CISC 322

Fall 2010

Assignment 1

Conceptual Architecture of PostgreSQL

PopSQL

Andrew Heard – 5901298 – andrew.heard@queensu.ca

Daniel Basilio – 5931724 - 8djdb@queensu.ca

Eril Berkok – 5805792 – 8ejb@queensu.ca

Julia Canella – 5834622 – 8jcc5@queensu.ca

Mark Fischer – 5482510 – mark.fischer@queensu.ca

Misiu Godfrey – 5676532 – 7mg5@queensu.ca

Table of Contents

Abstract

In this project the conceptual architecture of PostgreSQL was uncovered in the hopes of eventually deriving a concrete architecture. By combing through a variety of documentations and comparing sources, a general view of the conceptual architecture emerged. Many ideas of how the system was structured were considered and discarded but eventually one was decided upson. What we found was generally an object-orientated architecture, which broke down into several subarchitectures: client/server for the front-end, implicit invocation for the postmaster, pipe and filterfor Postgres Server, and object oriented again for Database Control. With these conclusions in mind, the next step is to delve into the code structure and derive a concrete architecture.

Introduction & Architectural Overview

PostgreSQL is an object oriented, open source relational database system. It is an objected-oriented architecture because it consists of many subsystems that can interact and work together to access the database. A layered architecture was considered, but since both the Database Control and the Server Process need to access the General Utilities, it was concluded that an object-oriented structure was better fitting for the system. A Client-Server model was also an alternative, but since the Client Processes only act under this model it was discarded as well. A Repository model was also rejected since even though the subsystems were working on database data, they were not actually working on a *shared* data structure.

Within PostgreSQL there are several different systems used to implement the database and these are thelarge subsystems, which make up the overall object oriented structure of PostgreSQL. These subsystems are the Client Processes, Server Processes, Database Control, and General Utilities (see Figure 1). They are constantly communicating back and forth to be able to process user demands and write and retrieve data. Within these subsystems, architectures such as pipe and filter, client server, and implicit invocation are used to control the flow of information and interact with the client.

To be able to understand data flow, how the subsystems work with each other, and why they are implemented in a certain architecture, it must first be understood how PostgreSQL actually works. Firstly, there is exactly one client process connected one server process. The client application may not produce SQL statements but may use procedural languages or so on. Therefore the client interface library will interpret and convert into the proper SQL statements that the server understands. Many clients based on procedural languages, such as PL/pgsql, produce database queries that the server would not understand. A single server process can support many different versions of protocol but is informed of the kind the client is using when it is spawned. To control that only one client is connected to only one server, there is a master process (called the Postmaster) that spawns a new server process each time a client requests a connection. Once created, a combination of table locking and Multi-Version Concurrency Control(MVCC) ensures that the numerous server processes do not corrupt the data during concurrent data accesses. The client process, once connected, can send a query to the server process. The server process is responsible for many things. Firstly, it's responsible for parsing the query and identifying the type of query (complex or simple). Then the rewriter will turn a higher level

query into a lower level code. Next, the planner/optimizer creates an execution plan to execute the query in the most efficient path. The executor is then given the execution plan. The executor canThen the retrieved rows from the database are given to the client process over the connection created between it and the server process.

To create this reportresearch was collected and through study of documentation, a conceptual architecture was formed(Figure 1). This overall understanding of the architecture was concluded after considering and rejecting alternative architectures and then a closer study of each subsystem was performed. It was found that even though the overall architecture was object oriented, the subsystems used other forms of architecture. Through understanding what these subsystems do, it was easier to infer why these types of architectures were chosen to be included. Also by analayzing how these subsystems work, data flow can be traced throughout the system for simple and complex queries. As can be seen in Figure 2, each object within the subsystems plays its own role in how the query from the client is received and dealt with.

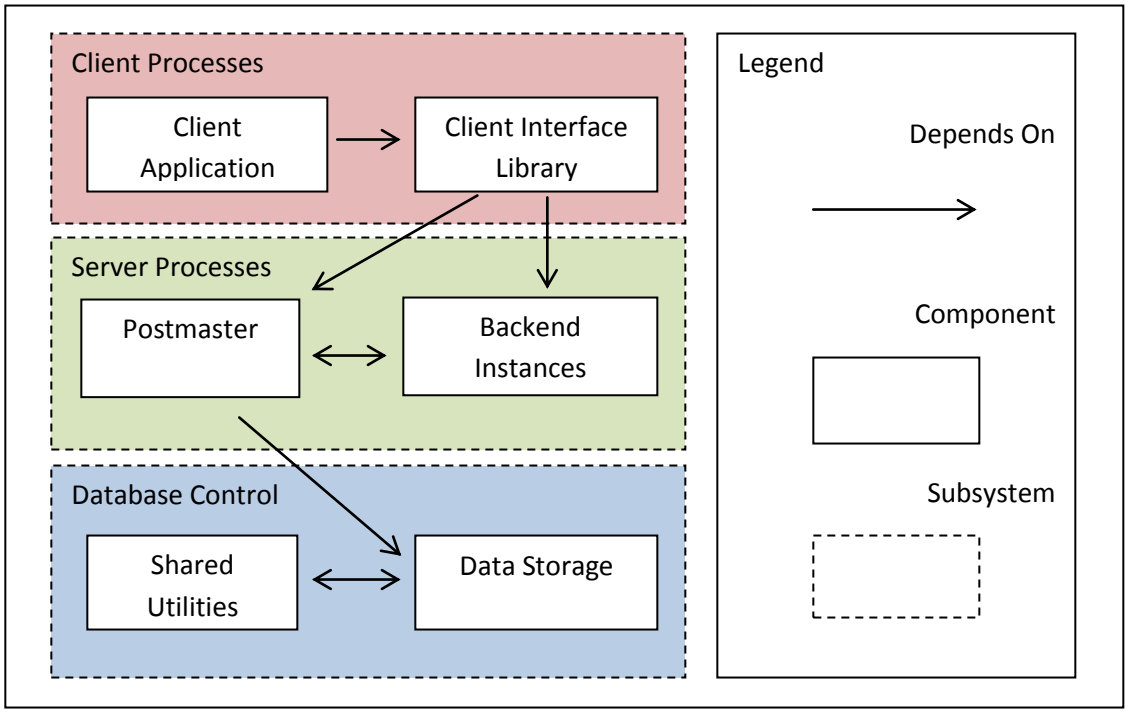

Figure 1: The overall conceptual architecture of PostgreSQL.

Front-End

Client Application and Client Interface Library

The first subsystem of the PostgreSQL architecture has two sub-components: the client application and the interface library. The client application is whichever medium the user is using to interact with the database. In order for the database to understand exactly what the user wants, there has to be a way for the client application to translate its request to the server. This is what the library is there for.

Client applications come in all kinds and sizes. First and foremost, there are multiple UIs that the user could be using. PostgreSQL supports many different ways of interacting with the database. Users can be using a variety of different applications, such as Mergeant, PGInhaler, SQuirreL, RISE, and many more.[13] Beyond this, users can be using different operating systems, and some applications support multiple operating systems as well. This means that the different ways a user can access PostgreSQL is very numerous. However, not all of these different applications or OS's speak in the same way. This drives the need for a central library for each application to talk through.

The Client Interface Library exists as a way for all the different applications to the database. It is essentially an API, just a list of different applications that can talk to the database, and the necessary functions to be able to do so. It is a crucial aspect of the overall architecture because without the ability to translate different applications to something the postgres process can understand; there would be no way for the user to use the database. Also, such a large library allows for many different ways to communicate, not limiting the user to a particular interface, which is always a bonus.

The PostgreSQL server is made up of two separate objects, the postmaster and the back ends. The postmaster is what receives all front end requests to access the database and then chooses a back end to match up to that front end. It does this by creating a back end instance and then linking the front end to that instance. After the link is created, the front end and back end instance can then communicate without having to go through the postmaster, freeing it up to listen to more requests.

The postmaster, despite being renamed in version 8, still exists. It has been aliased to the name of a master postgres process, but still performs the same functions as a postmaster and so we have left the name the same. This doesn't change our architecture because it always performed the same functions, and the postmaster itself is a portmanteau of POSTgres MASTER. [2, 11]

The postmaste is registered to listen to any and all client connections to the postgres database and then reacts when it receives one. When it is not currently receiving a call or creating a back end process, it sits operational but inactive waiting for another event to occur. In this way it is similar to a daemon thread in Windows. (that is, a process that runs in the background outside of user control) This is unlike the back end instance that is started by the postmaster and ended when the user logs off. The postmaster serves as a master back end instance, but is one that is not shut off and creates all other instances. [11]

One of the main issues with the postmaster is that the way it is structured requires the different back end instances and the postmaster process to be on the same machine. The postmaster has access to a large number of different back end instances that it can create. It also controls a single cluster of databases that the different instances can access. When it receives a call it creates one back end instance and links the client and instance. However, for this to be possible, the postmaster and back end instances must exist on the same machine. That means that there is no way to scale out the back end or postmaster.

The fact that they are on the same machine poses a problem when speaking about scalability. The fact that the postmaster creates each instance means that connections between the front end and back end are created quickly, most often without the user even experiencing a delay. The downside to this is that

they would be unable to scale out the back ends; they can only be scaled up. So this means you can have a machine that can run a thousand different back end instances, each one connecting to a different user, but this cannot be moved out to different machines. When a business or user decided that they need access to more back ends than they currently can, the only option would be to scale up the machine that currently holds the postmaster and back ends. This is possible up to a point, meaning that the postgres database cannot grow indefinitely, and as more back ends are added on, the speed of each will decrease due to the machine being overworked. It is therefore best suited for small to medium sized applications. It can be adapted to much larger systems but there must be some reconfiguring of the server processes for this to happen. For example, Yahoo uses Postgres with a heavily modified back end.[12]

Postgres Server

General Architecture

The Postgres server utilizes a pipe and filter architecture. This is because each component of the server process performs a highly specific task on an input, and passes on its results to a successor. This is done in a sequential order, with minimal non-linearity. At the beginning it is provided with a query, and incrementally transforms it into a set of data that is provided to the client. Each process makes use of a separate data table, whether it be for symbols or libraries. In fact, the Executor Stage even goes so far as to access the data tables themselves.

Parser

The parsing stage begins by accepting an SQL query in the form of raw ASCII text. The output emerges from the parsing and transformation of the input. First, the *lexer* does pattern matching on the text: recognizing identifiers and other keywords, amongst other things. Each one is converted into a token. These tokens are then put through the *parser*, which assembles them into a parse tree.

The *lexer* is produced using the Unix tool called *lex*. It begins as a schematic contained in scan.l that lays out the identifiers and keywords to be recognized by the lexer. *Lex* then turns this into a C program that is then compiled using a standard C compiler. The *parser* is produced using the Unix tool *yacc*. This subcomponent begins as a collection of grammar rules in the file gram.y, which *yacc* then turns into a C program that is in turn compiled by a standard C compiler (similarly to the lexer). Together, these two C programs perform the entirety of the parsing stage on the input query.

The *parser* checks whether the SQL query matches the syntactic rules of SQL but it does not understand the semantics of the query. If the SQL query is syntactically invalid, an error will be produced and the client will be notified; the query will not pass on to the next stages of query processing. Otherwise, the query, which is now assembled into a parse tree, is passed along to the Rewriting stage.

Traffic Cop

The traffic cop is the agent that is responsible for differentiating between simple and complex query commands. Transaction control commands such as *BEGIN* and *ROLLBACK* are simple enough so as to not need additional processing, whereas other commands such as SELECT and JOIN are passed on to the

rewriter. This discrimination reduces the processing time by performing minimal optimization on the simple commands, and devoting more time to the complex ones.

Rewriter

The beginning of this stage is composed of a rewriting of the parse tree received from the Parsing Stage. This involves expansion of subqueries into lower order commands.

Planner/Optimizer

The next step is to determine the path that requires the least computational complexity. This is the role of the Planner/Optimizer Stage.

SQL queries can be executed in many different orders that produce the same result. All of these combinations are represented in the input parse tree taken from the Parsing Stage. Some of the keywords and identifiers are commutative, which is what enables the multitude of different paths in the parse tree. The Planner/Optimizer will do one of two things. If there is a certain number of possibilities less than a threshold amount, it will exhaustively consider every possible path of the parse tree and find the one that is least computationally complex. Otherwise it will use a genetic algorithm to find an efficient path, though not necessarily the best one given that going beyond the threshold yields a much larger number of paths to traverse.

Once all of these computations are performed and the best plan is constructed, it is passed onto the Executor Stage.

Executor

The executor receives the plan produced by the planner/optimizer and extracts the data necessary from the database tables. The executor recursively goes through plan, which is a tree, and performs the required action at each node. If information about the sub-path is necessary, the executor will proceed to the children before executing the node, otherwise the node's action can be executed immediately. This is done in a pipeline manner, meaning rows are output as the nodes are taken into the executor as input rather than waiting to output the complete block of results data (i.e. batch processing). At this stage, "the executor mechanism is used to evaluate all four basic SQL query types: SELECT, INSERT, UPDATE, and DELETE."[14] The output is delivered back to the client.

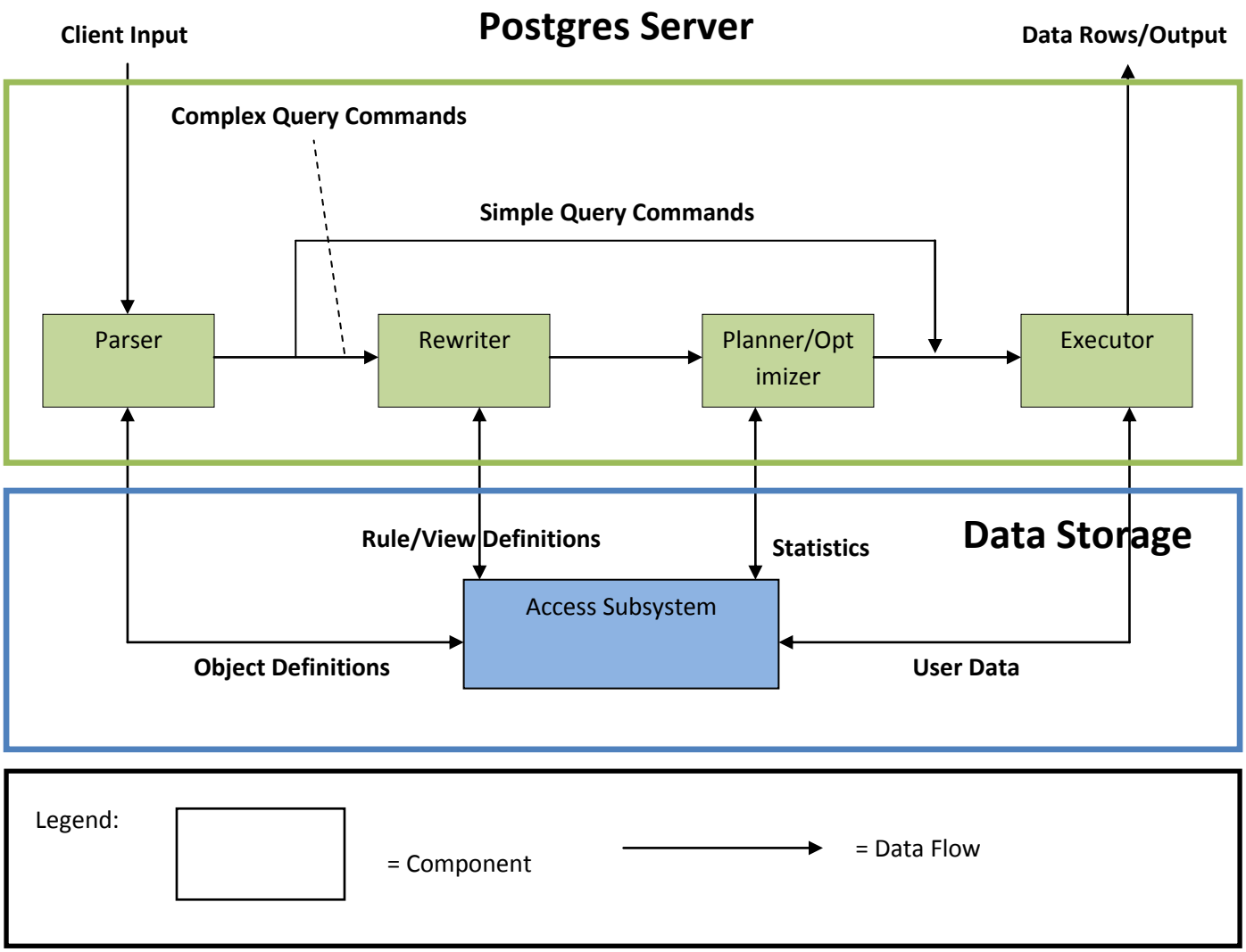

Figure 2: Conceptual architecture of Postgres Server component.

Back-end Control

Shared Utilities

While the backend server instances take care of clients, there are many services that are offered over the entire system. These utilities are used by most of the subsystems in PostgreSQL and are much akin to the Shared Components section that has come to be associated with database architectures.

Memory Manager

The Memory Manager lends speed to the system by managing what is stored in local memory and what is stored on the hard disk. The memory manager tries to organize which data are in memory and which are on the disk for high performance.

Replication and Loading

The replication and Loading subsystem helps provide concurrency control for the system. It is used to provide snapshots of the current database for the Access Manager to make sure commands have stable data to work with.

Utilities/Libraries

Utilities contain functionalities and code that is often used and doesn't contain many external dependencies. Most subsystems in database control will use utilities/libraries

Commands

The Commands subsystem acts most closely to the reference architecture's Batch Utilities. Commands perform operations on the data. Commands involve a series of operations sent to the Disk Interface. It also holds much of the business logic associate with actions over the data. For example, when a table is created, Commands will insure enough space is set aside, but it will also update the system catalogs to recognize the new table.

Administration and Monitoring

There is a subsection of the Shared Utilities devoted to monitoring and administrating the whole, or aspects of the system. Many of these subsystems are autonomous processes that, once initialized, will carry on their task without interfering with other systems unless required, or requested. These systems, with the exception of the Bootstrap system, are initialized by the Postmaster process when it starts up. These systems will typically deal directly with the disk interface subsystem for reading and writing data and rarely access others directly.

Background Writer

PostgreSQL backs up its databases by maintaining a write-ahead log. Due to the multiple-access and high-speed natures of a database, many transactions will be held in memory rather than disk storage and will only be written to disk periodically. Because of this, any transactions held in memory are vulnerable to loss due to unforeseen circumstances. This log records any changes that a client has made to the database since the last write to disk. In this way, a recovery of the current database can be obtained by re-performing these changes from the last backup. Standard output from all subsystems is routed through the system logging aspect of the Background Writer.

Statistics Collector

The statistics collector is a subsystem designed to collect statistics on server activity. It can be optionally enabled or disabled based on the database's configuration settings. Specifically, the statistics collector, when enabled, can record statistics regarding the table and index accesses of the database, the usage of user-defined functions, and commands being executed by any server process. The collector transmits information to other processes via temporary files. By default, the collector's settings apply to all server processes, but can be disabled individually.

Each individual process transmits it's own statistical counts to the collector periodically, and the collector itself writes a new report based on a defined interval. Each client server can then request to view the most recent report emitted by the collector.

Vacuum Cleaner

The auto-vacuum subsystem is a collection of processes that run in the background and will identify tables whose memory can be recovered, whose data statistics need to be updated, and to prevent the loss of old data via transaction ID wraparound. This daemon uses data collected by the statistics collector to troll the databases and perform these tasks accordingly. A copy of the data that are deleted from the database are stored in the logs and archives to enable rollbacks and reliability.

Bootstrap

Bootstrap allows the database backend system to start running without the need for a postmaster. Bootstrap mode doesn't parse SQL statements but is necessary to create the initial template database. This is because Bootstrap allows system catalogs to be created and filled from scratch, whereas ordinary SQL commands require the catalogs to exist already.

Data Storage

There exist several subsystems in PostgreSQL that are dedicated exclusively to dealing with the database portion of the system. These processes and libraries control access to and editing of the database and can be subcategorized as follows.

Access Manager

The access subsystem holds access methods and acts as a façade between PostgreSQL server instances and the data. Access receives data from server instances, and (with the use of some business logic) directs requests to the appropriate subsystem. Requests to find or read data are directed to the Disk interface, while requests to perform operations are directed toward the commands subsystem. It holds much the same role as the reference architecture's Access Methods, but doesn't implement any functionality for itself.

Finally, the access subsystem is also responsible for making sure authenticated users only have access to tables that they are permissions for.

Disk Interface

The Disk Interface is in charge of read and write operations. Disk interface uses the buffer manager and lock manager to manage concurrency issues. Internally, data consistency is maintained by using a Multiversion Concurrency Control Model (MVCC). Through this system, the Disk Interface is responsible for providing the database snapshot and insuring that operations don't conflict with one another.

Lock Manager

Because MVCC doesn't allow a user to know if the data they retrieve is the same as the data currently in the tables, there are explicit SQL commands that require a table to be locked while in use. In these cases, the Disk Interface uses the lock manager to perform locks on the tables.

Buffer Manager

The buffer manager interacts with the disk interface to keep information in a buffer. There are local buffers, table buffers, file buffers, and temporary buffers which are used for different purposes.

Back-end Control Dependencies

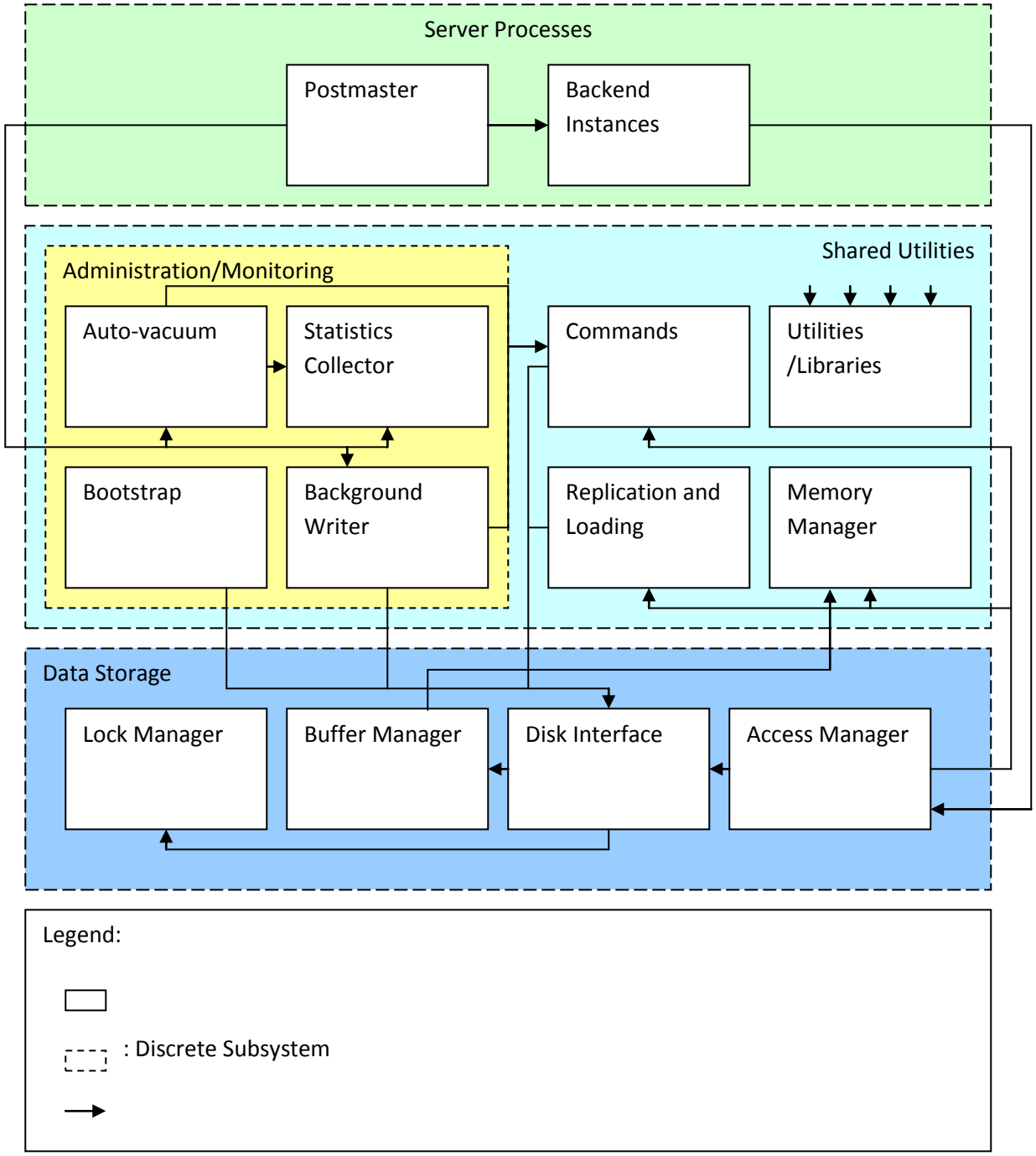

Figure 3: Back-end Control Dependencies

Evolution

Typically with a database management system, a license will net you the back-end and a very rudimentary front-end. Take, for instance, MySQL and its simple command-line interface. When a

developer licenses or acquires a database management system such as PostgreSQL, they will often take it and develop a custom-made front-end for it. So before where there was a simple command-line interface, now there might be a graphical user-interface in its stead. On top of that, a developer may take the front-end and integrate it with PHP/HTML in order to enable web access to their database. You can see this in many instances throughout the internet, whether it be banking systems, record systems, Wikipedia, Google, etc. These are the primary examples of how one may evolve PostgreSQL.

As far as what the PostgreSQL core design team is working towards, they have listed the following features as the priorities of post-9.0 PostgreSQL development (whether they be entirely new features, or evolution of some that are currently within the system):

- Administration and monitoring tools
- Driver quality and maintenance
- Modules and extension management
- Per-column locale/collation
- VLDB? (Very-large database)
- Cloud?

Implications for Division of Responsibility

With multiple developers working on the same architecture, in this case PostgreSQL, there are several things that arise as a result. Given that the front-end implementation often differs between the companies that use PostgreSQL, the PostgreSQL system needs to be developed with modifiability in mind with regards to its front-end. On that note, developers that choose to customize the front-end need to ensure that they are developing to take advantage of the features of the postgres server architecture.

Even within the Postgres server development, steps need to be taken to ensure that all the various components work together. In all likelihood there will be teams within a developer (or spread out across several) that are obligated to develop their components with the live architecture in mind – so that their components work when integrated with one another.

Use Cases

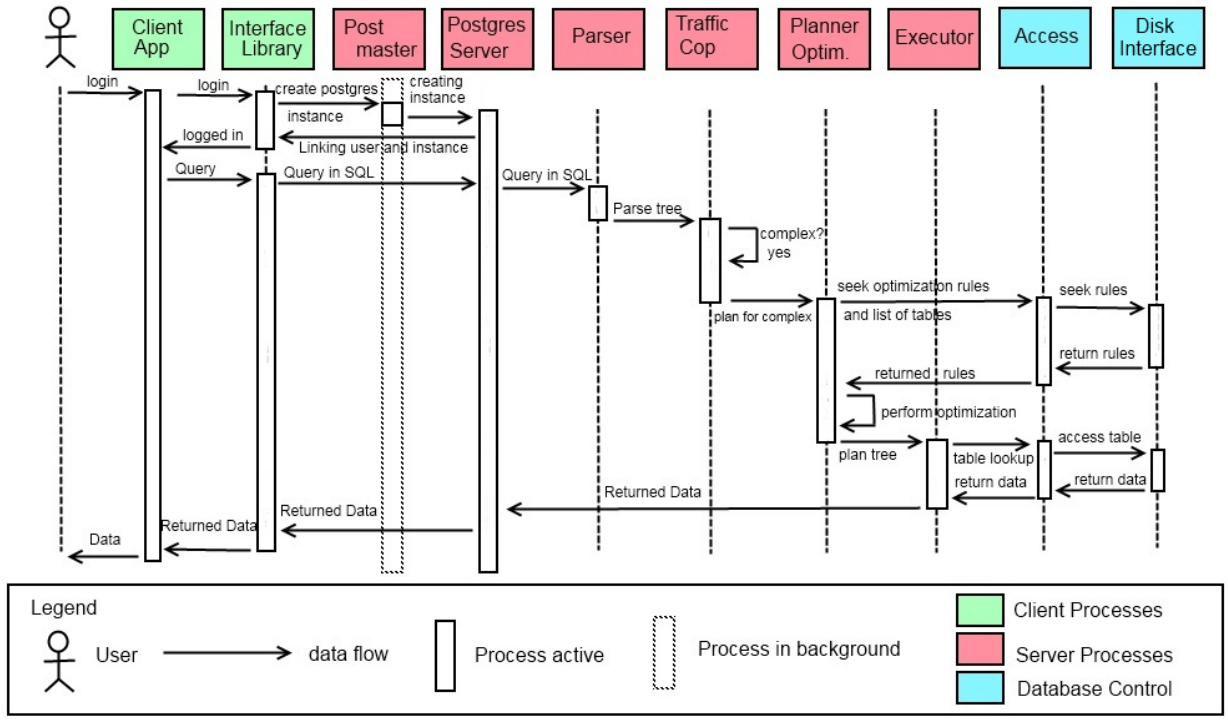

Figure 4: The user first logs in and the postmaster becomes active long enough to create a Postgres Process and link them together so they can communicate without the postmaster. The user then makes a query, and through the Postgres Process it is parsed, authenticated, optimized, and executed, then the data requested is returned to the user.

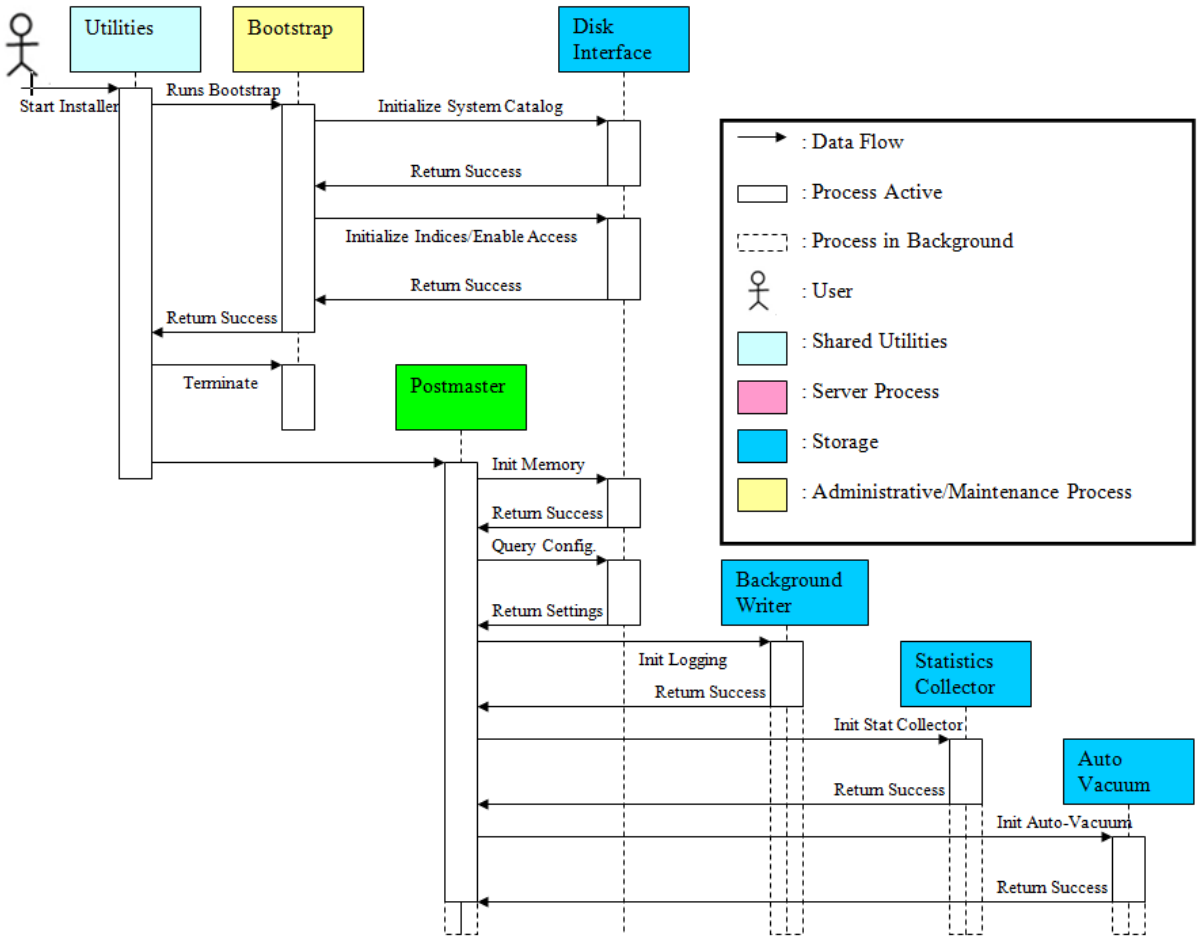

Figure 5: Use Case for creating a new database

Conclusions

The overall architecture of PostgreSQL is object-oriented and repository, with an object for each of the major components: the front-end, the Postgres server and the database control. Many instances of the Postgres server can exist, all accessing the same repository. Each of the major components has its own architecture. The front-end employs a client-server architecture between the client software and the library used to communicate with the Postgres server. The postmaster, a sub-component of the front-end, uses an implicit invocation architecture to listen for connection requests. The Postgres server employs a hybrid pipe and filter/repository architecture in order to process queries. Finally, the database control component uses an object oriented architecture.

In the future we will be able to study the source code of PostgreSQL in order to determine the actual code-level dependencies between the objects in the overall architecture of PostgreSQL. This will allow us to produce a concrete architecture of the system.

Lessons Learned

We learned several lessons during the process of determining PostgreSQL's conceptual architecture. We learned that consistency between all diagrams is necessary in order to retain readability. Also, that it is important to check that documents pertain to the current version of the software. And most importantly, we learned that the reference architecture is an invaluable tool for deriving a conceptual architecture.

Glossary

Terminology

SQL: a standard language for communicating with databases.
ASCII: a character encoding scheme for plain text.
Postgres server: an instance process initiated by the postmaster. It is connected with the Client Interface library to handle queries to the database.
Transaction ID Wraparound: PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is "in the future" and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits at this writing) a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound.

Tools

yaac: a Unix tool, which generates a parser from an analytic grammar. Stands for "yet another compiler compiler".
lex: a Unix tool, which generates lexical analyzers. Commonly used with yacc.

References

[1] PostgreSQL Global Development Group. *postmaster.c*. postgresql.org Source Documentation. Retrieved October 15th, 2010 from http://anoncvs.postgresql.org/cvsweb.cgi/~checkout~/pgsql/src/backend/postmaster/postmaster.c.

[2] PostgreSQL Global Development Group. *PostgreSQL: Documentation: Manuals: PostgreSQL 9.0: postmaster*. PostgreSQL 9.0 Documentation. Retrieved October 15th, 2010 from http://www.postgresql.org/docs/9.0/static/app-postmaster.html.

[3] PostgreSQL Global Development Group. *PostgreSQL 9.1devel Documentation*. PostgreSQL 9.1devel Documentation. Retrieved October 15th, 2010 from http://developer.postgresql.org/pgdocs/postgres/index.html.

[4] Bruce Momjian. *How PostgreSQL Processes a Query*. postgresql.org Source Documentation. Retrieved October 15th, 2010 from http://anoncvs.postgresql.org/cvsweb.cgi/~checkout~/pgsql/src/tools/backend/index.html.

[5] Tom Lane. *A Tour of PostgreSQL Internals*. postgresql.org Source Documentation. Retrieved October 15th, 2010 from http://www.postgresql.org/files/developer/tour.pdf.

[6] Peter Eisentraut. *Most Wanted Future PostgreSQL Features*. postgresql.org Source Documentation. Retrieved October 17th, 2010 from http://wiki.postgresql.org/images/2/24/Most-Wanted-FOSDEM-2010.pdf.

[7] Vikram Khatri, Nora Sokolof, and Manas Dadarkar. *Migrate from MySQL or PostgreSQL to DB2 Express-C*. developerWorks. Retrieved October 8th, 2010 from http://www.ibm.com/developerworks/data/library/techarticle/dm-0606khatri/.

[8] Ted J. Wasserman. *Leverage your PostgreSQL V8.1 skills to learn DB2, Version 8.2*. developerWorks. Retrieved October 8th, 2010 from http://www.ibm.com/developerworks/data/library/techarticle/dm-0603wasserman2/index.html.

[9] Felix Amevor. *POSTGRESQL ARCHITECTURE*. DATABASES. Retrieved October 8th, 2010 from http://amevor.de/?m=20080525&paged=5.

[10] Ewald Geschwinde, Hans-Jürgen Schönig. *PostgreSQL developer's handbook*. Google Books. Retrieved October 8th, 2010 from http://books.google.ca/books?id=LjXBeAWM-KgC&lpg=PP1&ots=fj6VUh2dBD&dq=postgresql%20developer's%20handbook&pg=PP1#v=onepage&q&f=false.

[11] PostgreSQL Global Development Group. *PostgreSQL: Documentation: Manuals: PostgreSQL 9.0: postgres*. Retrieved October 22nd, 2010 from http://www.postgresql.org/docs/9.0/static/app-postgres.html .

[12] Eric Lai. *Size matters: Yahoo claims 2-petabyte database is world's biggest, busiest.* Retrieved October 22nd, 2010 from http://www.computerworld.com/s/article/9087918/Size_matters_Yahoo_claims_2_petabyte_database_is_world_s_biggest_busiest

[13] PostgreSQL Global Development Group. *Community Guide to PostgreSQL GUI Tools*. Retrieved October 22nd, 2010 from http://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools

[14] PostgreSQL Global Development Group. 44.6. Executor. PostGreSQL 9.1 Level Documentation. Retrieved October 22nd, 2010 from http://developer.postgresql.org/pgdocs/postgres/executor.html .