10000,10000}

POSTGRES supports arrays of base and user-defined types, as well as “arrays of arrays,” as in the following example:

```
* create manager (name = char16, employees = text[]) \g

* append manager (name = "mike",
                  employees = "{"wei", "greg", "jeff"}") \g

* append manager (name = "alice",
                  employees = "{"bill", "joe"}") \g

* append manager (name = "marge",
                  employees = "{"mike", "alice"}") \g
```

This creates a class `manager`, and provides a list of employees.

6.3. Advanced POSTQUEL

Now we have covered the basics of using POSTQUEL to access your data. In this section we will discuss those features of POSTGRES which distinguish it from other data managers, such as inheritance and time travel. In the next section we will cover how the user can extend the query language via query language functions and composite objects, as well as additional extensions to POSTGRES using user defined types, operators, and programming language functions.

6.3.1. Inheritance

First, re-populate the `EMP` class by repeating the `append` commands in section 6.2.3. 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,[3] 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

[3]i.e., the inheritance hierarchy is a directed acyclic graph.

query finds the employees over 20:

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

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
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.

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.

6.3.2. 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
```

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["Jan 1 00:00:00 1970 GMT", "now"]
      where E.name = "Sam" \g
```

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

salary
1200
1200

There are two salaries for Sam, since he was deleted from and then re-appended to the

EMP class.

The default beginning of a time range is the origin of the system clock (which just happens to be "Jan 1 00:00:00 1970 GMT" on UNIX systems), and the default end is the current time; thus, the above time range can be abbreviated as "[,]."

7. User Extensions to POSTQUEL

Here, we will discuss user extensions to the POSTQUEL query language, query language functions, composite types, and user defined types, functions and operators.

7.1. User Defined POSTQUEL Functions

POSTQUEL provides two types of functions: **query language functions** (functions written in POSTQUEL) and **programming language functions** (functions written in a separately-compiled programming language such as C.) In this section we will cover POSTQUEL functions; programming language functions will be covered below with the discussion on user-defined types.

Any collection of commands in the POSTQUEL query language can be packaged together and defined as a function, usually returning either a set of instances or a set of base types. For example, the following function high_pay returns all employees in class EMP whose salaries exceed 50,000:

```
* define function high_pay
      (language = "postquel", returntype = setof EMP)
      as "retrieve (EMP.all) where EMP.salary > 50000" \g
```

POSTQUEL functions can also have parameters. The following function large_pay allows the threshold salary to be specified as an argument:

```
* define function large_pay
      (language = "postquel", returntype = setof EMP)
      arg is (int4)
      as "retrieve (EMP.all) where EMP.salary > $1" \g
```

In addition to their obvious utility as "aliases" for commonly-used queries, POSTQUEL functions are useful for creating composite types, as described below.

7.2. Composite Types

Since POSTQUEL functions return instances or sets of instances, they are the mechanism used to assign values to 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
```

The function `manager` takes an instance as its only argument, so POSTQUEL allows referencing into it with the use of the nested dot notation. Whenever such a function is defined over a class, a user can utilize the cascaded dot notation to reference into (i.e. access the fields of) the objects returned by the function.

The following query finds all the employees who work for Joe:

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

This is exactly equivalent to:

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

Here, 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`. Since the value of `manager` has a record-oriented structure, we call it a **composite object**. Consequently, the user can think of the function `manager` as an attribute of `EMP` and can reference it just like any other attribute, with the following two exceptions. First, one cannot do direct `append`s—that is,

```
* append emp (emp.manager.name = "Smith") \g
```

won't work. Non-projected retrieves will also be rejected, i.e.:

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

will result in a warning from the POSTQUEL language parser.

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 children
      (language = "postquel", returntype = setof KIDS)
      arg is (EMP)
      as "retrieve (KIDS.all)
              where $1.name = KIDS.dad
              or $1.name = KIDS.mom"\g
```

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

```
* retrieve(emp.name, emp.children.name)
```

if the query in the body of the `children` function returns many instances, the retrieve query will return all of them, in a "flattened" form. If the query in the body of `manager` returns more than one instance, the `manager` function will return only one instance, arbitrarily chosen from the set returned by the query in the function's body. See the POSTGRES Reference Manual's entry on the `define function` command for further details and examples.

8. User Defined Types, Operators, and Programming Language Functions

The central concept of extending POSTGRES lies in POSTGRES's ability to **dynamically load** a binary object file created by the user. This allows POSTGRES to call arbitrary user functions which can be written in a standard programming language. These functions can then be used:

- to convert between **internal** (binary) and **external** (character string) representations of user-defined types;
- as operators; and
- to define ordering for indices on user-defined types.

POSTGRES's concept of types includes **built-in** types and **user-defined** types. Built-in types are those required by the system to bootstrap itself. User-defined types are those created by the user in the manner described below. There is no intrinsic performance difference between using a system type or user-defined type, other than the overhead due to the complexity of the type itself.

8.1. Internal storage of types

Internally, POSTGRES regards a user-defined type as a "blob of memory" upon which user-defined functions impose structure and meaning. POSTGRES will store and retrieve the data from disk and use user-defined functions to input, process, and output the data.

8.2. Functions needed for a user-defined type

A completely defined user type requires the following user-defined functions:

- **input** and **output** functions for the type: 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. These at least are necessary to define the type.
- **operator** functions for the type: These functions define the meanings of "equal," "less than," "greater than," etc., for your type.

8.3. An Example User Defined Type

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

8.3.1. Data structures for our type

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 alternate declaration

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

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

8.3.2. Defining the input and output functions for our type

Suppose in defining our type "circle," we have a C source file called `circle.c`, and a corresponding object code file `/usr/local/postgres/circle.o`. (All functions related to our `circle` type have to be in the same object file.) For the sake of argument, suppose we our platform is a DECstation, where sizeof(double) is 8 bytes (this 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 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 s `CIRCLE` as input and returns a 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 == NULL) return(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(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 == NULL) return(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;
{
    return(circle1->radius == circle2->radius);
}
```

Now that we have written these functions and compiled the source file,[4] 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[5]:

```
* define function circle_in
      (language = "c", returntype = circle)
      arg is (char16)
      as "/usr/local/postgres/circle.o" \g

* define function circle_out
      (language = "c", returntype = char16)
      arg is (circle)
      as "/usr/local/postgres/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 a

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

[4] You will need to supply an option like `-I/usr/local/postgres/src/backend` to your C compiler so it can find the POSTGRES "`.h`" files. Also, various platform-specific compiler options may be required to support POSTGRES dynamic linking (for example, the DECstation ULTRIX compiler requires the "`-G0`" option.) See "define function" in the Reference Manual for details.

[5] By default, user-defined C functions use addresses instead of values for all but "small" (<= 4-byte) argument and return types, so we can use the POSTQUEL type `char16` as a placeholder for the C type `char *`.

on your type.

If `internallength` is defined incorrectly, you will encounter strange errors which may crash the data manager itself. 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.

8.3.3. Defining an operator for our type

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. But we can do nothing else until we have some circle operators. 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/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
```

8.3.4. Using our 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. Any string of "punctuation characters" other than brackets, braces, or parentheses can be used in defining an operator.

8.4. Additional info on creating a user-defined function

8.4.1. Use palloc and not malloc

In order for POSTGRES to correctly manage memory associated with processing your type, you must use the memory allocator `palloc` and avoid standard UNIX memory managers such as `malloc`. If you do not, POSTGRES will chew up ever increasing amounts of memory. `palloc` has the same arguments as `malloc`, that is

```
char *palloc(size)
unsigned long size;
```

To free memory allocated with `palloc`, use `pfree`, which is analogous to the UNIX library function `free`:

```
void pfree(ptr)
char *ptr;
```

8.4.2. Re-loading user functions

In the process of creating a user-defined type, you may find it necessary to re-load a function in the course of debugging. This is **not** done automatically when you edit or re-compile the file, but **is** done if you quit and restart the data manager.

We would re-load our example functions by using the following command:

```
* load "/usr/local/postgres/circle.o" \g
```

8.4.3. Writing a Function of an Instance

We've already discussed user functions which take POSTGRES base or user defined types as arguments; in this section, we will discuss inheritable C functions or methods.

C language methods are useful particularly when we want to make a function **inheritable**; that is, to have the function process every instance in an inheritance hierarchy of classes.

In using a function of an instance in qualifying an instance, POSTGRES defines the "current instance" to be the instance being qualified at the moment your function is called.

The instance itself will be passed in your function's parameter list as an opaque structure of type TUPLE, and you will use POSTGRES library routines to access the data in the object as described below.[6]

Suppose we want to write a function to answer the query

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

In the query above, a reasonable `overpaid` function might be:

```
bool
overpaid(t)
TUPLE t;   /* the current instance */
{
    extern char *GetAttributeByName();
    short salary, seniority, performance;

    salary = (short) GetAttributeByName(t, "salary");
    seniority = (short) GetAttributeByName(t, "seniority");
    performance = (short) GetAttributeByName(t, "performance");

    return (salary > (seniority * performance));
}
```

`GetAttributeByName` is the POSTGRES system function that returns attributes out of the current instance. It has two arguments: the argument of type TUPLE passed into the function, and the name of the desired attribute. `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")
```

To let POSTGRES know about the `overpaid` function, do:

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

You can have additional complex, base or user-defined types as arguments to the inheritable function. Thus,

```
* retrieve (EMP.all)
```

[6] In POSTGRES 4.1, `TUPLE` is defined as `void *`.

```
            where overpaid2(EMP, DEPT, "bill", 8) \g
```

could be written, and `overpaid2` would be declared:

```
bool
overpaid2(emp, dept, name, number)
    TUPLE emp, dept;
    char *name;
    long number;
```

8.5. Arrays of types

As discussed above, 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.

8.5.1. Arrays of user-defined types

Using the "circle" example discussed above, we will create a class containing an array of circles:

```
* create circles (list = circle[]) \g
```

and do some appends

```
* append circles (list = "{"(1.0, 1.0, 5.0)",
                         "(2.0, 2.0, 10.0)"}") \g

* append circles (list = "{"(2.0, 3.0, 15.0)",
                         "(2.0, 2.0, 10.0)"}") \g

* append circles (list = "{"(2.0, 3.0, 4.0)"}") \g
```

We can now run queries like:

```
* retrieve (circles.list[1]) \g
```

which returns the first element of each `list`:

list
(1, 1, 5)
(2, 3, 4)

and

```
* retrieve (circles.all)
      where circles.list[1] = "(0.0, 0.0, 4.0)" \g
```

which returns:

list
{"(2, 3, 4)"}

Note the {}s, indicating that an array has been retrieved, as opposed to a single element.

8.5.2. Defining a new array type

An array may be defined as an element of a class, as shown above, or it may be defined as a type in and of itself. This is useful for defining **arrays of arrays**.

The special built-in functions `array_in` and `array_out` are used by POSTGRES to input and output arrays of any existing type. Here, we define an array of integers:

```
* define type int_array
      (element = int4, internallength = variable,
       input = array_in, output = array_out) \g
```

The `element` parameter indicates that this is an array, and setting `internallength` to `variable` indicates that the array is a variable-length attribute.[7]

We can use our type defined above to create an array of integer arrays:

```
* define type int_arrays
      (element = int_array, internallength = variable,
       input = array_in, output = array_out) \g

* create stuff (a = int_arrays) \g

* append stuff (a = "{{1, 2, 3} , {4, 5}, {6, 7, 8}}") \g

* append stuff (a = "{{88, 99, 3}}") \g

* append stuff (a = "{{5, 4, 3} , {2, 2}}") \g

* retrieve (stuff.a[1])
      where stuff.a[1][1] < stuff.a[1][2] \g

* retrieve (stuff.a)
      where stuff.a[3][1] < stuff.a[1][2] \g

* retrieve (s.all) from s in stuff
      where s.a[2][2] = stuff.a[1][1] \g
```

We can also define operators for equality, less than, greater than, etc. which operate on our new array type as necessary.

[7]Note that any type using `array_in` and `array_out` **must** be variable-length.

8.5.3. Creating an array type from scratch

There are many situations in which the above scheme for creating an array type is inappropriate, particularly when it is necessary to define a fixed-length array. In this section, we will create an array of four longs called `quarterly`, and a variable-length array of longs called `stats`.[8]

The only special things we need to know when writing the input and output functions for `quarterly` is that POSTGRES will pass a "simple" (i.e. fixed-length) array of `long`s to the output function and expect a simple array of `long`s in return from the input function. A simple array suitable for `quarterly` can be declared:

```
long quarterly[4];
```

For the variable-length array `stats`, the situation is a little more complicated. Because POSTGRES will not know in advance how big the array is, POSTGRES will expect the length of the array (in bytes) to be encoded in the first four bytes of the memory which contains the array. The expected structure is:

```
typedef struct {
    long length;
    unsigned char bytes[1]; /* Force contiguity */
} VAR_LEN_ATTR;
```

The input function for the `stats` array will look something like:

```
VAR_LEN_ATTR *
stats_in(s)
    char s;
{
    VAR_LEN_ATTR *stats;
    long array_size, *arrayp, nbytes;

    /*
     * nbytes is the total number of bytes in stats,
     * INCLUDING the byte count at the beginning
     */
    nbytes = array_size * sizeof(long)  +  sizeof(long);

    stats = (VAR_LEN_ATTR *) palloc(nbytes);

    stats->length = nbytes;

    arrayp = &(stats->bytes[0]);

    /*
     * put code here that loads interesting stuff into
     * arrayp[0] .. arrayp[array_size]
```

We assume `sizeof(long)` == 4.

```
     */

    return(stats);
}
```

The output function for `stats` will get the same `VAR_LEN_ATTR` structure.

Now, assuming the functions are in `/usr/local/postgres/stats.c` and `/usr/local/postgres/quarterly.c`, we can define our two arrays. First we will define the fixed-size array `quarterly`.[9]

```
* define function quarterly_in
      (language = "c", returntype = quarterly)
      arg is (char16)
      as "/usr/local/postgres/quarterly.o" \g

* define function quarterly_out
      (language = "c", returntype = char16)
      arg is (quarterly)
      as "/usr/local/postgres/quarterly.o" \g

* define type quarterly
      (element = int4, internallength = 16,
       input = quarterly_in, output = quarterly_out) \g
```

Now we define the `stats` array:

```
* define function stats_in
      (language = "c", returntype = stats)
      arg is (char16)
      as "/usr/local/postgres/stats.o" \g

* define function stats_out
      (language = "c", returntype = char16)
      arg is (stats)
      as "/usr/local/postgres/stats.o" \g

* define type stats
      (element = int4, internallength = variable,
       input = stats_in, output = stats_out) \g
```

Now we can run some queries:

```
* create test (a = quarterly, b = stats) \g

* append test (a = "1 2 3 4"::quarterly,
               b = "5 6 7"::stats) \g
```

`internallength` == 16 follows from our assumption about `sizeof(long)`.

```
* append test (a = "1 3 2 4"::quarterly,
               b = "6 4"::stats) \g

* append test (a = "7 11 6 9"::quarterly,
               b = "1 2"::stats) \g

* retrieve (test.all) where test.a[4] = test.b[2] \g
```

which returns:

a	b
1324	64

NOTE that when you use your own functions to input and output array types, **your function** will define how to parse the external (string) representation. The braces notation is only a convention used by `array_in` and `array_out` and is **not** part of the formal POSTQUEL definition.

8.6. Large Object types

The types discussed to this point are all **small** objects—that is, they are smaller than 8 Kbytes[10] 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, which you should have available to consult as you read the following.

8.6.1. Defining a large object

Just like any other type, a large object type requires input and output functions. For the purposes of this discussion, we assume that two functions, `large_in` and `large_out` have been written using the large object interface, and that the compiled functions are in `/usr/local/postgres/large.o`. We also presume that we are using the "file as an ADT" interface for large objects discussed in the Reference Manual.

We define a large object which could be used for storing map data:

```
* define function large_in
      (language = "c", returntype = map)
      arg is (char16)
      as "/usr/local/postgres/large.o" \g

* define function large_out
      (language = "c", returntype = char16)
      arg is (map)
      as "/usr/local/postgres/large.o" \g

* define type map
```

[10] 8 * 1,024 == 8,192 bytes

```
                (internallength = variable,
                 input = large_in, output = large_out) \g
```

Note that large objects are **always** variable-length.

Now we can use our `map` object:

```
* create maps (name = text, a = map) \g

* append maps (name = "earth",
               a = "/usr/local/postgres/earth") \g

* append maps (name = "moon",
               a = "/usr/local/postgres/moon") \g
```

Notice that the above queries are identical in syntax to those we have been using all along to define types and such; the fact that this type is a large object is completely hidden in the large object interface and POSTGRES storage manager.

8.6.2. Writing functions and operators for large object types

Like any other POSTGRES type, you can define functions and operators for large object types. The only caveat is that, like any other functions which process a large object, they **must** use the large object interface described in Section 7 of the POSTGRES Reference Manual. Possible queries which involve functions on large objects could include

```
* retrieve (emp.name) where beard(emp.picture) = "red" \g

* retrieve (mountain.name)
      where height(mountain.topomap) > 10000 \g
```

Because all functionality is available to large objects, **any** aspect of POSTGRES is available for use with them, including index access methods, if the appropriate operator classes have been defined. Operator classes for index access methods will be discussed later in this manual.

9. The POSTGRES Rule System

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. POSTGRES actually has two rule systems, the **instance-level** rule system and the **query rewrite** rule system.

9.1. The Instance-level Rule System

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 theoretical foundations of the POSTGRES rule system can be found in [STON90].

9.2. The Query Rewrite Rule System

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 and discussion can be found in the demo 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] and [STON90].

9.3. When to use either?

Since each rule system is architected 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.

10. Administering POSTGRES

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

10.1. User administration

The `createuser` and `destroyuser` 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 specifics on their use.

10.2. Moving database directories out of /usr/local/postgres/data/base

By default, all POSTGRES databases are stored in separate subdirectories under `/usr/local/postgres/data/base/`.[11] To move a particular data base to an alternate directory (e.g., on a filesystem with more free space), 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`.
- Copy the directory `/usr/local/postgres/data/base/foo` and it contents to its ultimate destination. It should still be owned by the `postgres` user.
- Remove the directory `/usr/local/postgres/data/base/foo`.
- Make a symbolic link in `/usr/local/postgres/data/base` to the new directory.

10.3. Troubleshooting POSTGRES

Occasionally, POSTGRES will fail with cryptic error messages that are due to relatively simple problems. The following are a list of POSTGRES error messages and the likely fix. These messages are ones you would likely see in the monitor program.

```
Message: semget: No space left on device
```

[11] Data for certain classes may stored elsewhere if a non-standard storage manager was specified when they were created.

```
Explanation and Likely Fix:
```

Either the kernel has not been configured for System V shared memory, or some other program is using it up. On most machines, the UNIX command `ipcs` will show shared memory and semaphore usage.

To delete all shared memory and semaphores (may be necessary if a backend fails), run the `ipcclean` command. Note, however, that `ipcclean` deletes **all** semaphores belonging to the user running it, so the user should be certain that none of his/her non-POSTGRES processes are using semaphores before running this command.

```
Message: Unable to get shared buffers

Explanation and Likely Fix:
```

This message means that a POSTGRES backend was expecting shared memory to be available and it was not. Usually this is due to `ipcclean` being run while a `postmaster` was also running.

```
Message: Can't connect to the backend (...)

Explanation and Likely Fix:
```

This message means that you are running a LIBPQ application but it could not link up with a `postmaster`. If you see this error message, you should see if a `postmaster` is truly running. If one is running, the problem is likely related to your network.

11. 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, Berkeley, ERL Memo M90/33, April 1990.

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

[SCHA90] Shapiro, L., "Join Processing in Database Systems with Large Main Memories," ACM-TODS, Sept. 1986.

[STON86] (missing)

[STON87] Stonebraker, M., "The POSTGRES Storage System," Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[STON88] (missing)

[STON90] 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.

[STON90B] (missing)

[WANG88] (missing)

APPENDIX: User defined types and indices

In this section, we will discuss how to extend POSTGRES to use a user-defined type and associated functions with existing access methods. This way, you can define a BTREE or RTREE index on your own type. To do this, we will discuss how to define a new operator class in POSTGRES for use with an existing access method.

Our example will be to add a new operator class to the BTREE access method. The new operator class will sort integers in ascending absolute value order. This tutorial will describe how to define the operator class. If you work the example, you will be able to define and use indices that sort integer keys by absolute value.

There are several POSTGRES system classes that are important in understanding how the access methods work. These will be discussed, and then a sample procedure for adding a new set of operators to an existing access method will be shown as an example.

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)
`am*`	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. For BTREES, this object ID is 403. 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, BTREES impose a strict ordering on keys, less to greater. Since POSTGRES allows the user to define operators, POSTGRES cannot in general look at the name of an operator (eg, >, <) and tell what kind of comparison it is. In fact, some access methods (like rtrees) don't impose a less-to-greater ordering, but some other ordering, like containment. POSTGRES needs some consistent way of taking a scan qualification, looking at the operator, deciding if a usable index exists, and rewriting the query qualification in order to improve access speeds. This implies that POSTGRES needs to know, for example, that <= and > partition a BTREE. Strategies is the way that we do this.

Defining a new set of strategies is beyond the scope of this discussion, but how the BTREE strategies work will be explained, since 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 BTREES, 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 add procedures corresponding to the comparisons above to the pg_amop relation (see below). The access method code can use these numbers, regardless of data type, to figure out how to partition the BTREE, compute selectivity, and so on. Don't worry about the details of adding procedures yet; just understand that there's a set of these for int2, int4, oid, and every other data type on which a BTREE can operate.

Strategies are used by all of the POSTGRES access methods. Some access methods require other support routines in order to work. For example, the BTREE 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 RTREE 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 BTREES, 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.

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 BTREE operator class has a set of procedures, one through five, above. Some existing opclasses are int2_ops, andint4_ops, oid_ops. You need to add an instance with your opclass name (for example, 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: The oid for your pg_opclass instance **may 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 BTREES, the operators we require are:

absolute value less-than
absolute value less-than-or-equal
absolute value equal
absolute value greater-than-or-equal
absolute value greater-than

Suppose the code that implements the functions defined is stored in the file /usr/local/postgres/int4_abs.c. The code is

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

#include "tmp/c.h"

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

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

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

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

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

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

There are a couple of important things that are happening below. First, note that operators for less, less equal, equal, greater equal, and greater 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 **doesn't** 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 functions return **boolean** values; the access methods rely on this fact.

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

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

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

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

* define function int4_abs_gt
      (language = "c", returntype = bool)
      arg is (int4, int4)
      as "/usr/local/postgres/int4_abs.o" \g
```

Now define the operators that use them. As noted, the operator names must be unique for two `int4` operands. You can do a query on `pg_operator`:

```
* retrieve (pg_operator.all) \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 die 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 BTREE (== 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 btree, 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	>>&

(Note that your oid numbers may 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. (Of course, there are lots of other queries we could used to get the oids we wanted.)

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 BTREE 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” is 1, “less equal” is 2, “equal” is 3, “greater equal” is 4, and “greater” is 5.

Okay, now it’s time to test the new opclass. First we’ll create and populate a class:

```
* 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.

```
* 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.