Team C-Lion's Assignment 1 Report:

The Conceptual Architecture of PostgreSQL

Yonis Abokar, Qasim Ahmed, Chandler Cabrera, Taswar Karim, Amir-Hossein Khademi, Chirag Sardana, Sarwat Shaheen

Lassonde, York University

EECS 4314: Advanced Software Engineering

Professor Zhen Ming (Jack) Jiang

October 06, 2021

Table of Contents

Abstract

PostgreSQL is a Database Management System (DBMS), known for its reliability, extensibility, and performance. Its reputation can be attributed to it being open source, having had more than 25 years of active development since its creation in 1996; as a result, there are many documents online detailing its design, creation, and expansion. Exploring postgres' functionality, structure, and qualities gave us a better understanding of conceptual software architecture. From our research, we found that PostgreSQL uses many different architectures; this report details its high-level structure. Postgres relies on a client-server architecture to connect the front-end application with the database. The server itself uses a pipe-and-filter architecture to execute the client's SQL query, The query is processed through the many subsystems that exist within the system. PostgreSQL performance can be attributed to the steps it takes in this subsystem to optimize based on predicted computational cost. Finally, it uses a repository style architecture to store the data, which allows it to maintain memory integrity even when accessed by multiple users concurrently. POSTGRES also aims to add the fewest features needed to completely support data types. These features included the ability to define types and to fully describe relationships which something used widely but maintained entirely by the user and the evolution of this system has changed over time which has been constantly evolving to address performance issues such as scalability which is constantly evolving due to the being open source and having many developers as contributors.

Introduction and Overview

PostgreSQL is an object relational database management system (RDMS) that is able to evolve rapidly based on the needs of a dynamic industry. In 1986, Michael Stonebreaker created PostgreSQL based on the development of Ingres, one of the first RDBMS [5]. PostgreSQL was designed to be extensible, such that features could be added quickly. For instance, object-oriented functionality was added to reduce runtime using caching and connection pooling, and also to support future query languages. PostgreSQL consists of subsystems that effectively integrate into a modular system achieving robustness. At the highest level of conceptual architecture, PostgreSQL uses a client/server model. In the model, the client is a computer/program that sends requests to a server and the server is the component that responds to incoming requests. The client application sends a request to the database server (postgres) to perform database operations. If the client and server don't reside on the same computer, a TCP/IP network connection needs to be established. To do this, the client sends a SYN packet to the database server. In response, the server acknowledges it and sends a SYN+ACK packet back to the client [3].

The database server can serve multiple connection requests from clients.[5] Each client connection makes a request to the database server and forks a new PostgreSQL backend server process. Once the process is forked, the client and this new server communicate with each other without the intervention of the database's parent process [5]. When a connection comes to an end, the client and the new server processes disappear. Within the database server, a separate server process called the postmaster is always running in the background and is listening to client connection requests.

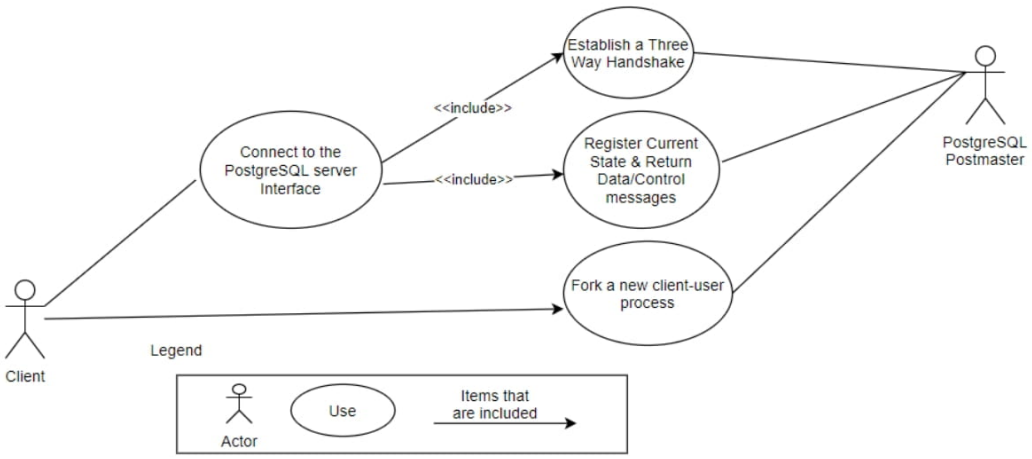

Figure 1*:* Use case diagram depicting client establishing a connection to a PostgreSQL server

With query processing, the subsystem of the backend process receives queries from the client and transforms a query into an optimal form to reduce run time. Afterwards, the backend process communicates with the memory component to retrieve data for the user. The feature of concurrency aids in the system performance with multiple users; yet the transaction of a database can lead to data inconsistency issues. PostgreSQL uses the technique of MVCC which establishes locks for concurrent transactions on the same data rows.

Architecture of Subsystems

As mentioned above, the highest level of the conceptual architecture with PostgreSQL consists of a client-server model which connects server-side processes and front-end applications. The front-end applications can include web applications, graphical interface applications. Client and server don't need to reside on the same hosts.

Client

The front-end applications can include web applications, graphical interface applications. Client and server don't need to reside on the same hosts. As explained earlier TCP/IP connection is needed for client and PostgreSQL processes to talk to each other. When a new connection request comes from the client, the client process sends the request to the postmaster process and the postmaster forks a new process called postgres.

Postmaster

This process is the first process started when you start PostgreSQL. At startup, it performs recovery, runs background processes and initializes shared memory. It also creates a backend process called postgres when there is a connection request from the client process.

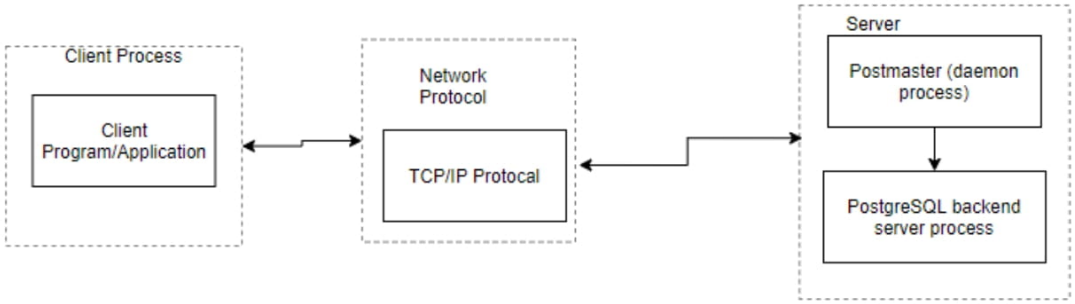

Figure 2*:* Client Server Architecture Model

Postgres backend process

Moreover, Postgres backend server process manages the database files, query processing, and communicates with background processes to process request queries from the client. The PostgreSQL server minimizes performance overhead by only accepting SQL queries to maximize cohesion with front-end programs and reduces computation time that would be spent parsing different languages.

One of the advantages of PostgreSQL minimizes performance overhead is by only accepting SQL queries, to maximize cohesion with front-end programs and reduces computation time that would be spent parsing different languages. It is for this reason that the client's application must use an interface library. The interface translates the frontend developer's intention (in their chosen programming language) to an SQL query that can be read by the postgres server. Once the query processing is done results if applicable are returned to the client process. One of the popular client applications of PostgreSQL is psql, which is a terminal that can be used to execute sql commands

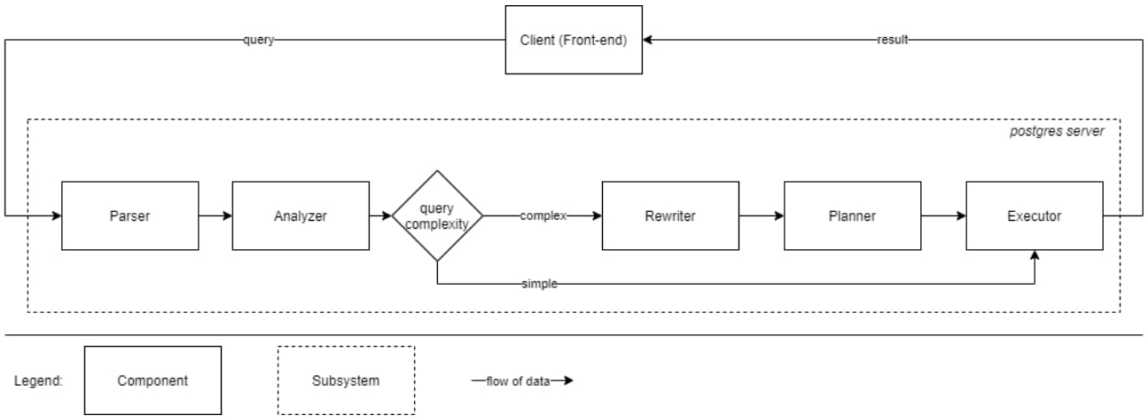

Figure 3: conceptual architecture of postgres server subsystem

Postgres is modelled using Pipe and Filter architecture (*Figure 3)*. Postgres pipelines the plaintext through a number of filters, with each filter processing it for a different purpose before returning it to the client. Due to the sequential nature of Pipe and Filter, performance must be made up in other systems. Despite SQL being a database management system based on tables, the bulk of the processing is done using trees. At each filter, a tree is generated/modified based on the data piped in from the previous step, with the purpose of each filter outlined below. [9]

Parser

The parser converts the plain text (received from the client) into a parse tree. The analyzer takes this data and transforms it into a query tree, organizing the parsed data into its correct data type / function. The requirement for parser is that if there is a syntax error, the parse tree won't build, and an error is returned. [9]

Analyzer

The analyzer runs a semantic analysis of the parse tree and generates a query tree. Each node contains data on the type of command to be run or the data in list form, with each leaf containing the simplest forms of data, like integers or constants. The main function of the analyzer is to separate the parse tree into an internal representation of the SQL statement, giving a general overview of its intended function. [9]

Rewriter

The query tree from the analyzer is passed to the rewriter, which transforms the tree according to the pg_rules system catalog. Any subqueries written into the original plain text are expanded and analyzed. [9]

Planner

The planner filters the query tree from the rewriter and generates a new type of query tree (called a plan tree). This tree is optimized based on two types of cost: startup (the amount of computation done before the first tuple is fetched) and run (cost to fetch all the tuples in the tree). The planner achieves this by carrying out in three main steps. First it performs preprocessing on the query tree. This is akin to simplifying a math statement - mathematical operations are simplified to constants, boolean expressions are normalized, and mathematical expressions are flattened. The planner then tries to find the cheapest access path. An access path is a unit used by postgres to estimate the computational cost of various SQL operations. Finally, by using the cheapest access path through the query tree found in the previous step, a more efficient query tree is generated. [9]

Executor

Using the query tree piped in from the previous step, the executor recursively performs the associated operations at each node of the plan tree. After each node is visited and all operations have been carried out, the resulting data is sent back to the client. The executor has four main types of operators: access, join, sort, and aggregation methods. Access methods are used to retrieve data from storage, with improved performance coming from the use of different types of scans (sequential, index, and bitmap). Join methods are used to combine two separate tables from the database into one table. Postgres supports sorted merge joins (joining two tables using their common tuple), nested-loop joins (each table is looped through until their common tuple is found), and hybrid hash joins (tuples are represented in hash tables, and only matching hashes are compared). Each of these joins have their own advantages in terms of time, computation cost, and memory therefore achieving better performance. Within the executor, sorting follows a main rule where small amounts of data are sorted in-memory via quicksort and larger sizes are sorted using other sorting algorithms determined by Postgres rules. Furthermore, aggregation is achieved using two main types of methods. Sort-based aggregation is used when the number of distinct groups is very large. Otherwise, PostgreSQL uses an in-memory, hash-based approach to save on memory-usage. [9]

Database Control

There are two main subsystems to this component: access and storage. The access subsystem is solely responsible for retrieving data from the server. It does not have the capability to change or manipulate the data

in any way. If a filter requires data from storage, this subsystem can index, scan, search, compile, and return the queried data. The storage subsystem is responsible for ensuring data concurrency and security. The architecture of this subsystem is detailed below. (Heard, A. et Al, 2010)

Memory Architecture

The two main goals during the development of PostgreSQL were implementation and management. The purpose was to place file-system files for data storage (‘cooked’ files), rather than retrieving physical data from disk partitions. These file systems are organized within directories called tablespaces. Tablespaces are effective because they are located on separate disks, allowing parallel access reducing computation time. The PostgreSQL backend server is allocated local and shared memory. The server uses local memory for query processing and shared memory to access shared data between multiple server processes. In terms of architectural styles, shared memory can be classified as repository style as multiple clients can access the shared memory. The components of shared memory consist of a shared buffer pool and a WAL buffer. [10]

Shared buffer pool

In this memory area, all the pages and tables which are needed for the query are loaded directly from disk, to reduce the disk access and hence results in faster processing. This space is used by different processes for quick read/write of data [7]. The main advantage is that memory read/write is faster than read/write on the disk. In database jargon the data that is modified in the buffer pool is referred to as dirty data and needs to be written to the disk for permanent storage and consistency reasons [7].

WAL Buffer

This is the place where WAL data is temporarily stored before it is written to the disk. In other words, metadata for changes in the database is stored as WAL data. This data is extremely helpful in reconstructing actual data in the event of a loss or if recovery is required [4].

Using the layered design does have performance drawbacks, due to double buffering when stored data is first fetched from disk (the filesystem cache) and transferred into the PostgreSQL buffer pool. [2]

***Figure 4**: High level view of layered memory architecture [5]*

Qualities of PostgreSQL

Transaction Management

In databases, concurrency implies the ability for multiple users to make requests on DBMS. Given the importance of concurrency, it is imperative to eliminate any issues pertaining to multiple users accessing data. For instance, a transaction reads data written by another uncommitted transaction known as dirty read [5].

Another known issue named non repeatable read occurs when a transaction reads the same data twice and the data value changes during querying for the second time creating data inconsistency. Generally, the DBMS ensures reliable data and adheres to ACID (Atomicity, Consistency, Isolation and Durability) by means of transaction locking. The technique of locking refers to a mechanism where every transaction cannot read or write data unless read/write locks are obtained [5].

Interestingly, the advantage of PostgreSQL is the utilization of the MVCC (Multi-version Concurrency Control) isolation scheme which maintains different snapshot versions of data from table rows ensuring data consistency [5]. In terms of performance, MVCC is more beneficial than using standard locking due to the removal of conflict between querying data and writing data. In terms of the implementation of MVCC, PostgreSQL creates a snapshot of all transactions by associating state information for each executed command, for instance a SELECT statement creates a transaction ID for the present executed SQL statement known as xmin [11]. In addition, the snapshot contains a transaction ID called xmax pertaining to concurrent executions that are updating/deleting data accessed by xmin [11]. Ultimately, the transaction IDs are written into a log file called pg_xact that maintains the status of committed, aborted, running transactions. With this structure of tracking transactions, the phenomena of dirty read and non-repeatable read are eliminated by virtue of tracking transactions for table rows. For instance, read committed is the isolation level that prohibits dirty read which creates a snapshot when querying data when executing a SELECT statement [5]. From there, if an update/delete statement is running parallel with the SELECT statement, the protocol will check the transaction ID of the running update/delete statement in the log files and when the update/delete statement is committed then afterwards the SELECT statement will execute [5].

Another MVCC protocol is repeatable read isolation level which also uses snapshot isolation for each transaction. In contrast to read committed, repeatable read produces an error message given that it will identify any transaction ID updating/deleting pertaining to the SELECT transaction. Lastly, concurrent transactions involving INSERT statements or multiple updates on the same data results in a serializable isolation level. Serializable is the strictest isolation level using two-phase locking [5]. The process of two-phase locking organizes concurrent execution in a serial manner using designated locks for specific commands such as access share for select statements and row exclusive for update/delete/insert statements. MVCC makes sure locks cannot be obtained unless it is released for that specific command. Ultimately, serializability removes blocking resulting in a deadlock [5].

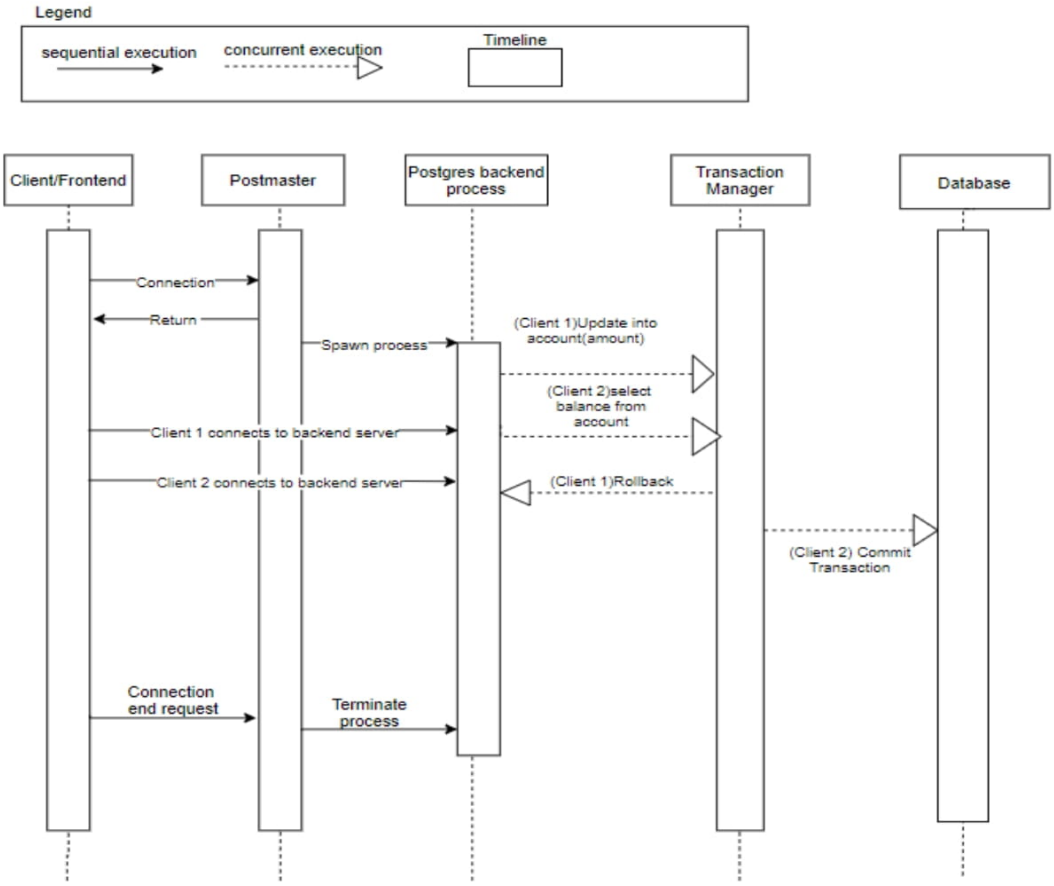

***Figure 5**: Sequence Diagram for Repeatable Read Isolation Level*

Control and Data Flow

The beginning of the flow of data begins with the aforementioned *Postmaster* process (The Postmaster is a Client Communications Manager in charge of establishing a connection with the client, remembering the current state of the caller, responding to SQL commands and returning data and control messages as required). Once the communication is established - the user is then allocated a new process by the *daemon/main process* through forking, and the client can now send SQL commands to the newly created process.

Upon the client's first SQL command, PostgreSQL assigns a thread of computation to the command by making sure the thread's data and control outputs are connected via the communications manager to the client. The User's query is executed by invoking a code in the Relational Query processor which comprises a set of modules such as the Parser, Analyzer, Rewriter, Planner & Executor (detailed in *Architecture of Subsystems)* which work together in order to execute and deliver the query results to the client interface. As the process of parsing and analyzing queries are taking place, subsequent amounts of data are being accessed, stored and retrieved from the local and shared memory as required.

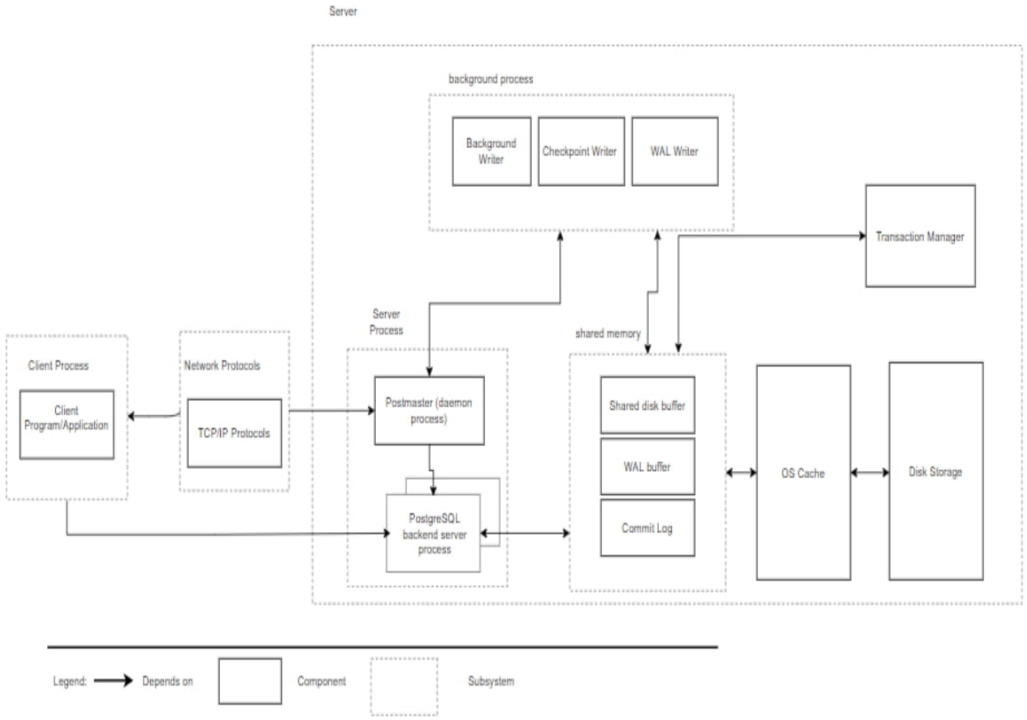

***Figure 6**: Conceptual Architecture of PostgreSQL[12]*

Evolution of PostgreSQL

Architecture Evolution

The evolution of DB architectures has been constantly evolving to address performance issues such as scalability. In the beginning use of databases (1960s), business applications consisted of assembly languages written in COBOL and FORTRAN. In terms of conceptual architecture, applications were monolithic which consisted of the user interface, application logic and CRUD operations. Afterwards, the monolithic style evolved into separation of applications called n-tier architecture that separates the user interface (Presentation tier), application server and programs that access the application logic (Application Tier) and the database (Data tier).

Furthermore, the introduction of RISC based UNIX computers precipitated the client server architecture, a model that allows for clients to access a resource known as servers. In the present time, the client-server model is a standard implementation with popular database management systems (DBMS) such as PostgreSQL.

Evolution of Object-Oriented features

PostgreSQL has primarily been an Object Relational Database Management System (ORDBMS), since it can support some object-oriented features. Some standout Object-Oriented (OO) features, like table inheritance and function overloading are available to the clients of the database system.

Furthermore, a potential instance of our architecture evolving can expand upon some of these critical OO functionality. An important OO feature not supported by the current architecture is the ability to define methods on datatypes. Potentially integrating similar OO features within our architecture would require extending the functionality of the sub-systems of the PostgreSQL server.

An example of this would be rewriting some of the modules within the Parser, Analyzer, Rewriter, and Executor to modify their interactions to support OO functionality and store further OO related data within the shared buffer for future access. More OO features can be added as the system evolves.

Current Performance Issues

Every bit of optimization or improvement in the evolution of historical database architectures focuses on the need to improve performance of the overall database system and the clients they support. The most important metric to measure performance is the overall runtime of each database transaction performed by the client of the system.

One common method of reducing runtime in most modern database management systems is using caching solutions. The idea that the most frequently used queries can be cached, either in memory or within the database itself, to reduce runtime and optimize performance can be outsourced to other domains within our conceptual architecture. We can forward this idea of caching in the context of creating database connections. In most cases, clients need to create, maintain, and close connections each time they make use of the database system. With PostgreSQL, each new connection can take up to 1.3MB in memory [1].

This amount of overhead each time a user of the DBMS tries to establish connection may cause serious deterioration of database runtime performance. On an enterprise level, in a production environment, where millions of these concurrent connections are created, the memory resources of the entire system may get exceeded, while causing fatal damage to important client data. We may apply the idea of caching to a further level in the future evolution of the architecture of the database system by the use of connection pooling [1].

We may overcome this runtime overhead of creating and closing connections for each individual client/user by caching these database connections that can be reused each time a client makes a future request to establish a database connection. We can add this mechanism within the architecture of our evolving system as a subtle way to improve the performance of our database system when it is being used during high traffic times at the enterprise level.

Potential support for future Query Languages

Most modern database architectures are centered and optimized for the Structured Query Language (SQL), under the relational database model. With the advancement of research, both in academia and industry, the need for faster database models and languages are imminent. Our database architecture needs to support such models or languages to reflect some of the needs of the ever changing industry.

One such example of this is the advent of the proposed Graph Query Language (GQL) [11]. Much like SQL, GQL is also intended to be a declarative database query language. As technologies evolve, the process of modeling data may become more refined requiring sophisticated representations. The graph model uses edge and vertex to represent data entities and relationships. Using this model may enable complex computations to be carried out by different modules within our database system.

In future instances of the system, the use of the potential GQL may be factored within our architecture. We may either extend our current parser, analyzer, and executor to compile bits of GQL code or by even keeping separate modular PostgreS servers, one for each type of query language. Keeping this extra

functionality in future instances of the system will allow it to be more versatile and move away from the ubiquitous relational model of representing data.

Developer Division of Responsibilities

PostgreSQL is one of the most popular open-source relational database systems. The PostgreSQL Database Developer is responsible for developing database code that would be used for cloud platforms. This is a central role for developers that will build effective APIs that access and manipulate very large datasets, many of them geospatial, working with product managers, front end developers and back end developers as needed. PostgreSQL is a client server system that uses a different tiered layered architecture which will be discussed in this report. With more than 30 years of development work, PostgreSQL has proven to be a highly reliable and robust database that can handle a large number of complicated data workloads. PostgreSQL is considered to be the primary open-source database choice when migrating from commercial databases such as Oracle. With PostgreSQL, you can create users and roles with granular access permissions which fall under the developer responsibilities. The new user or role must be selectively granted the required permissions for each database object. This gives a lot of power to the end user, but at the same time, it makes the process of creating users and roles with the correct permissions potentially complicated. However, there are some limitations and deprivation of this open-source software:

1. Each process creates a separate service for every client, which turns into a lot of memory utilization thus not making it efficient.
2. If we make a comparison, PostgreSQL is not good when it comes to performance.
3. It is not as popular as other database management systems.
4. This also has a lack of skilled professionals.
5. When it comes to speed, PostgreSQL is not worthy as compared to other tools.
6. Making replication is more complex.
7. Installation is not easy for the beginner.

Conclusion

PostgreSQL offers its users a huge (and growing) number of functions. These help programmers to create new applications, admins better protect data integrity, and developers build resilient and secure environments. PostgreSQL also gives its users the ability to manage data, regardless of how big and complex the database is. Additionally, development is extensible, and its many features like indexes define APIs, which help to build out with PostgreSQL web applications. It is scalable which helps the quantity of data that the user deals with is manageable. It can also accommodate a huge number of concurrent users. Many clients hire PostgreSQL developers to create strong web applications because it has many popular and great reputations for its reliability, architecture, data integrity, extensibility, robust feature set, and dedication to the open-source community. The software consistently delivers innovative solutions thanks to it being open source and the community being able to give feedback. This allows it to run on the major operating systems such asr Linux, Windows, OpenBSD, and FreeBSD and is a default database for the macOS Server which make it appealing to a lot of users. As mentioned, PostgreSQL is an open-source relational database that many organizations and people select because it has transactions with Consistency, Atomicity, Isolation, and Durability properties. It has automatically updatable views, triggers, materialized views, foreign keys as well as stored procedures and is compatible with handling different workloads, including single machines to data warehouses and web services with concurrent users making it a good option for users to use as a database management system. In summary It's easy to see why PostgreSQL is still gaining popularity and why developers embrace PostgreSQL to play a central role in the enterprise data center and together with the support of its active open-source community, ensure that it will remain a leader that is kept up to date as DBMS methods and technology stacks evolve.

Lesson Learned

One of the difficulties we encountered were conflicting sources of information. Due to the long-running nature of PostgreSQL's history, there are many sources of information that are inaccurate, simply because they are outdated. In the future, great care will be used to look at the most recent articles, and ones that list the version of software that is being referenced.

Another lesson learned during the making of this report is the challenge of having many people write one research paper. Going forward, greater care will be taken to adhere to formatting standards, keeping track of sources/citations, and consistent writing conventions.

Data Dictionary and Naming Conventions

Concurrency	The idea of doing multiple transactions at same time by different users. Data can be accessed at the same time by different users.
Transaction	This can be thought of as a change that took place in the database.
Multi-Version Control Concurrency (MVCC)	This is a mechanism for concurrency control which allows concurrent access to data, and makes sure that data remains consistent
Transmission Control Protocol (TCP)	A network protocol that allows devices and applications to send and receive messages over the internet.

Structured Query Language (SQL)	Specific Language that is used to access and perform actions on databases.
Database Management System (DBMS)	This is the software system that allows a user to perform actions on a database.
Atomicity, Consistency, Isolation and Durability (ACID)	A property in transactions that should be ideally satisfied to ensure validity of data
Graph Query Language (GQL)	An SQL like language for API's.
Write-Ahead Logging (WAL)	Method to ensure consistency and integrity of data

REFERENCES

[1]Aboagye, M. (2020, October 22). Improve database performance with connection pooling. Stack Overflow Blog. https://stackoverflow.blog/2020/10/14/improve-database-performance-with-connection-pooling/

[2]Harder, T. (2005). DBMS Architecture -- the Layer Model and its Evolution. Research Gate, 45–57.

[3]Kurose, J. F., & Ross, K. W. (2017). *Computer networking: A top-down approach*. Pearson.

[4]Ahmed, N. (n.d.). *Let's get back to basics - postgresql memory components*. Let's get back to basics - PostgreSQL Memory Components. Retrieved October 6, 2021, from https://www.postgresql.fastware.com/blog/back-to-basics-with-postgresql-memory-components.

[5]File browser. PostgreSQL. (n.d.). Retrieved October 6, 2021, from https://www.postgresql.org/ftp/source/v13.4/.

[6]Kushwaha, P. (2020, June 22). Architecture of PostgreSQL DB - The Startup. Medium. https://medium.com/swlh/architecture-of-postgresql-db-d6b1ac4cc231

[7]Maymala, J. (2015). PostgreSQL for Data Architects (1st ed.). Packt Publishing.

[8]PopSQL, Heard, A., Basilio, D., Berkok, E., Canella, J., Fischer, M., & Godfrey, M. (2010).

Conceptual Architecture of PostgreSQL.
https://cisc322.files.wordpress.com/2010/10/conceptual_architecture_of_postgresql.pdf

[9]Suzuki, H. (2012). *The Internals of PostgreSQL*. InterDB. Retrieved October 5, 2021, from https://www.interdb.jp/pg/

[10]University of California, Berkley, Stonebraker, M., Rowe, L., & Hirohama, M. (n.d.). *The Implementation of Postgres*. https://dsf.berkeley.edu/papers/ERL-M90-34.pdf

[11]Wikimedia Foundation. (2021, May 6). *Graph query language*. Wikipedia. Retrieved October 6, 2021, from https://en.wikipedia.org/wiki/Graph_Query_Language.

[12]Silberschatz, A. S. (n.d.). *BK - database system concepts*. Retrieved October 6, 2021, from https://www.db-book.com/db7/online-chapters-dir/32.pdf.