THE POSTGRES TUTORIAL

Greg Kemnitz and Michael Stonebraker
EECS Department
University of California, Berkeley

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 180,000 lines of C. POSTGRES is available free of charge and is being used by perhaps 100 sites around the world. This tutorial describes the major concepts of the system and attempts to provide an accessable 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.

1. 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 data 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 data 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

This research was sponsored by the Defense Advanced Research Projects Agency through NASA Grant NAG 2-530 and by the Army Research Office through Grant DAALO3-87-K-0083.

In Section 2 we indicate how to construct a POSTGRES data base and the ways of interacting with one. Then, in Section 3 we indicate the POSTGRES notion of classes and give examples of the query language, POSTQUEL, which demonstrate traditional relational query language capabilities such as restrictions and joins. We also explain how to use functions and operators that have been defined to POSTGRES. After completing this introductory tutorial, you may want to read through the advanced tutorial in Part II, and if you intend to handle the duties of a DBA, you should look at Part III. Part IV of the tutorial package is intended for developers and if you intend to browse the POSTGRES source code, you should have a look at that as well.

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 we 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 rules system documented in [STON90B]. We are now delivering Version 2.1, which is the subject of this tutorial. Further information on this system can be obtained from the reference manual [MOSH91] and the above mentioned papers.

2. CREATING A DATA BASE

Once POSTGRES has been installed at your site by following the directions in the release notes, you can create a data base, foo, using the following command:

```
%createdb foo
```

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

Once you have constructed a data base, 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" subroutine 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 in the "libpq" section of the reference manual and is treated further in Section XXX of this tutorial. The third way on interacting with POSTGRES is to use a facility called "fast path", which allows you to directly execute functions stored in the data base. This faciltiy is described in Section YYY of this tutorial. 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 descried in a collection of reports [WANG88, SCHA90] and is not treated further in this tutorial.

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

%monitor foo

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

Welcome to the C POSTGRES terminal monitor

Go
*

The "Go" indicates the terminal monitor is listening to you and 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

*

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

*

If you make a typing mistake, you can move to the vi text editor by typing:

*\

The workspace will be passed to the editor, and you have the full power of vi to make any necessary changes. Whem you are ready to return to POSTGRES, simply type:

wq

and you will be returned to the monitor. To quit the monitor and return to UNIX, simply type:

*q

and the monitor will respond:

*I live to serve you.
*GoodBye
%

For a complete collection of monitor commands, consult the "monitor" section of the reference manual.

If you are the data base administrator for a data base, you can destroy it using the following UNIX command:

%destroydb foo

3. CLASSES and the Query Language POSTQUEL

3.1. Basic Capabilities

In order to begin using POSTGRES, create the foo data base as described in the previous section and then enter 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.** Moreover, each instance has a unique (never-changing) identifier (OID).

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 = c12, salary = float8, age = int4, dept = c12)
*create DEPT (dname = c12, floor = int4)
*
```

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.

To populate a class with instance, one can use the append command as follows:

```
*append EMP (name = "Joe", salary = 1400., age = 40, dept = "shoe")
*append EMP (name = "Sam", salary = 1200., age = 29, dept = "toy")
*append EMP (name = "Bill", salary = 1600., age = 36, dept = "candy")
*
```

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

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

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

Notice that parentheses are required around the target list of returned attributes. Like QUEL, POSTUEL 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"
*
```

Moreover, like QUEL, any retrieve query can be redirected to a new class in the data base 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
```

*

Joins are done in POSTQUEL in essentially the same way as 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
*
```

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.

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

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

This command replaces the salary of Joe by that of Sam. Lastly, deletions are done using the delete command, as follows:

```
delete EMP where EMP.salary > 0
```

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

3.2. Advanced POSTQUEL

In this section we will illustrate the POSTGRES notions of inheritance, user-defined operators, user defined functions, complex objects, and time travel. Starting with the foo class from the previous section, re-append the three persons who were just deleted.

Now create a second class STUD_EMP, as follows:

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

In this case, an instance of STUD_EMP inherits all data fields from its **parent,** EMP, namely name, salary, age, and dept. Moreover, student employees have an extra field, location, that indicates 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. Moreover, 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
*
```

On the other hand, if one wanted the names of all student employees or 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. In POSTGRES an installation can **customize** POSTGRES with an arbitrary number of **user-defined data types** as explained in Section ZZZ of this tutorial. In addition, POSTGRES can be customized with three kinds of user defined functions:

operators
C functions
POSTQUEL functions

and we illustrate their use in this section. Definition of these functions is deferred to Section 2.1.

Suppose a new operator for points has been defined, !^, which compares two points and returns true if the first point is "north of" the second point. In this case, the user can find all the student employees who live north of the point (5.0,5.0) as follows:

```
*retrieve (STUD_EMP.name) where STUD_EMP.location !^ "(5.0, 5.0)"
*
```

User defined operators such as !^ can be used in any POSTQUEL query wherever an operator is syntactically valid. There is no requirement that a constant be on one side of the operator. For example the following query finds all the student employees who live north of Joe.

```
*retrieve (S1.name) from S1 in STUD_EMP, S2 in STUD_EMP
*where where S1.location !^ S2.location and S2.name = "Joe"
*
```

Operators are automatically optimized by POSTGRES; hence if there is an efficient access path that allows solution to the above queries, then it will be used.

The second class of functions available in POSTGRES are functions coded in the porgramming language, C, which have been **registered** to POSTGRES. Suppose a function, distance, has been registered which accepts an argument of type point and returns a floating point number which represents the distance from the point to the origin of the co-ordinate system. Any user can then utilize distance in a query as illustrated in the following example:

retrieve (STUD_EMP.name) where distance (STUD_EMP.location) > 5.0

This query finds the student employees who live more than 5 miles from the origin.

C functions can also have an argument which is a class name, e.g:

*retrieve (EMP.name) where overpaid (EMP)
*

In this case overpaid has an operand of type EMP and returns a boolean. A functions whose argument is a class name is autmomatically inherited down the class hierarchy in the standard way. Hence, overpaid is automatically available for the STUD_EMP class. Therefore, the following query is also valid:

*retrieve (STUD_EMP.name) where overpaid (STUD_EMP)
*

In some circles such functions are called **methods.** Moreover, overpaid can either be considered as a function using the above syntax or as a new attribute for EMP whose type is the return type of the function. Using the latter interpretation, the user can restate the above query as:

retrieve (STUD_EMP.name) where STUD_EMP.overpaid

Hence, overpaid is interchangeably a function defined for each instance of EMP or a new attribute for EMP. The same interpretation of such functions appears in IRIS [FISH90].

C functions are arbitrary C procedures. Hence, they have arbitrary semantics and can run arbitrary POSTQUEL commands during execution. Therefore, queries with C functions in the qualification cannot be optimized by the POSTGRES query optimizer. For example, the above query on overpaid student employees will result in a sequential scan of all instances of the class.

The third kind of function available in POSTGRES is **POSTQUEL functions.** 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 returns EMP as
*retrieve (EMP.all) where EMP.salary > 50000
*

POSTQUEL functions can also have parameters, for example:

```
*define function large-pay (int4) returns EMP as
*retrieve (EMP.all) where EMP.salary > $1
*
```

Moreover, since POSTQUEL functions return sets of instances, they are the mechanism used to assign values to composite objects. For example, consider extending the EMP class with a manager field:

```
*add to EMP (manager = EMP)
*
```

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 (c12) returns EMP as
*retrieve (EMP.all) where EMP.name = DEPT.manager and DEPT.name = $1
*
```

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, age = 40, dept = "shoe", manager = mgr-lookup ("shoe"))
*
```

A user can query a composite object by using a **cascaded dot** notation. For example, to find the name of the manager of Joe, one could write:

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

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 (EMP) returns EMP as
*retrieve (E.all) from E in EMP
where E.name = DEPT.manager and DEPT.name = EMP.dept
*
```

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" filed 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.

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

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

The same query is also available using functional notation:

```
*retrieve (EMP.name) where lookup-mgr(EMP).name = "Joe"
*
```

Lastly, 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"
*
```

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

REFERENCES

[FISH90]

[MOSH91] Wensel, S. (ed.), ‘‘The POSTGRES Reference Manual, Version 2.1’’ Electronics Research Laboratory, University of California, Berkeley, CA, Report M91/10, February 1991.

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

[SCHA90] Schank, P. et. al., PICASSO Reference Manual," Electronics Research Laboratory, Memo UCB/ERL M90/79, Sept. 1990.

[STON86] Stonebraker, M. and Rowe, L., ‘‘The Design of POSTGRES,’’ Proc. 1986 ACM-SIGMOD Conference, Washington, D.C., June 1986.

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

[STON88] Stonebraker, M. et. al., "The POSTGRES Rules System," IEEE Transactions on Software Engineering, July 1988.

[STON90] Stonebraker, M. and Rowe, L., "The Implementation of POSTGRES," IEEE Transactions on Knowledge and Data Engineering, March 1990.

[STON90B] Stonebraker, M. et. al., ‘‘On Rules, Procedures Caching and Views,’’ Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., June 1990.

[WANG88] Wang, Y., "The PICASSO Shared Object Hierarchy," MS Report, University of California, Berkeley, June 1988.