Postgres Architecture and Historical Lore

1. Introduction

In the beginning, there was Ingres. And then, after ways to improve the relational model made themselves known, and as ways to implement object-oriented techniques of data organization without completely throwing out time-honored ways of doing things became necessary, Postgres came into being.

The original release of Postgres was partially written in Franz Lisp and partially written in C. However, this proved to be a rather difficult environment in which to work and debug, and was quite a problem for any potential users - it was painfully slow, and required expensive third-party licenses for the Lisp environment. Because of this, Postgres was ported to C, and the first C-only release went out in late 1988.

Subsequent releases of the Postgres software have stressed improvements and complete rewrites of various features including the Postgres rule system, the transaction system, the executor, access methods, as well as several ports to different machines.

In the following discussions, a general overview of the internal structure of Postgres will be presented, together with references to directories where these structures may be found. This is intended to be a rather quick "tour" of the Postgres code, stressing general principles and ideas rather than details such as function names. For that level of detail, the code itself is the best (and only) reference. It must be admitted here that precision is being avoided as much to keep this document from rapidly getting out of date as much as anything else.

2. General overview of Postgres at the communication level

The three main programs in the Postgres environment are the following:

User Applications
The "postmaster"
The Postgres backend

2.1. User Applications

A User Application is any program that uses LIBPQ to send and recieve data from a Postgres backend. A User Application can run on any host that can access the Postgres server from a TCP-IP network. User applications included in the Postgres distribution include the terminal monitor (monitor), the database creation program (createdb), and the database deletion program (destroydb).

2.2. The postmaster

The Postmaster handles most of the network initialization and connection activities. Once the Postmaster has handled the network stuff for a user application, it forks off a Postgres backend. Once this is done, the Postmaster is no longer involved, and goes back to waiting for connections from new user applications.

The first time the Postmaster is invoked, it allocates shared memory and semaphores used by Postgres backends for locking and buffer pools.

The postmaster and the backend have to run on the database server, and the database directories should be on local disks.

2.3. The Postgres backend

The Postgres backend is the true database engine. Once it has been forked by a postmaster, it is ready to recieve queries from and send back answers to the user application.

For every user application, there is one Postgres backend forked by the Postmaster.

3. Overview of the Postgres Main Modules

In this section, the general program flow in Postgres will be discussed, with a brief description of each program module, together with instructions on where to find the source code for that module. All modules discussed here are part of the Postgres backend, and are relatively tightly coupled at this time. The Postgres main modules include:

- The Parser
- The Query Rewrite Rule System

- o The Planner/Optimizer
- o The Executor
- o The Instance Level Rule System
- o The Access Methods
- o Housekeeping functions
- o Lock managers, cache utilities, and other miscellaneous functions.

In the execution of a Retrieve query, the following general flow is followed. Appends and other queries that access data (as opposed to "housekeeping" queries such as createdb, copy, etc) follow a similar flow.

```
              ----------------
          |  1.  Parser  |
          ----------------
                |
                | (Initial Query Parse Tree)
                |
                v
      +-------------------------------+
      | 2.  Query Rewrite Rule System |
      +-------------------------------+
                |
                | (Re-written Query Parse Tree)
                |
                |
                v
        +-----------------------+
        | 3.  Planner/Optimizer |
        +-----------------------+
                |
                | (Execution Plan Tree)
                |
                v
         +------------------+
         | 4.  Executor     |
         +------------------+
                |
                | (Open request)
                |
                |
                |        (get instances until done)
                v
          +-------------------+            +-----------------+
          | 5. Access Methods |<-----------| 4.  Executor    |
          +-------------------+            +-----------------+
                |                        ^
                | (instance)             |
                |                        | (instance)
                v                        |
          +-------------------------------+        |
          | 6. Instance Level Rule System |---------+
          +-------------------------------+
```

3.1. The Parser

The Parser parses a Postquel query and generates a parse-tree. As long as a correct parse-tree is generated, other parsers for other query languages (ie SQL) can be "dropped in" to the Postgres system. The Postquel query parser lives in:

˜/src/parser

3.2. The Query Rewrite rule system

The Query Rewrite rule system is essentially part of the parser. It generally works in the following manner: If there is a query-rewrite rule on this class of instances, add the rule to the initial user query through the use of boolean algebra. If not, just let the query go through.

The Query Rewrite rule system is used to implement not only obvious rules, but also versions, views, and postquel procedures. The Query Rewrite system lives in:

˜/src/rewrite

3.3. The Planner/Optimizer

The Planner/Optimizer takes a parse-tree and, using various cost functions and heuristics, generates an execution plan. The Planner/Optimizer also "drives" the executor in that it calls it once to initialize things and then calls it subsequently to fetch instances. The planner lives in:

˜/src/planner

3.4. The Executor

The first time the Executor is invoked by the Planner, it initializes itself, the access methods, and the instance-level rule system. In subsequent calls by the Planner, it walks the execution plan, fetching instances by calling the access methods and checking them against the query qualification (which is part of the execution plan) to see if they are part of the "answer" to the user query. Before doing this, however, it checks to see if a fetched instance triggers a instance-level rule. The Executor lives in:

˜/src/executor

3.5. The Access Methods

These are the low-level routines that hit the disk and handle any indices (ie btrees, rtrees) that the user may have defined. The Heap access method is used as the primary access method, and other access methods are defined on indices. The Executor calls these routines to fetch instances - the Access Methods then use a scanning mechanism defined by the Planner to get them and fetch them back to the executor. The Access Methods primarily live in:

˜/src/access/common (Code used by all access methods)
˜/src/access/heap (Heap access method - the lowest level)

˜/src/access/{index index-btree index-ftree index-rtree}
(Code to handle indices, and
Various different types of indices)

˜/src/access/transam (The Postgres Transaction System)

3.6. The Instance-level Rule System

The Instance-level rule system operates in the following manner: in the "system data" part of each instance, there is a field for instance level "rule locks". If a fetched instance has a rule lock, the associated rule(s) is executed. Each instance-level rule has an associated execution plan, so the executor will be ran from within the instance-level rule system. The modified instance, if any, is then handed back to the executor for further processing. The Instance-level Rule System lives in:

˜/src/rules/prs2 (Postgres Rule System 2 - the bulk of the rule system)
˜/src/rules/stubs

3.7. Housekeeping Functions

For queries that are not directly related to retrieving or appending, such as creating databases, defining new operators, rules, and registering user functions, the parser simply bypasses the planner and executor and directly calls utility routines to handle these special commands. The code for these routines is in:

˜/src/commands

3.8. Lock managers, cache utilities, and other miscellaneous functions

There are numerous other functions that are used in Postgres; these include functions to manage system attributes, lock managers, buffer and cache managers, hash table handlers, as well as the system built-in functions for pre-defined types. For lack of anything better to call these, these live in the UTIL module, and live in:

˜/src/utils/adt (built-in functions)
˜/src/utils/cache (cache handlers)
˜/src/utils/error (error handlers)
˜/src/utils/fmgr (the "Function Manager" - handles ADT's and user functions)
˜/src/utils/hash (hash table handlers)
˜/src/utils/init (initialization code)
˜/src/utils/mmgr (memory manager code)
˜/src/utils/sort (sort code)
˜/src/utils/time (time range qualification handlers)

3.9. 'Main' Programs

The "main programs" for the Postgres backend, the Postmaster, and other utilities like the vacuum daemon, etc live in

˜/src/support (everything but the backend)
˜/src/tcop (the backend main program - in postgres.c)

4. Postgres internal data structures

Since Postgres was originally written in Lisp, and for a time was a Lisp-C hybrid, many of its internal data structures are rather opaque to the C programmer. However, they are not overly difficult to understand and use once the basic ideas behind them are made clear. The discussion in this section will attempt to make these ideas clear, without dwelling excessively on details and trivia.

4.1. The CONS-cell abstraction

Internally, Postgres passes data about primarily in trees made of Lisp-like CONS-cell structures. These CONS-cells can point to interesting data only, to interesting data and another CONS-cell, or to two other CONS-cells. In particualar, parse-trees and execution plans are encoded in these CONS-cell structures. Utility functions for handling the Lisp-like structures are in

˜/src/lib/C

4.2. The Postgres Node System

A collection of data structures that is central to most Postgres operations is the Postgres "node system". This is a collection of data structures that encode various directives for the executor and planner, as well as numerous other "utility" purposes. Nodes are declared using a C++-like syntax that is used by a shell script to generate initialization and accessor functions. C preprocessor magic is used to ultimately turn this declaration into an ordinary C typedef.

4.2.1. An Example Node

An example of a Postgres node is the following:

```
class (Oper) public (Expr) {
/* private: */
        inherits(Expr);
        InstanceId      opno;
        InstanceId      opid;
        bool        oprelationlevel;
        InstanceId      opresulttype;
        int       opsize;
```

```
        FunctionCachePtr    op_fcache;
/* public: */
};
```

(this particular node is from ˜/src/lib/H/nodes/primnodes.h)

This particular node is used in execution plans to indicate that an operator must be executed. The "inherits" macro is used to inherit fields from the previously defined Expr (expression) node, and the explicit fields are fields that are unique to the Oper node itself. Declarations for all Postgres nodes are in header files in

˜/src/lib/H/nodes

As stated earlier, shell scripts read the node declarations and generate initializer and accessor functions for them. As long as the above declaration scheme is used, these scripts will work for new nodes as well. These scripts are in

˜/src/lib/Gen

4.2.2. Usage of the Node System

The different nodes are primarily used to give directives to the Executor in the execution plan. Nodes are used to indicate the scan type to be used, whether sorting is to be used or not, and are used to indicate how query qualifications are to be handled.

4.2.3. Printing Nodes and Node Structures

The best place to figure out what a structure using nodes and CONS-cells is like is to look carefully at the code in

˜/src/lib/C

The code in this directory prints node structures into strings and dumps them into strings. This code is primarily used by the Instance Level Rule System to save and restore execution plans that are associated with rules, which it stores on disk in a system class in string format. The function

LispDisplay(CONS-cell, 0)

displays the contents of a CONS-cell structure in a human readable but rather unfriendly format.

4.3. Other Important Data Structures

Aside from the CONS-cell and Node structures, there are very few other important global data structures, although there are numerous local structures that are important if one is modifying variosu functions. One exception is the HeapTuple data structure and its associated structures, since it is the type returned by virtually all the access methods, including those that handle system classes. The declaration for the HeapTuple type is in

˜/src/access/htup.h

5. The Postgres Function Manager

An integral part of Postgres is its ability to use user-defined functions and operators. The Postgres Function Manager is responsible for handling this aspect of Postgres.

5.1. The Function Manager

The Function Manager is used by all parts of Postgres that examine the internals of user data. It consists of several parts:

- o The Function Manager Interface
- o Data structures containing information about dynamically loaded files
- o The dynamic loader

5.1.1. The Function Manager Interface

The basic reason for the existance of the Function Manager is that due to the nature of user-defined functions, calls to them cannot be hardcoded anywere in the Postgres backend. A mechanism must exist which obtains addresses of user-defined and builtin functions (there is very little difference between these), handles the task of calling user-defined functions with appropriate arguments, and returning an appropriate value. The Function Manager Interface provides the abstraction from the details of calling user-defined functions from the rest of Postgres. The source to the Function Manager Interface is in

~/src/utils/fmgr/fmgr.c

5.1.2. Data Structures used for Function Execution

In the Function Manager, there are two in-memory data structures used for looking up function addresses. One is an array of builtin functions that is sorted by function OID. Another is a list of dynamically loaded files, which contains the addresses of each function in these files. Functions that populate and access these data structures are in

~/src/utils/fmgr/dfmgr.c
~/src/port/*/dynloader.c

and the data structures themselves are defined in

~/src/lib/H/utils/dynamic_loader.h
~/src/lib/H/utils/fmgr.h
~/src/lib/H/utils/builtins.h
~/src/lib/H/catalog/pg_type.h

and

~/src/utils/fmgr/fmgr.c

5.1.3. The Dynamic Loader

The Dynamic Loader loads a user-defined function from a file and determines the names and addresses of each function in that file. It then populates appropriate data structures with this information. The source to the Dynamic Loader is in

~/src/utils/fmgr/dfmgr.c
~/src/port/*/dynloader.c

5.2. General Algorithm

The Postgres Function Manager uses the following general algorithm for determining the addresses of functions. Once the address is found, the Function Manager Interface has functions which actually call the function.

Generalalgorithm

Is function in builtin list? If it is, return its address from this list

If the function is not in the list of builtins, it is a dynamically loaded function. If so, look for the function name in the list of dynamically loaded functions. If it is, return its address.

If the function has not yet been found, dynamically load it and determine its address. Return this address.

6. How does a user-defined function get executed?

The program flow discussed in Section 3 - with the exception of the rule systems - is essentially similar to that of any data manager. How, then, is Postgres so different? The main differences between Postgres and other data managers is in its concept of what data is. In Postgres, data is whatever the user says it is - the Postgres backend itself imposes no arbitrary definitions on data other than that it is a blob of memory of a known size.

The user defines what data is by defining functions and binding them to various operators. These operators are then bound to user-defined types, and thus the system is complete. In this section, how a user-defined function is executed will be discussed.

For the sake of this discussion (since operators will be discussed later), assume that Postgres is operating on the following query

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

and that overpaid(x) is a user-defined function taking an integer as an argument.

Initially, the Parser will walk this query, and generate a parse tree. While doing this, it will examine the following system classes:

- o PG_PROC - information about Postgres functions is stored here.
- o PG_TYPE - information about Postgres types is stored here.

PG_PROC contains information about the C file containing the function (if any), its return value, its number of arguments, its argument types, and other information. The Parser will also check PG_TYPE to make sure that emp.salary is an appropriate argument for overpaid(). Once everything has been found to be proper, appropriate nodes containing information about the function will be inserted into the parsetree.

After the Planner has turned the parsetree into an execution plan, the Executor will walk the plan, executing overpaid() on each instance fetched from the "emp" class.

The first time overpaid() is executed, the Executor will call the Postgres Function Manager to obtain the address of the function. (See discussion in Section 5 for details on the Function Manager). In subsequent calls, it will simply give Function Manager the address of the function so it can be called.

A user function like overpaid() is "registered" in Postgres using the Postquel

DEFINE C FUNCTION

query.

7. How does a User Defined Operator get executed?

When a user defines an operator, a user-defined function is associated with it. Therefore, execution of an operator is little different than execution of a function, with only one major exception: the Planner knows how to optimize operator queries but does not know how to deal with functions.

The major system class dealing with operators is

PG_OPERATOR

Additionally, the same classes discussed in Section 6 are used in the execution of operators. Binding of user functions to operators is handled with the Postquel

DEFINE OPERATOR

query.

8. How is a User Defined Type handled?

A User Defined Type minimally consists of two functions, the input function for the type, and the output function for the type. Additionally, ordering (less than, greater than) and equality functions are necessary if the type is to be used in qualified queries, or if an index is to be defined on the type.

In the system class

PG_TYPE

is information related to user-defined types, such as

- o The names of the registered functions for input and output of the type.
- o The size of the type

The Postquel query

DEFINE TYPE

populates the PG_TYPE class.