Concept Architecture of PostgreSQL

—by Hao Chen, Heechul Lim and Jin Xiao

Abstract

PostgreSQL is an open source, object-oriented relational database system. Up-to-date, thousands of database applications has been designed using PostgreSQL and it's wide acceptance justifies the validation of its conceptual architecture, via systematic analysis. This document attempts to sketch the conceptual architecture of PostgreSQL based on developer documentation and source code structure. The aforementioned conceptual architecture can aid in the validation process to be conducted at a later stage, providing a concrete architecture as a result. It is our hope that such a concrete architecture can serve as the PostgreSQL reference architecture for PostgreSQL developers.

1. Introduction

PostgreSQL is an open source, object-oriented relational database system. It was first developed in 1977, under the name "Ingres." In late 1990s, Postgres adopted SQL standard and took on the name, "PostgreSQL". Up-to-date, thousands of database applications has been designed using PostgreSQL and it's wide acceptance justifies the validation of its conceptual architecture, via systematic analysis. This document attempts to sketch the conceptual architecture of PostgreSQL based on developer documentation and source code structure. The aforementioned conceptual architecture can aid in the validation process to be conducted at a later stage, providing a concrete architecture as a result. It is our hope that such a concrete architecture can serve as the PostgreSQL reference architecture for PostgreSQL developers.

The paper first introduces the overall architecture of PostgreSQL as a classical client-server architecture. Each sub-system: server, client, and storage manager are examined in detail. To illustrate the applicability of our conceptual architecture, we trace work flow of

typical SQL queries in our architectural model. Lastly, we took a critical look at the advantageous and disadvantageous of PostgreSQL and speculate on its evolvability.

PostgreSQL uses a mixture of architectural styles. At the top level, the client and server interacts in the classical client-server model, while the data access structure is strictly layered. At the server layer, the query processing is structured as a pipeline, while the database access by server sub-systems is structured as a bulletin board. The interaction between the client and the server is largely request/reply driven and each client is provided with a separate server thread. All of the server thread access a commonly shared data management system.

2. Overall Architecture

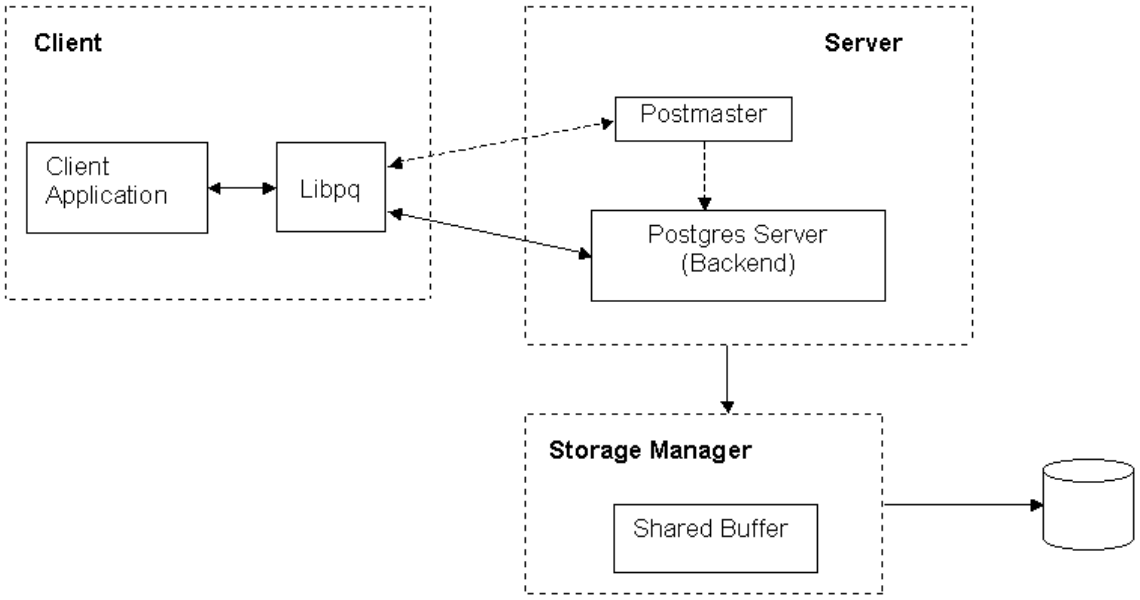

Figure 1 PostgreSQL System Concept Architecture

The front-end of PostgreSQL is a typical client-server architecture, while the back-end is a mixed architecture of layered and pipeline design.

Sub-system descriptions

1) Libpq is responsible for handling the communication with the client processes: establishing connection to postmaster; obtaining postgre server thread for the

operational session. It forwards user's operation requests to the back-end. The operation request is forwarded on an as-is basis.

2) The server side is composed of two sub-systems: the postmaster and the postgre server. The postmaster is responsible for accepting incoming connection request from client, performing authentication and access control on client request, and establishing client to postgre server communication. The postgre server handles all queries and commands from client. PostgreSQL is a "process per-user" model, which means that one client process can be connected to exactly one server process. Since the postmaster controls incoming all requests from clients and invokes new postgres without network connection, one implication of this architecture is that the postmaster and the postgres always run on the same machine (i.e. the database server), while the front-end (client) application can run anywhere. Because of this, the scalability of PostgreSQL is limited and PostgreSQL is usually used in relatively small database application.

3) Storage Manager is responsible for general storage management and resource control in the back-end, including shared buffer management, file management, consistency control and lock manager.

Concurrency Control

Multiple threads of PostgreSQL can be executed concurrently accessing a shared data storage. We hereby define a service thread that is performing a data reading as reader, and a service thread that is performing a data writing as writer. In PostgreSQL, readers do not block writers and writers do not block readers. A writer only blocks a writer if they are writing to the same data entry. In the above case, PostgreSQL provides two solutions (based on ISO SQL standard): read committed and serializable. In the case of read committed, the writer will read the new values before performing its write operation. In the case of serializable, the writer will abort if the data value has been modified since it began its transaction.

3. **The Server subsystems**

The server host of PostgreSQL largely consists of two parts: Postmaster and Postgres. When a client (front-end) sends a request to access database in the server, the postmaster of the server spawns a new server process, called postgres, which directly communicates with the client. Hence, the postmaster is always running, waiting for requests from a client, while the postgres, which is a process, starts and stops upon the request of clients.

After a connection is established, the client process can send a query in plain text form to the back-end. There is no parsing done in the front-end. The server then parses the query, creates an execution plan, executes the plan, and transmits the retrieved tuples to the client over the established connection.

Query/Command Processor

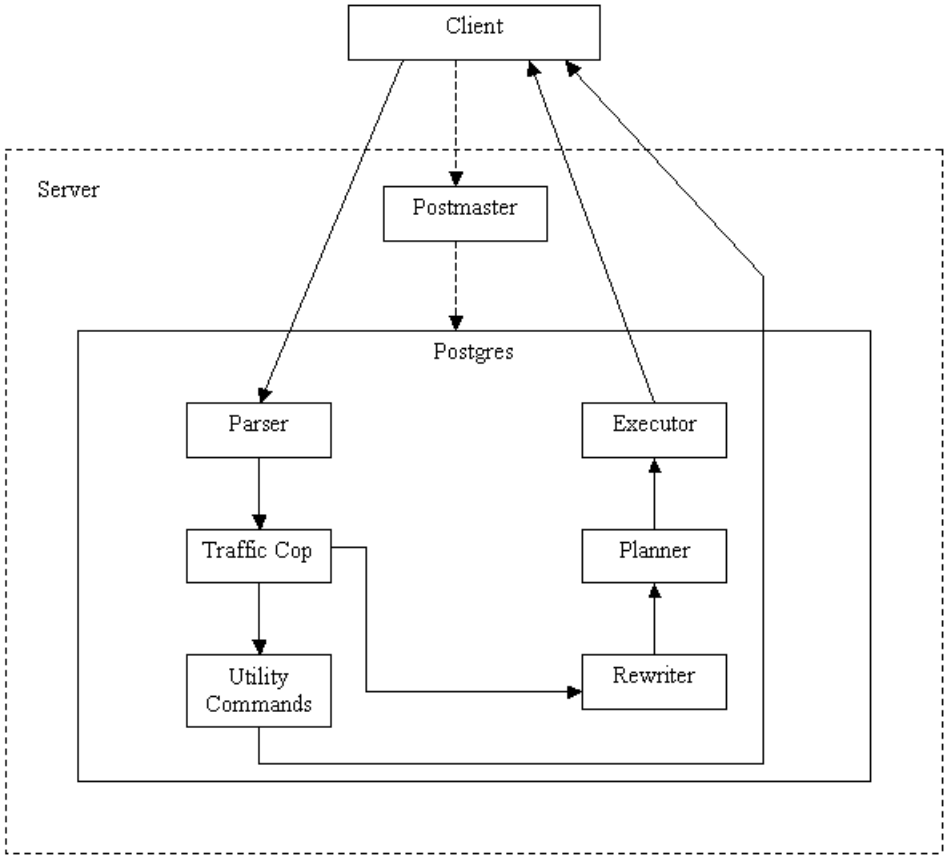

Figure 2 Query/Command Processor Architecture

Figure 2 shows the conceptual architecture of the PostgreSQL server structured as a pipeline. It shows the general control and data flow within the back-end from the time the back-end receives a query to the time it sends the results. The six major subsystems exist in the server:

1) The Parser first checks the query transmitted by the application program for valid syntax. If the syntax is correct, a parse tree is built up and handed back; otherwise, an error is returned. Then, the parse tree is transformed into internal formula used by the database back-end.

2) The Traffic Cop identifies the query as a utility query or a more complex query. The complex queries in PostgreSQL are select, insert, update and delete. These queries are sent to the next stage (i.e. Rewriter.) The utility queries are sent to the Utility Commands.

3) The Utility Commands handles queries that do not require complex handling. Vacuum, copy, alter, create table, create type, and many others are handled by the utility commands.

4) The Query Rewriter is a subsystem between the Parser and the Planner. It processes the parse tree passed by the Traffic Cop and, by applying any applicable rule in present, it rewrites the tree to an alternative form. This stage enables PostgreSQL to support a powerful rule system for the specification of views and ambiguous view updates.

5) The planner provides an optimal execution plan for a given query. The basic idea of the planner is cost-estimate-based selection of the best plan for a query. It first combines all possible ways of scanning and joining the relations that appear in a query. All the created paths lead to the same result and the planner estimates the cost of executing each path. After all, it chooses the cheapest path of all and passes to the Executor.

6) The executor takes the plan passed back by the planner and starts processing the top node. It executes a plan tree, which is a pipelined demand-pull network of processing nodes. Each node produces the next tuple in its output sequence each time it is called. Upper-level nodes call their subnodes to get input tuples, from which they compute their own output tuples. Upper-level nodes are usually join nodes. Each join node combines two input tuple streams into one. In contrast, bottom-level nodes are scans of physical tables, either sequential scans or index scans. The executor makes use of the storage system while scanning relations, performs sorts and joins, evaluates qualifications and finally hands back the tuples derived.

Query/Command Utility

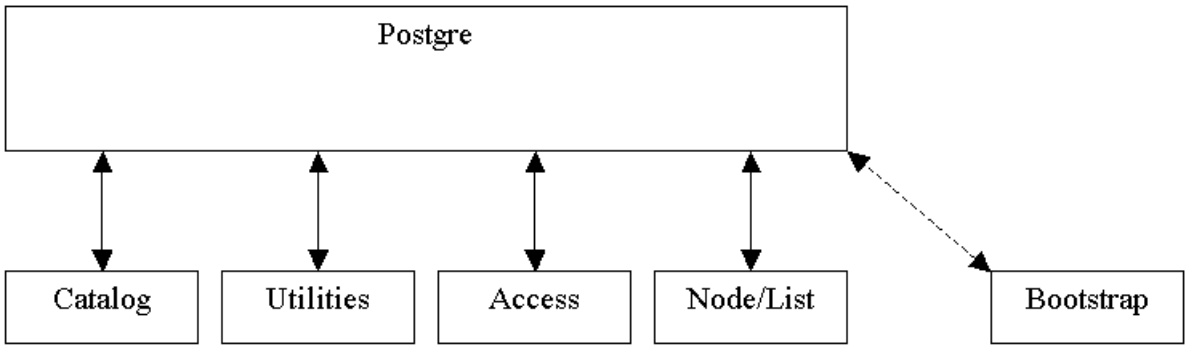

Figure 3 Query/Command Utility Architecture

Figure 3 shows the conceptual architecture of the PostgreSQL server utilities structured as an object-oriented design. Each utility component is distinct in its functionality as described in detail below:

1) Catalog: it provides system catalog manipulation, and contains creation and manipulation routines for all aspects of the catalog system, such as table, index, procedure, operator, type, aggregate, etc. The catalog module is used by all sub-systems of the back-end.

2) Access: it defines data access for heap, indexes and transactions. Its function is three folds: to provide common data access routines; to provide data access structure in the form of hash, heap, index, btree, etc.; act as phase manager during transactions. The access module is used by all sub-systems of the back-end.

3) Nodes: nodes/lists module defines the creation and manipulation of nodes and lists, which are containers of request and data during query processing. The nodes/lists submodule is used by all sub-systems of the back-end except traffic cop.

4) Utils: utils provides various utilities to the back-end, such as initialization, sort routines, error log, hash routines, etc. It is widely accessed by all sub-systems of the back-end.

5) Bootstrap: the bootstrap module is used when PostgreSQL is being run for the first time on a system. This module is required because postgreSQL commands commonly access data table. Such data tables does not exist when Postgre is been run for the first time.

Catalog system

PostgreSQL uses catalogs to a much greater extent and context than other DBMSes. PostgreSQL not only uses catalogs to define tables, but also uses it to describe datatypes, functions, operators, etc. This feature provides much greater user extensibility and control. User defined datatypes are used to associate new data items particular to specialized databases; User defined functions can be either standard functions or aggregate functions; User defined operators can be used in expressions as standard expression. All of the catalog items are maintained and accessed via the catalog sub-system, providing a uniform organization.

4. The Storage

It provides uniform data storage access for the back-end. Only one storage module is active on a PostgreSQL server. The functionality of the storage module includes: provide shared memory and disk buffer, mediate access to kernel file manager, and provide semaphores and file locks. The storage module is used by rewrite & path generation module and command module.

PostgreSQL uses non-overwriting storage management, which means updated tuples are appended to the table and older versions are removed sometime later. PostgreSQL achieves cache synchronization by using a message queue. Every back-end can register a message which then has to be read by all back-ends.

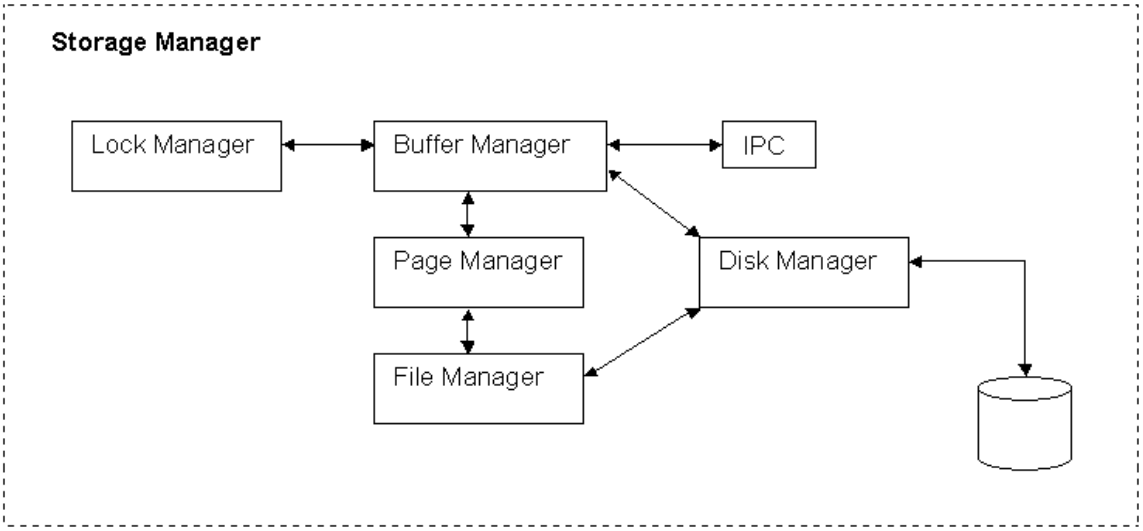

Figure 4 Storage Manager Architecture

- File Manager: provides management of general files and large buffered files.
- Buffer Manager: provides management of shared buffers.
- Page Manager: uses LRU algorithm to manage pages.
- Lock Manager: provides “read” and “write” locks to achieve consistency.
- IPC: realizes cache synchronization.
- Disk Manager: provides interface for physical storage/disk.

This figure shows the conceptual architecture of PostgreSQL's storage manager. In this figure, double-headed arrows indicate dependence on or from all of the storage manager subsystem.

5. Use Case: Query Workflow

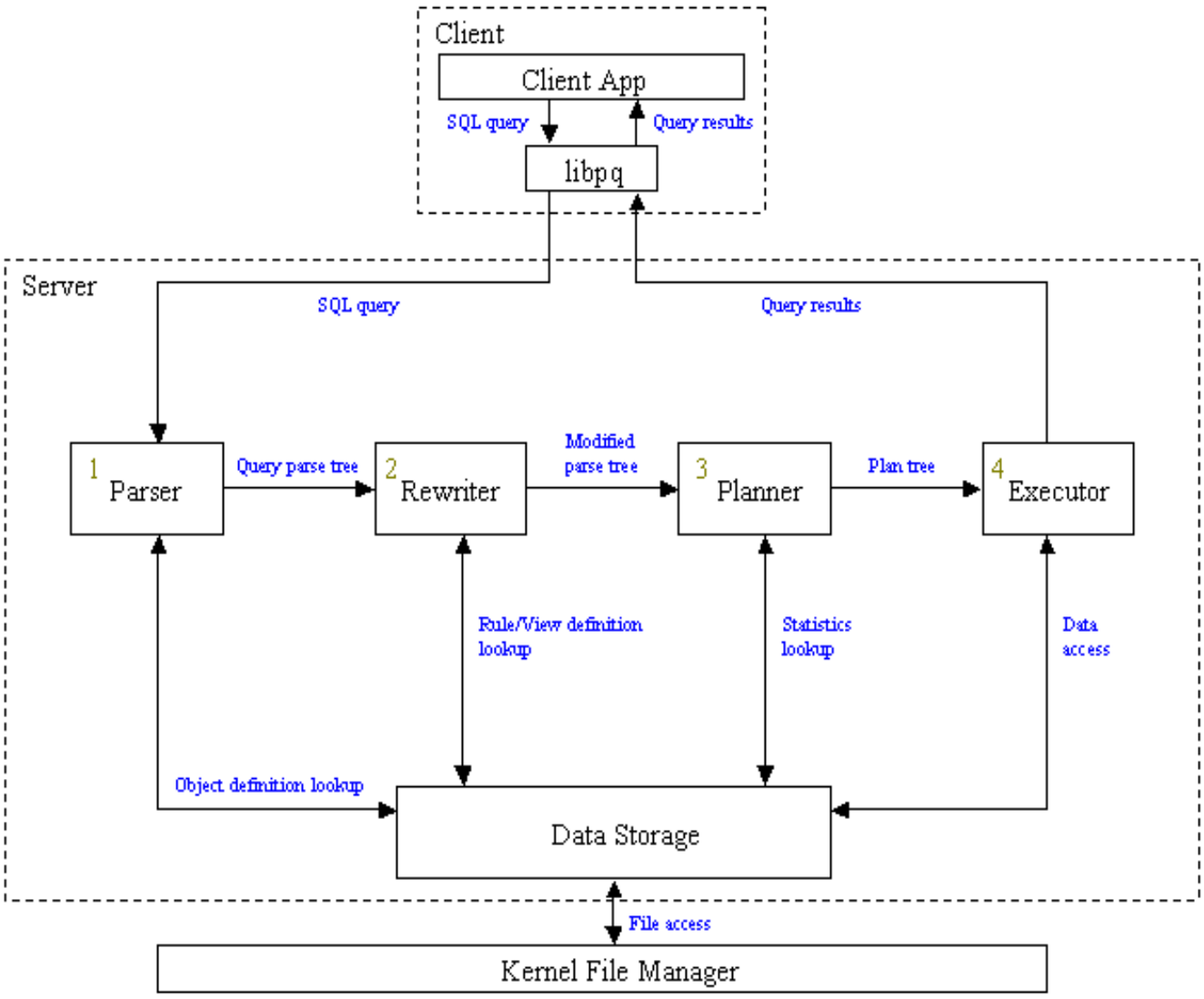

Figure 5 Work Flow of Query

1) The SQL query string is converted into a query tree.
2) The query tree is modified by the rewriter in the following way: the rewriter looks up keywords in the query tree and expand it with the provided definition.

3) The Planner takes in the modified parse tree, generates all possible query paths. The planner then evaluates the paths to determine the optimum path and establishes a query plan for this path.

4) The query plan is transformed into a series of executable SQL queries and processed to obtain results.

6. Conclusion

The server architecture of PostgreSQL has some advantages. First, it pre-filters an incoming query into utility and complex query. This step reduces needless overhead for queries that don't require complex handling such as rewriting, and planning. Also, the system is likely to find a good query plan without any explicit user's assistance. The catalog system provides flexible user control for the definition of new datatypes, functions, and operators.

On the other hand, the centralized nature of PostgreSQL makes it ill-suited for the management of large-scale databases. Furthermore, the database to be accessed must reside at the same location as the server process.

The evolvability of PostgreSQL is two sided. On the positive side, PostgreSQL's flexible catalog system and elegant client-server mode makes it well suited for home office database management and for applications requiring specialized data manipulation and definition. On the negative side, PostgreSQL's centralized distribution and lack of location transparency, makes a very undesirable choice for large-scale or distributed database management.

Reference:

Assignment 0