The POSTGRES User Manual

Edited by Jon Rhein and Greg Kemnitz
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 3.0 of POSTGRES. POSTGRES Software is available for SUN 3 and SUN 4 class machines, for DECstation 3100 and 5000 machines, and for the SEQUENT Symmetry machine.

1.2. PERFORMANCE

Version 3.0 has been tuned modestly. Hence, 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 Version 3 contributors (in alphabetical order) consisted of James Bell, Jennifer Caetta, Ron Choi, Adam Glass, Jeffrey Goh, Wei Hong, Anant Jhingran, Greg Kemnitz, Jeff Meredith, Michael Olson, Lay-Peng Ong, Spyros Potamianos, and Cimarron Taylor.

Greg Kemnitz served as chief programmer and was responsible for overall coordination of the project and for individually implementing the "everything else" portion of the system.

The above implementation team contributed significantly to this manual, as did Claire Mosher.

2. ABSTRACT

The POSTGRES project undertook to build a next generation DBMS whose purpose was to rectify the known deficiencies in current relational DBMSs. This system, constructed over a four year period by one full time programmer and 3-4 part time students, 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.

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.

3. 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 insufficient for future data processing applications.

POSTGRES tried to build a data model with substantial additional power, yet requiring the understanding of as few concepts as possible. The relational model succeeded in replacing previous models in part because of its simplicity. We wanted to have as few concepts as possible so that users would have minimum complexity to contend with. Hence, POSTGRES leverages the following four 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]. We are now delivering version 3, which is the subject of this manual.

4. 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.

5. WHAT SHOULD BE 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 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 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.

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, **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. First, you can run the POSTGRES terminal monitor which allows you to interactively enter, edit, and execute commands in the query language POSTQUEL. Second, 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. The third way of interacting with POSTGRES is to use the facility called **fast path,** which allows you to directly execute functions stored in the database. This facility is described in the Reference under "Fast Path". Lastly, POSTGRES is accessible from the PICASSO programming environment. PICASSO is a graphical user interface (GUI) toolkit that allows a user to build sophisticated DBMS-oriented applications. PICASSO is described in a collection of reports [WANG88, SCHA90] and is not treated further in this manual.

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

% monitor <dbname>

As a result, you will be greeted by the following message:

```
Welcome to the 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.

* GoodBye

%
```

For a complete collection of monitor commands, see the manual page on "monitor" in the Unix section of the Reference.

If you are the database administrator for a database, 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.

6.2. Classes and the Query Language POSTQUEL

6.2.1. Basic Capabilities

In order to begin using POSTGRES, create the **foo** database as described in the previous section, and start the terminal monitor. 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 a unique (never-changing) identifier (OID).

6.2.2. Creating a New Class

A user can create a new class by specifying the class name, along with all attribute names and their types, for example:

```
* create EMP (name=char[12], salary=float8, age=int4, dept=char[12])

* create DEPT (dname=c[12], floor=int4)

* \g
```

So far, the create command looks exactly like the create statement in a 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 as follows:

```
* append EMP (name="Joe",salary=1400.0,age=40,dept="shoe")

* append EMP (name="sam",salary=1200.0,age=29,dept="toy")

* append EMP (name="Bill",salary=1600.0,age=36,dept="candy")

* \g
```

This will add 3 instances to EMP, one for each 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 one would type:

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

* \g
```

Notice that parentheses are required around the target list of returned attributes. Like Quel, POSTQUEL allows you to return computations in the target list as long as they are given a name, e.g:

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

* \g
```

6.2.5. Redirecting retrieve queries

Moreover, like Quel, any retrieve query can be redirected to a new class in the database and arbitrary boolean operators (and, or, not) are allowed in any query:

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

* \g
```

6.2.6. Joins

Joins are done in POSTQUEL in essentially the same way as in QUEL. 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 range over all instances of the class. A POSTQUEL query can contain an arbitrary number of class names and surrogates. The semantics of such a join are identical to those of QUEL, namely 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.

6.2.7. Updates

Updates are accomplished in POSTQUEL using the replacement statement, e.g:

```
* 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.

6.2.8. Deletions

Lastly, deletions are done using the delete command, as follows:

```
* 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 fully supports both fixed-length and variable-length arrays, and here we illustrate their use. First, we create a class with an array type.

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

The above query will create a class named SAL_EMP with a variable length field **name** and a fixed-length field **pay_by_quarter,** which represents the employee's salary by quarter. Now we do some appends; note that when appending to a non-character array, we enclose the values within {curly brackets} and separate them by commas.

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

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

* 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:

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

* \g
```

This query retrieves the names of the employees whose pay changed in the second quarter.

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

* \g
```

This query retrieves the third quarter pay of all employees.

```
* delete SAL_EMP where SAL_EMP.name[1] = ’j’

* \g
```

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

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

```
* create manager (name = char[], employees = text[])

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

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

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

* \g
```

This creates a class "manager", and provides a list of employees. Note that the **text** type in POSTGRES is defined as an array of character, so queries like

```
* retrieve (manager.name) where manager.employees[1][1] = 'b'

* retrieve (manager.name) where manager.employees[3] = "jeff"

* retrieve (manager.employees[3][2])

* delete manager where manager.employees[2][1] = 'g'

* \g
```

will all work.

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

Starting with the **foo** class from the previous section, reappend the three persons who were just deleted.

Now create a second class STUD_EMP, as follows:

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

* \g
```

In this case, an instance of STUD_EMP inherits all data fields from its parent, EMP, to wit: name, salary, age, and dept. Furthermore, student employees have an extra field, location, that shows their address as a (longitude, latitude) pair. In POSTGRES a class can inherit from zero or more other classes, and the inheritance hierarchy is thereby a directed graph in general. Additionally, in POSTQUEL a query can either reference 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 40:

```
* retrieve (E.name) from E in EMP where E.age > 40

* \g
```

On the other hand, if one wanted the names of all student employees and employees over 40, the notation is:

```
* retrieve (E.name) from E in EMP* where E.age > 40
```

Here the * after EMP indicates that the query should be run over EMP and all classes below EMP in the inheritance hierarchy. This use of * allows a user to easily run queries over a class and all its descendent classes.

Notice that location in STUD_EMP is not a traditional relational data type. A POSTGRES installation can be customized with an arbitrary number of user-defined data types; this will be discussed later.

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 the salary of Sam at time T one would query:

```
* retrieve (EMP.salary) using EMP[T] where EMP.name="Sam"

* \g
```

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

7. User Extensions to POSTQUEL

Here, we will discuss user extensions to the POSTQUEL query language, via 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 and functions provided by the user via a programming language. 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, which is assumed to return a collection of instances. For example, the following function defines the high-paid employees:

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

* \g
```

POSTQUEL functions can also have parameters, for example:

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

* \g
```

POSTQUEL functions are useful for creating composite types, as described below.

7.2. Composite Types

Moreover, since POSTQUEL functions return 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:

```
* addattr EMP (manager=EMP)

* \g
```

Here, we have added an attribute to the EMP class which is of type EMP, i.e. it has a value which is zero or more instances of the class EMP. Specifically, the value of the manager field is intended to be an instance of EMP which is the manager of the indicated employee. Since the value of manager has a record-oriented structure, we call it a composite object. We will now illustrate assigning values to instances of manager. First, we will define the function mgr_lookup:

```
* define function mgr_lookup
     (language = "postquel", returntype = EMP)
     arg is (char16) as
     retrieve (EMP.all) where EMP.name=DEPT.manager and DEPT.name=$1

* \g
```

This function can be used to assign values to the manager attribute in the EMP class, for example:

```
* append to EMP
  (name="Sam",salary=1000.0 ,age=40,dept="shoe",
   manager=mgr_lookup("shoe"))

* \g
```

Since EMP.manager is a composite object, POSTQUEL allows referencing into it with a second use of the dot notation. Whenever a composite object appears in a class, a user can utilize the cascaded dot notation to reference into the object.

In this case, the same POSTQUEL function is used to define the value of manager for every EMP instance. As a result, there is a second more efficient way to utilize POSTQUEL functions to assign values to the manager attribute. Specifically, we will define a second POSTQUEL function, lookup_mgr, as follows:

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

* \g
```

In this case, the function lookup_mgr has an argument which is an instance of the class EMP. Therefore, it takes a value for each instance of EMP, which is the result of the query with the field "EMP.dept" filled in with its appropriate constant.

Consequently, the user can think of the function lookup_mgr as an attribute of EMP and can reference it just like any other attribute (except for direct appends - that is, append emp (emp.manager.name = "Smith") won't work).

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

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

* \g
```

The same query is also available in functional notation:

```
* retrieve (EMP.name) where lookup_mgr(EMP).name="Joe"

* \g
```

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 input and output user types, used as operators for qualifying data, and can be used to define ordering for use in defining 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:

o 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.

o 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 way to define a circle is:

```
(center_x, center_y, radius)
```

in whatever Cartesian units we desire. Now we essentially have defined what the input to our circle input function looks like, and what the output from the circle output function should look like. Now we have to come up with an internal representation for a circle in memory. Note that as this memory will be written to disk and read from disk, our internal representation must be both **COMPLETE** and **CONTIGUOUS,** that is, we can have no pointers within our structure. The following declaration for a circle is legal and reasonable given the format we chose above:

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

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

Now, just for the sake of argument, we will define what may seem at first glance to be a correct type and explain why it **will not work.** Suppose we define our circle type thus:

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

where POINT is as in the first example. It will **not** work because **center** is a volatile pointer and will not have its value written to disk. **Beware** of such constructions when defining types, since they may appear to work until you shut down the data manager, and because there is no way for POSTGRES to let you know that there is a problem until garbage is written to disk and passed to your output function.

8.3.2. Defining the input and output functions for our type

Suppose in defining our type "circle," we have a file called "circle.c" and it is in /usr/postgres/tutorial/circle.c. It has been compiled, and the compiled object is in /usr/postgres/tutorial/circle.o. Note that as mentioned above, all functions related to our "circle" type have to be in one object (".o") file. The source code for circle.c is at the end of this section.

For the sake of argument, suppose we are on a DECstation, where sizeof(double) is 8 bytes (this will be important later).

In "circle.c," there are three functions:

circle_in, which is the input function for circles. It takes a string as input and outputs a pointer to type CIRCLE. It must know how to parse the string itself.

circle_out, which is the output function for circles. It is passed a pointer to type CIRCLE as input and returns a string. This string must be suitable for input to circle_in. That is, the following piece of code should work:

```
CIRCLE *circle, *circle_in();
char *circle_out();

circle=circle_in(circle_out(circle_in("(2.0,3.0,5.0)")));
```

equal_area_circle, which is the equality function for circles. For the purposes of this discussion, circles are equal if their areas are equal.

Now that we have written these functions and compiled them, we have to let POSTGRES know that they exist. We will do this first by running the following queries to define the input and output functions:

```
* define C function circle_in
      (file="/usr/postgres/tutorial/circle.o", returntype=circle)
      arg is (char16)

* \g

*  define C function circle_out
      (file="/usr/postgres/tutorial/circle.o", returntype=char16)
      arg is (circle)

* \g
```

Now that we have done this, we can define the type. Note that in and out functions have to be defined BEFORE we define the type. Defining the type first will not work. POSTGRES will notify you that return type circle is not defined yet, but everything is OK. The query to define the "circle" type is:

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

```
* \g
```

where **internallength** is the size of the circle structure itself in bytes. For circles, the arguments are three doubles, thus

```
sizeof(circle) = 3 * sizeof(double) = 3 * 8 = 24
```

When defining your own types, you should write a test program that does a

```
printf("size is %d0, sizeof (mystruct));
```

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 redefine the circle type. Note that we would **not** have to redefine our functions - POSTGRES already knows about them.

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 with the following query:

```
* define C function equal_area_circle
      (file="/usr/postgres/tutorial/circle.o", returntype=bool)
      arg is (circle,circle)

* \g
```

Now that we have done this, we will now bind this function to the equality symbol with the following query:

```
* define operator =
      (arg1=circle,arg2=circle,procedure=equal_area_circle)

* \g
```

8.3.4. Using our type

Now we are ready to create a class and run some queries on our new type:

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

* append tutorial(a="(1.0,1.0,10.0)"::circle)
* append tutorial(a="(2.0,2.0,5.0)"::circle)
* 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
```

And the monitor will display the result of the retrieve command...

```
----------------
|     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.

8.3.5. C Source code for our type

The following is the source code for circle.c.

```
#include <stdio.h>
#include "utils/geo-decls.h"

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

CIRCLE    *circle_in();
char    *circle_out();
int    pt_in_circle();

#define LDELIM    '('
```

```
#define RDELIM    ')'
#define NARGS    3

CIRCLE *
circle_in(str)
char    *str;

{
    double  atof(), tmp;
    char    *strcpy(), *p,
            *coord[NARGS], buf2[1000];
    int     i, fd;
    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
pt_in_circle(point, circle)
    POINT    *point;
    CIRCLE    *circle;
```

```
{
    extern double    point_dt();

    return (point_dt(point, &circle->center) < circle->radius );
}
```

8.4. Additional info on creating a user-defined function

8.4.1. Use palloc and not malloc

In order for POSTGRES to free 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 used in identical manner as the Unix "free" library call:

```
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/postgres/tutorial/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. Rather than getting the instance itself in your function's parameter list, you will use POSTGRES library routines to access the data in the instance as described below.

Suppose we want to write a function to answer the query

```
* retrieve (emp.all) where overpaid(emp)
```

In the query above, a reasonable "overpaid" function would be:

```
bool
overpaid() /* note that there are no arguments */

{

    extern Datum GetAttribute();
    short age, salary, performance;

    salary=(short) GetAttribute("salary");
    seniority=(short) GetAttribute("seniority");
    performance=(short) GetAttribute("performance");

    return (salary > (seniority * performance));

}
```

GetAttribute is the POSTGRES system function that returns attributes out of the current instance. It has one argument which is the name of the desired attribute, and its return type is a type "Datum" which is defined as being large enough to hold pointer values and all other types--currently it is defined as a "long." **GetAttribute** will align data properly so you can cast its return value into the desired form. So if you have an attribute "name" which is of the POSTQUEL type "char16," the **GetAttribute** call would look like:

```
char *str

str = (char *) GetAttribute("name")
```

Note that the instance argument must be the first argument when using your function in POSTQUEL queries.

To let POSTGRES know about the "overpaid" function, just do the following:

```
* define function overpaid
      (file="/usr/postgres/tutorial/overpaid.o", language = "c",
       returntype=bool)
  arg is (SET)
```

```
* \g
```

The special flag "set" in the argument list tells POSTGRES that this function will be processing an instance.

Your function may only access one instance via the **GetAttribute** call, but you can have additional base or user-defined types as arguments. Thus,

```
* retrieve (emp.all) where overpaid2(emp, "bill", 8)
```

could be written, and overpaid2 would be declared thus

```
bool
overpaid2(name, number)

char *name;
long number;
```

Note that the arguments are "shifted" one to the left, and the "emp" argument is omitted; its values are referenced via **GetAttribute.**

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 thus:

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

* \g
```

and do some appends

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

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

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

* \g
```

We can now run queries like the following:

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

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

* \g
```

NOTE: Recall how we defined circle equality above.

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 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
      (input = array_in, output = array_out, internallength = variable,
      element = int4)

* \g
```

The **element** parameter indicates that this is an array, and setting **internallength** to **variable** indicates that it is a variable length attribute. **NOTE** that any type using **array_in** and **array_out** *must* be variable length.

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

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

Now we use our new type:

```
* create stuff (a = int_arrays)

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

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

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

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

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

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

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

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 discuss how to go about doing this.

For the purpose of this section, we will create an array of four longs called **quarterly,** and a variable length array of longs called **stats.** Also, assume we are on a machine where sizeof(long) is 4.

The only special things we need to know when writing the in/out functions for **quarterly** is that POSTGRES will pass a "simple" array of longs to the output function and expect a "simple" array of longs in return from the input function. A sample "simple" array suitable for quarterly can be declared as follows:

```
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 as follows:

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

The in function for the stats array will look something like this:

```
VAR_LEN_ATTR *stats_in(s)

char s;

{
   VAR_LEN_ATTR *stats;
   long array_size, *arrayp;

   ... figure out how long the stats array should be ...

   /* We need the extra sizeof(long) to hold the length */

   stats = (VAR_LEN_ATTR *) palloc(array_size * sizeof(long) + sizeof(long));

   /* length is length of the entire structure in bytes, INCLUDING ITSELF */

   stats->length = array_size * sizeof(long) + sizeof(long);

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

   for (i = 0; i < length; i++, arrayp++)
   {
      ... populate the stats array with numbers from s ....
   }

   return(stats);
```

```
}
```

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

Now, using the **define function** command, and assuming the functions are in /usr/postgres/tutorial/stats.c and /usr/postgres/tutorial/quarterly.c, we can define our two arrays. First we will define the fixed-size array "quarterly". Note that internallength is known to be 16, because

```
sizeof(long) * 4 = 4 * 4 = 16
```

on our hypothetical machine.

First we define the in/out functions:

```
* define function quarterly_in
      (file = "/usr/postgres/tutorial/quarterly.o", language = "c",
       returntype = quarterly)
      arg is (char16)

* define function quarterly_out
      (file = "/usr/postgres/tutorial/quarterly.o",language = "c",
       returntype = char16)
      arg is (quarterly)

* \g
```

And now we can define the type:

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

* \g
```

Now we define the stats array. First we define its in/out functions:

```
* define function stats_in
      (file = "/usr/postgres/tutorial/stats.o", language = "c",
       returntype = stats)
      arg is (char16)

* define function stats_out
      (file = "/usr/postgres/tutorial/stats.o", language = "c",
       returntype = char16)
      arg is (stats)

* \g
```

and now we define the type. Note that setting internallength to variable tells POSTGRES that it is a variable length type.

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

* \g
```

Now we can run some queries:

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

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

* retrieve (test.all) where test.a[3] = test.b[1]

* \g
```

```
------------------
|    a    | b |
-------------------
| 1 3 2 4 | 6 4 |
------------------
```

NOTE that when you use your own functions to input and output array types, *your function* will define how to parse the string. The brackets notation is only a convention used by **array_in** and **array_out** and is *not* a 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 8K 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 basic model of this interface is the Unix file system interface; the particulars of this interface are detailed in Section 7 of the POSTGRES Reference.

8.6.1. What can you do with large objects?

The answer to this question is anything that can be done with small objects, as long as all functions which access a large object use the POSTGRES large object interface. All discussion after this point will presume that you have read this section of the POSTGRES Reference.

8.6.2. 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 these functions are in /usr/postgres/tutorial/large.c. We also presume that we are using the "file as an ADT" interface for large objects discussed in the Reference. Now, we define a large object which could be used for storing map data:

```
* define function large_in
```

```
      (file = "/usr/postgres/tutorial/large.o", language = "c",
       returntype = map)
      arg is (char16)

* define function large_out
      (file = "/usr/postgres/tutorial/large.o", language = "c",
       returntype = char16)
      arg is (map)

* define type map
      (input = large_in, output = large_out, internallength = variable)

* \g
```

NOTE: Large objects are **always** variable length. Now we can use our large object:

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

* append maps (name = "earth", a = "/usr/postgres/maps/earth")
* append maps (name = "moon", a = "/usr/postgres/maps/moon")

* \g
```

Notice that the above queries are identical to the syntax 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.

8.6.3. 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. Possible queries which involve functions on large objects could include

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

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

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

8.7. 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.

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.

pg_am 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)
>
> 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 400. You don't need to add a new instance to this class; all you're interested in is the object ID of the BTREE instance.

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

* \g

---------------
| oid         |
---------------
| 400         |
---------------
```

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 sort 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's, int4's, oid's, and every other data type on which a BTREE can operate.

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.

The next class of interest is PG_OPCLASS. This class exists only to associate a name with an oid. In pg_amop, every 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, "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 you have an access method and an operator class. You need some operators; use the method for defining operators discussed previously. Now, suppose the c code that implements the functions defined is stored in the file

/usr/postgres/tutorial/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 *do* 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.

so here goes -- let's define some functions.

```
* define function int4_abs_lt
       (file = "/usr/postgres/tutorial/int4_abs.o", language = "c",
        returntype = bool)
       arg is (int4, int4)

* \g

* define function int4_abs_le
       (file = "/usr/postgres/tutorial/int4_abs.o", language = "c",
        returntype = bool)
       arg is (int4, int4)

* \g

* define function int4_abs_eq
       (file = "/usr/postgres/tutorial/int4_abs.o", language = "c",
        returntype = bool)
       arg is (int4, int4)

* \g

* define function int4_abs_ge
       (file = "/usr/postgres/tutorial/int4_abs.o", language = "c",
        returntype = bool)
       arg is (int4, int4)
```

```
* \g

* define function int4_abs_gt
        (file = "/usr/postgres/tutorial/int4_abs.o", language = "c",
         returntype = bool)
        arg is (int4, int4)

* \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 is the C function 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, equal, and greater 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 (== 400, see above);

amopclaid:  the oid of the pg_opclass instance for int4_abs_ops
            (== whatever you got instead of 17314, see above);

amopopr:  the oids of the operators for the opclass (which we'll
          get in just a minute); and

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 oids of the operators we just defined. We'll look up the names of all the operators that take two int4's, and pick ours out. (Note that your numbers may be different.)

```
* 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

-----------------------------
| oid       | oprname    |
-----------------------------
| 96        | =          |
-----------------------------
| 97        | <          |
-----------------------------
| 514       | *          |
-----------------------------
| 518       | !=         |
-----------------------------
| 521       | >          |
-----------------------------
| 523       | <=         |
```

```
-----------------------------
| 525       | >=       |
-----------------------------
| 528       | /        |
-----------------------------
| 530       | %        |
-----------------------------
| 551       | +        |
-----------------------------
| 555       | -        |
-----------------------------  _
| 17321     | <<&      |   \
-----------------------------    \
| 17322     | <=&      |     \
-----------------------------      \
| 17323     | ==&      |       | these are the ones we want
-----------------------------      /  -- the values you get for oids
| 17324     | >=&      |     /   will probably be different, and
-----------------------------    /    you should substitute them for
| 17325     | >>&      |   /     the values i use 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 400 and int4_abs_ops is oid 17314. Then we add the instances we need:

```
* append pg_amop (amopid = "400"::oid,      /* btree oid */
             amopclaid = "17314"::oid, /* pg_opclass tup */
             amopopr = "17321"::oid,   /* <<& tup oid */
             amopstrategy = "1"::int2, /* 1 is <<& */
             amopselect = "btreesel"::regproc,
             amopnpages = "btreenpage"::regproc)

* \g

* append pg_amop (amopid = "400"::oid,
             amopclaid = "17314"::oid,
             amopopr = "17322"::oid,
             amopstrategy = "2"::int2,
             amopselect = "btreesel"::regproc,
             amopnpages = "btreenpage"::regproc)

* \g

* append pg_amop (amopid = "400"::oid,
             amopclaid = "17314"::oid,
             amopopr = "17323"::oid,
             amopstrategy = "3"::int2,
```

```
            amopselect = "btreesel"::regproc,
            amopnpages = "btreenpage"::regproc)

* \g

* append pg_amop (amopid = "400"::oid,
            amopclaid = "17314"::oid,
            amopopr = "17324"::oid,
            amopstrategy = "4"::int2,
            amopselect = "btreesel"::regproc,
            amopnpages = "btreenpage"::regproc)

* \g

* append pg_amop (amopid = "400"::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

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 ˜postgres/demo, which is included with the POSTGRES distribution. Additional discussion of the Instance-level Rule System can be found in the Reference in "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 ˜postgres/video, and further discussion is in the Reference 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** commands create and destroy POSTGRES users. Please read the "man pages" on these commands in the Reference for specifics on their use.

10.2. Moving database directories out of ˜postgres/data/base

If you do not want all users to have databases in ˜postgres/data/base/<db>, you can put their data directories in arbitrary places by using the following mechanism:

o Create a database using the **createdb** command.

o Move the directory ˜postgres/data/base/<newdb> to its ultimate destination. It should still be owned by the "postgres" user.

o Make a symbolic link from ˜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

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 the "ipcclean" command deletes all semaphores belonging to the user running it, so the user should be certain that no 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. **NOTE:** "ipcclean" will delete ALL semaphores and shared memory *whether they are being used or not.*

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.