Conceptual Architecture of PostgreSQL

Madison Hartley
215062417

Alberto Mastrofrancesco
214107742

Arsh Punia
215199961

Ashish Ranjan
215580327

Richard Robinson
215353287

Daniel Santaguida
214790695

Doris Zhou
214746283

September 25, 2021

Abstract

PostgreSQL is a popular, open-source relational database management system. To understand the system's architecture, this paper dissects the system into interacting components and discusses how they interact to perform database-related tasks. Several architectural styles are discussed, and client-server architecture style has been determined to be the overall architecture. The detailed architecture discussion focuses on the system's client-server interface, query flow, memory management, and backend services. The paper details PostgreSQL's ability to handle concurrency, its architectural evolution, and how developers handle responsibilities. Two sequence diagrams are presented to outline the component interaction.

Contents

1 Introduction

PostgreSQL is a free, open-source relational database management and the backbone for many impactful software projects. To understand how PostgreSQL efficiently handles user queries and maintains its databases, this paper examines its architecture styles in detail based on conceptual designs.

The overall architecture of PostgreSQL is client-server. The architecture style is identified based on the clear separation of PostgreSQL's many available clients and its standalone server. The clients of PostgreSQL include its most popular command-line client psql, graphic user interface client pgAdmin, and web clients like phpPgAdmin. They may look and feel different, but they all connect to the PostgreSQL server to perform database services. The clients use TCP/IP protocol to connect to the server.

PostgreSQL does loosely resemble a layered architecture style, with the lowest layer being the database cluster and the highest layer being the clients. Given the interactions between components are more complex than layered architecture, this style does not accurately describe the general architecture of PostgreSQL. Some components of PostgreSQL use a pipe-and-filter architecture style, with the query execution process being the prominent example. However, It is observed that the overall architecture does not process streams of data, and there are no clearly defined independent operations. The memory management demonstrates repository style, with multiple processes accessing the same shared memory. However, the overall architecture is more complex. After analyzing all possible alternatives, it is decided that the overall architecture more prominently reflects that of client-server, though some components use other architecture styles.

As mentioned in the previous paragraph, PostgreSQL's front-end clients are diverse: graphic user interface, CLI tool, or web servers. Some tools are updated with PostgreSQL, but many are developed by users. These clients connect to the PostgreSQL database using TCP/IP protocol and send connection or query requests to the server.

The PostgreSQL server has three types of processes: the Postgres server process, backend process, and backend services. The server process handles all the requests and fork backend processes to handle query execution, thus achieving concurrency. As these backend processes are forked, the clients will no longer talk to the server process. From there, the backend process transforms the SQL text using five subsystems Parser, Analyzer, Rewriter, Planner, and Executer. To execute a query, the backend process converts the SQL text to a raw parse tree and query tree, then executes the statement by reading and writing the table and index in the database cluster by utilizing the buffer manager. The backend process uses a pipe-and-filter architecture, by passing the text along these subsystems. The backend services handle background activities such as writing to log files, writing metadata, and collecting statistical information.

The memory management system includes local and shared memory, with buffer manager help transferring memory to permanent storage. The local memory belongs to a single process, while shared memory is created by the server process for all processes to share. The buffer manager is not only responsible for transferring shared memory to permanent storage.

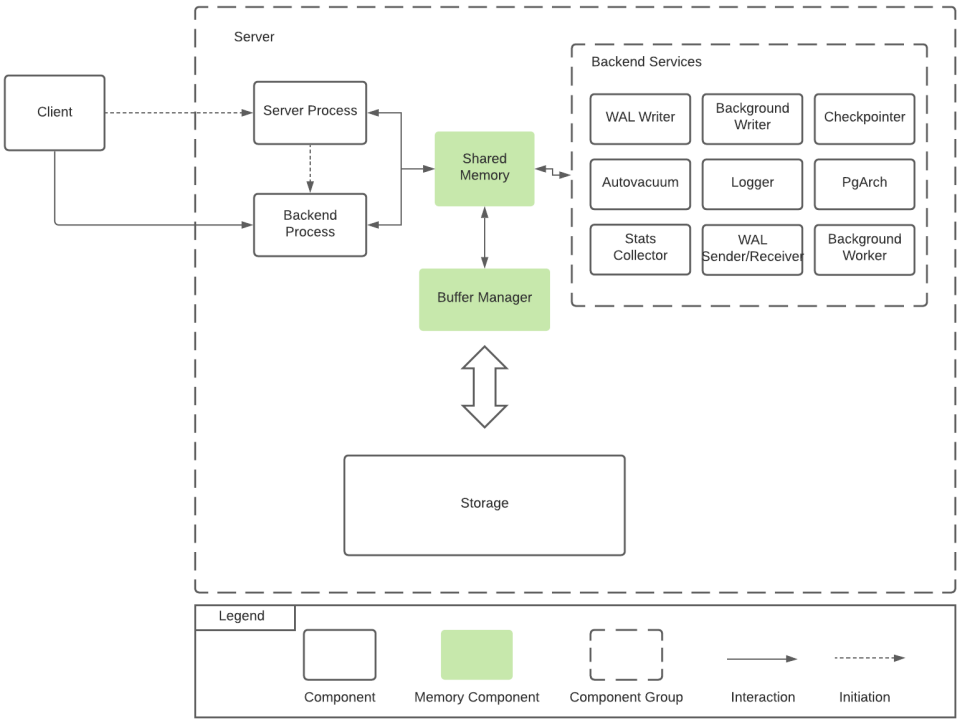

Figure 1: The high-level architecture of PostgreSQL

It also reads missing pages from the database cluster when processes request them. The reading and writing of the database cluster resembles a hybrid of repository architecture and some elements of the pipeline style. All backend processes send requests to the buffer manager to read a page. The buffer manager will load the page if the page does not already exist in the buffer manager.

Database cluster contains all the databases managed by PostgreSQL. The databases themselves logically are database objects. Belonging to the databases are the indexes, tables, and other objects. Physically, the database cluster is a base directory, and each database is created as a sub-directory.

2 System Overview

The architecture of a PostgreSQL instance can be seen as a collection of smaller components that interact together to deliver the overall functionality. For the purposes of this assignment, the interacting components can be classified under the following subsections:

1. Client-Server Interface
2. Query Management
3. Memory Management
4. Backend Processes

2.1 Client-Server Interface

The client-server interface subsystem allows the clients to connect to and interact with the server. From a conceptual architecture perspective, this job is the responsibility of the postmaster and postgres processes. The `postmaster` is the first process to be instantiated whenever the PostgreSQL server is started. It can be understood as the "driver" process that orchestrates the creation and interaction of all other backend processes (including the `postgres` process).

For clients interacting with the server, control is passed either to the `Postmaster` module, or directly to the Postgres entry point within the `tcop`. The `Postmaster` module waits for new connection requests and starts a Postgres process only upon a successful connection. Any subsequent queries from a client session go through its respective Postgres process handler pipeline (which, as mentioned before, resides in the `tcop` module) to the appropriate module and function as described in the "Query Management" section. Please note that each client session has an associated `postgres` process.

2.2 Query Management

Query management pertains to the parsing, optimization, and execution of queries that are issued by the client programs. From a conceptual standpoint, query management can be understood as an amalgamation of the following functions:

1. **Parser.** The `tcop` module forwards incoming client queries (for existing sessions) to the parser. The parser is then responsible for checking the syntax, identifying the query type, building the query tree, and adding command-specific structures to be used in later stages.
2. **Traffic Cop.** Once the queries have been successfully parsed, the Traffic Cop routes them to the appropriate function as described below:
   (a) Utility commands such as `CREATE TABLE` or `COPY` that do not require complex handling are forwarded to the commands function.
   (b) Non-utility queries that have a cached execution plan are subjected to some syntax and semantic checks after which they are sent directly for execution (soft-parse). A soft-parse helps avoid a computationally intensive optimization process.
   (c) Non-utility queries that do NOT have an execution plan are sent to the rewriter for additional processing and optimization before execution.
3. **Rewriter.** The rewriter modifies the parsed query to account for any applicable rules and sends the rewritten query trees to the planner.
4. **Planner/Optimizer.** The planner and optimizer work to create an optimal execution plan for the rewritten query. This is done by arranging the query components as a tree and deriving an optimal path from that. That optimal path is converted into an execution plan and forwarded to the executor.

5. **Executor.** The executor processes the execution plan to extract and update the required set of rows. The possible operations here are handled using heap scans, index scans, sorting, joins, etc.

6. **Commands.** The commands module exists primarily to address the SQL commands that do not require complex handling

2.3 Memory Management

PostgreSQL relies on a two-pronged approach for memory management. First, there is the shared memory that is used by nearly all the backend processes. Second, some processes also have their own chunk of memory known as process memory or local memory.

There are four types of local memory:

1. **Temporary buffers.** Memory that is used to store and access temporary tables

2. **Maintenance work memory.** Memory that is used for maintenance and housekeeping operations

3. **Working Memory.** Memory that is used by the executor for "ORDER BY" and "DISTINCT" operations

4. **Vacuum Buffer.** Memory that is used by the Autovacuum worker processes. For more information on the Autovacuum process, please refer to the Backend Processes section.

Subsequently, there also are four types of shared memory:

1. **Shared Buffers.** Any modified ("dirty") database blocks or pages are stored inside the shared buffers such that they can be accessed by all backend processes.

2. **WAL Buffers.** WAL stands for Write Ahead Log. These logs record the various modifications to the data BEFORE that modification is applied to the actual data file. In other words, the WAL contains metadata information about changes to the actual data. The WAL buffers are used to store the relevant WAL data.

3. **CLOG Buffers.** CLOG stands for "Commit Logs". These logs maintain information about commit status of database transactions. Consequently, the CLOG buffers are leveraged to hold commit log pages.

4. **Memory for Locks.** This memory is used to maintain the locks that are used in and shared across the PostgreSQL instance.

2.4 Backend Services

Any PostgreSQL instance is supported by multiple backend modules that are responsible for operations like vacuuming, logging, Disk I/O, etc. All backend services are forked from the postmaster process and are processes in their own right. The backend processes are given below:

1. **WAL writer processes.** As mentioned before, the WAL logs contain the metadata information about changes to the actual data. The WAL writer process is responsible for periodically writing the in-memory WAL files to the disk. WAL Logs are instrumental in enabling database recovery in the event of a crash or restart. A periodic "push" to the disk also leads to efficiency gains by reducing disk I/O.

2. **Background writer process.** This process writes the dirty pages in the "shared buffers" to the disk. The "dirty pages" here refer to modified database blocks. Just like with the WAL writer process, there is a speedup resulting from reduced disk I/O operations. Furthermore, this process also aids efficiency by reducing the number of I/O operations done during the checkpoints.

3. **Checkpointer process.** Checkpoints are transaction points that mark the flushing of all WAL and dirty shared buffer data to the disk. In case of a database crash, the database recovery utility can look at the last checkpoint and start the REDO operation from there. Any changes made to data files before that recovery checkpoint are guaranteed to be already on disk.

4. **Autovacuum process.** In the event of an `UPDATE` or `DELETE` operation in PostgreSQL, the data is not immediately altered, instead the "older version" is retained to support any other simultaneous or outstanding operations. Once those older versions (also known as dead tuples) are no longer in use, the Autovacuum process swoops in and cleans them out to make room in the memory.

5. **Logger process.** As the name suggests, this process writes the instance information to the log files.

6. **PgArch Process.** In simple terms, the PgArch process archives the WAL logs. Because the WAL logs are recycled, the PgArch process backs the existing WAL log up before overwriting it with more recent metadata. The archived WAL logs (that are now present on the disk) can then be leveraged for snapshot recovery of databases.

7. **Stats collector.** The stats collector is responsible for collecting statistical information from other backend processes. This information can include things like operational load, usage statistics, and number of operations. The information collected by the stats collector can be leveraged by other services such as Autovacuum to enumerate (and then delete) dead tuples.

8. **WAL sender/receiver process.** These two services are used in case streaming replication is enabled. Streaming replication allows you to ship the WAL logs records to some standby servers [Streaming Replication] as a fail safe. Consequently, the sender and receiver processes facilitate the transfer of WAL logs to these standby servers.

9. **Background worker process.** If a query gets parallelized across several CPUs, the various queries are run as separate processes known as background worker processes.

3 Use Cases

When the client sends a statement to Postgres, it is identified as simple or complex. Figure 2 illustrates a simple statement, which in this case is creating a database index. A database index helps to speed up querying of certain data in a table. When such a statement is delivered to the server, the Postmaster module first handles it. It then creates a Postgres instance, and passes control to the Traffic Cop. This is the main pipe for the rest of the sequence. After preliminary parsing of the statement, the index is defined using the Commands module. The command to create an index in turn relies on the lower-level Catalog module, which is responsible for actually creating the index. [1]

Figure 3 illustrates one of the most common "complex" statements, a `SELECT` query, which returns all rows in a given table matching the predicate, if any. Again, Postmaster handles the incoming request, and after creating a Postgres instance, passes control to the Traffic Cop. Here, several filters occur; the `SELECT` statement is first parsed by the Parser module, which returns parse trees. It then gets processed by the Rewrite module, which applies any rules, if any. The Optimizer module then creates a plan for the execution of the statement, and the Executor module then finally executes the query and returns the selected rows back to the client. [1]

4 Evolution of the System

Before Postgres was Ingres, which was conceived in the 1970s. Ingres' predecessor is a relational database management system (henceforth RDBMS) that utilised B-trees and applied triggers for integrity checks. Michael Stonebraker and Eugene Wong began research on relational databases. [6] They worked on INGRES until the 90's, when they began developing "post-Ingres", or, POSTGRES. [5] In 1995, Andrew Yu and Jolly Chen wrote an SQL interpreter for POSTGRES, leading to the creation of PostgreSQL in 1996. [8]

The architecture began as a way to store data independently of the user's application code by storing data in a table and assigning keys, and accessing the data via high-level programming language. This works by not predefining the data types and instead leaving that to the user. The relational database management system integrated support with Oracle using ENTERPRISEDB in 1997, and continued to grow from there. Support was later added for WAL, checkpoints, advisory locks, and many more updates over time to improve efficiency. Recent updates from the past few years include adding inserted data triggering autovacuum, parallelized vacuuming of indices, and enhanced query planning. Version 10 brought improved monitoring and control, and version 11 brought index creation being handled as a parallel process, JIT compilation, and HTTPS support. [7]

5 Control and Data Flow

Control and data flows between four major subsystems. The functionality of these systems have been described in the interacting components section of this report. The following

section will outline specific features,actions or processes that result in a change in control or data flow within these systems.

The client server interface subsystem gives clients the ability to connect to the server. This is done through the Postmaster service, which is the first process to start in PostgreSQL. The data flows from the client to the server and specifically Postmaster service or Traffic Cop module.

Data flows and control flows inside various sub components in the query manager. Control starts at the Parser, flows to the Traffic Cop module, Rewriter, Planner/Optimizer, and finally the Executor. Data flow can shift based on the routing of the traffic cop module and the specifics of the query.

As mentioned in previous sections, PostgreSQL has two categories of memory: shared memory and local memory, Share memory is shared across multiple processes. Data is constantly flowing between shared memory and back end processes. Local memory is memory specific to a process or a function and data flows when this process accesses memory.

The data and control flow in backend services is structured into multiple sub components much like the query manager. These processes are interacting with the disk, shared memory and their own local memory storage. Data is constantly flowing when these processes are collecting statistical information, writing log files or cleaning out memory.

6 Concurrency

PostgreSQL supports concurrency through its various levels of transaction isolation, as well as its explicit locking. PostgreSQL uses Multiversion Concurrency Control, where database snapshots are taken either at the beginning of a transaction or at the beginning of a particular command. These database snapshots are generally updated as transactions are committed in order to preserve data consistency in concurrent transactions. The way these snapshots are utilised are dependent on the particular level of transaction isolation being used for a transaction.

Transaction isolation levels exist in PostgreSQL to give users the option to read or write from the database with different levels of restrictions. The different levels of transaction isolation that exist within PostgreSQL are read committed, repeatable read, and serializable. Each of these levels have different requirements for transactions to succeed. However, each of these levels also introduce their own risk of encountering various phenomena.

The following phenomena may be encountered depending on the level of transaction isolation used: dirty read, non repeatable read, phantom read, and serialization anomaly. Dirty reads occur when data is read from uncommitted writes from other transactions. Non repeatable reads occur when the second read of a piece of data produces different results than the first read. A phantom read is when rows that satisfy a particular queries conditions change because of a concurrent transaction that had just been committed. Finally, a serialization anomaly is when a group of transactions can be committed, but any orderings of the same transaction produces a different result.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Table 1: Transaction Isolation Levels [3]

Each of the transaction isolation levels behave differently. The first level is the read committed isolation level, which is the default isolation level for a transaction. When a `SELECT` query is used at this level, a snapshot of the database is taken before its execution. If there were previous writes in the same transaction as the `SELECT` query, the database snapshot will include those writes, even though they are uncommitted. Write commands behave similarly to `SELECT` queries, however, behave differently when other write commands are executing concurrently. If a concurrent write command is rolled back, then the original write command will continue as normal. If a concurrent write command is successfully committed, the original write command will re-check the rows which initially satisfied its requirements, and perform its updates on the new rows.

The second level of transaction isolation is the repeatable read level. `SELECT` queries behave similarly to those in the read committed isolation level, however, in the repeatable read level, multiple `SELECT` queries in a row will use the same database snapshot instead of taking a new snapshot at the beginning of each query. Write commands behaviour depends on concurrent write commands. If a concurrent write command is rolled back, the original write command will continue as normal. However, if a concurrent write command is successfully committed, the original write command will fail and produce an error referencing concurrent updates.

The final and strictest transaction isolation level is the serializable isolation level. Transactions at the serializable isolation level behave the same as transactions at the repeatable read isolation level. The difference between the two levels is that the serializable isolation level includes a check for transactions that behave inconsistently from all possible executions of the same transaction, but executed serially. Serializable transactions also make use of predicate locks, which is used on updated tables where previous read commands may have been affected.

In some situations, MVCC may not produce wanted results. To assist MVCC, PostgreSQL utilizes table-level locks, row-level locks, and advisory locks. Table-level locks are automatically used by PostgreSQL. This type of lock is used to lock tables. This lock is typically held until the end of a transaction, or until a rollback. Some locks cannot be held by more than one transaction, and these locks conflict with other locks of the same type.

The second type of locks are row-level locks. Row level locks are also used automatically by

Requested Lock Mode	Access Share	Row Share	Row Exclusive	Share Update Exclusive	Share	Share Row Exclusive	Exclusive	Access Exclusive
Access Share								X
Row Share							X	X
Row Exclusive					X	X	X	X
Share Update Exclusive				X	X	X	X	X
Share			X	X		X	X	X
Share Row Exclusive			X	X	X	X	X	X
Exclusive		X	X	X	X	X	X	X
Access Exclusive	X	X	X	X	X	X	X	X

Table 2: Conflicting Lock Modes [3]

PostgreSQL. They are very similar to table-level locks as their only difference is that they affect specific rows instead of tables.

Requested Lock Mode	For Key Share	For Share	For No Key Update	For Update
For Key Share				X
For Share			X	X
For No Key Update		X	X	X
For Update	X	X	X	X

Table 3: Conflicting Row-Level Locks [3]

The final type of locks are advisory locks. Advisory locks are not automatically used by PostgreSQL, and are typically applied manually. They can either be held at the connection level or at the transaction level. If held at the connection level, they are released either manually, or when the connection ends. If they are held at the transaction level, they are automatically released at the end of a transaction.

7 Division of Responsibilities

PostgreSQL is a very large database system that has been operational for over 30 years. This system provides a variety of well-built features to aid in maintaining and setting up any scale database. It is still being actively innovated, and further developments are being modified and added. Taking into account the magnitude of PostgreSQL, there are many developers

working to sustain and maintain this system. The many developers all have to work with the same architecture of PostgreSQL, this can result in many different factors being cautioned.

In the case of the front-end implementation, many companies and services use PostgreSQL in a vast majority of different ways. The system must therefore be able to be flexible and adapt to the different tasks/uses it is required for.

On the server side of development, developers must ensure that the system can be flexible and adapted to any variable provided/given. When making sure everything operates and will be able to interact with other components, the whole architecture of this PostgreSQL system must be in mind of the developers.

In the case of development, back-end, different developers have the ability to work on different modules, hence division of responsibility. PostgreSQL's developers can work on the vast majority of modules in the architecture simultaneously. The most common component that a developer can work on, and explained thoroughly above, is in regard to Query Flow. Query Flow contains modules that relate to the processing and execution of queries. This includes: Traffic Cop, Parser, Rewrite, Planner/Optimizer, Executor. All these listed under Query Flow are defined above under the heading "Query Management".

Responsibilities can be divided among each developer. In one instance, developers can and may work on different modules, for example one developer might be working on the Parser while two others work on the Optimizer and Executor, respectively. The tasks divided by the developers can also be devised based on the subsections listed above in section 2, Client-server Interface, Query Management, Memory Management and Backend Processes. In order to achieve overall functionality these components must interact and adapt to each other.

8 Conclusion

Though the overall architecture of PostgreSQL is client-server, it makes use of many different architecture styles such as pipe-and-filter and repository. This design decision illustrates that architecture design is a complex process and the perfect architecture style does not exist. Many modern software systems use a combination of different architecture styles.

Through the collaboration process, we learned that it is important to communicate. Though we all dealt with different parts of the report and assignments, we had to reach a mutual agreement on different technical terms and the overall architecture. The effective communication among us is what helped us complete this assignment.

A Sequence Diagrams

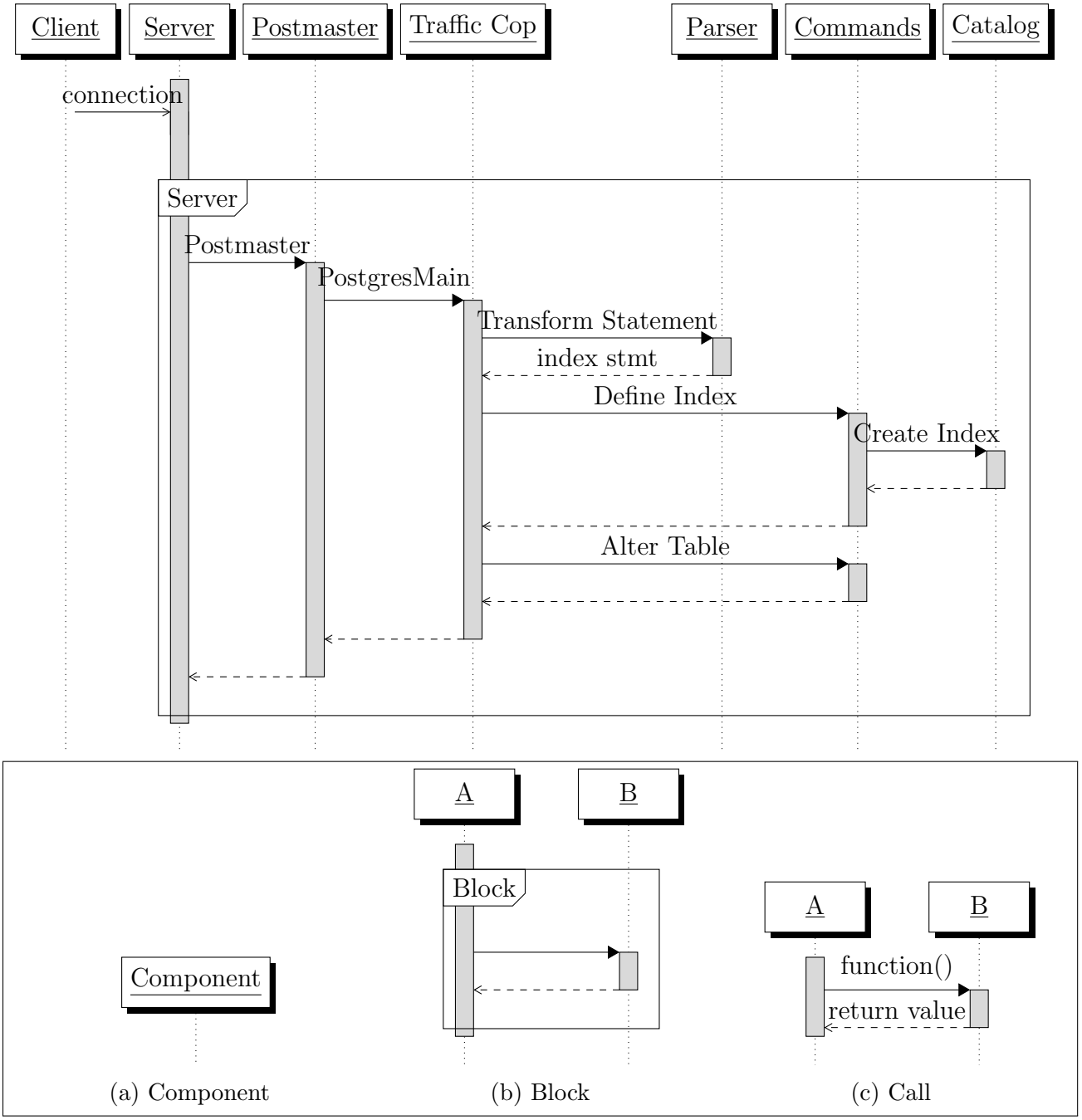

Figure 2: Sequence diagram describing the high level flow of creating an index in PostgreSQL, with legend.

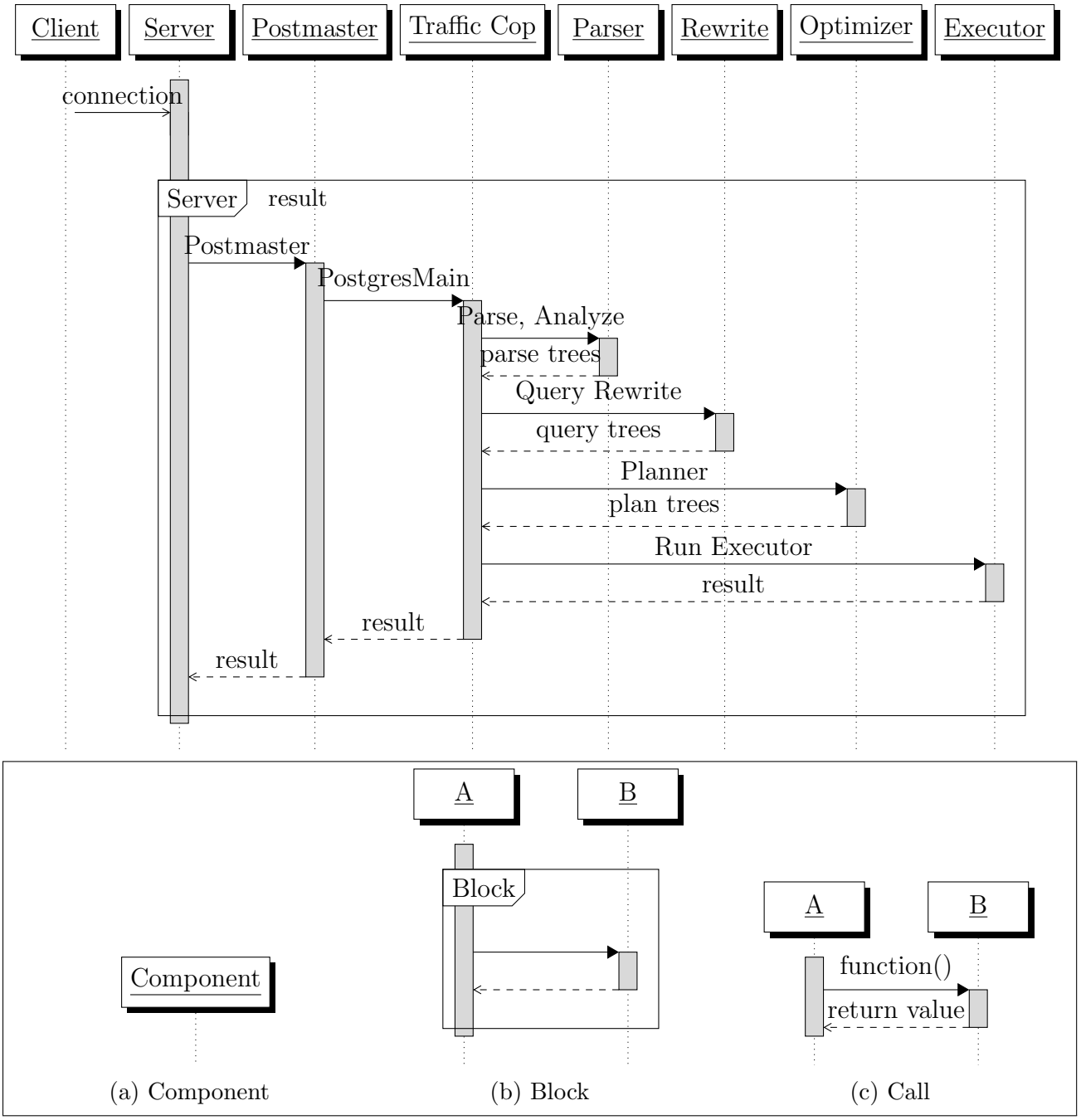

Figure 3: Sequence diagram describing the high level flow of executing a `SELECT` query, with legend.

References

[1] PostgreSQL. The PostgreSQL Global Development Group.

[2] H. Suzuki, "The Internals of PostgreSQL." [Online]. Available: https://www.interdb.jp/pg/index.html. [Accessed: 01-Oct-2021].

[3] "PostgreSQL: Documentation," PostgreSQL. [Online]. Available: https://www.postgresql.org/docs/. [Accessed: 20-Sep-2021].

[4] "Backend flowchart," PostgreSQL, 05-Sep-2012. [Online]. Available: https://www.postgresql.org/developer/backend/. [Accessed: 30-Sep-2021].

[5] "A Brief History of PostgreSQL," PostgreSQL, 24-July-2014. [Online]. Available: https://www.postgresql.org/docs/8.4/history.html. [Accessed: 30-Sep-2021].

[6] T. Haigh, "Michael Stonebraker," A.M. Turing Award Laureates, 2014. [Online]. Available: https://amturing.acm.org/award_winners/stonebraker_1172121.cfm. [Accessed: 06-Oct-2021].

[7] "Feature Matrix," PostgreSQL. [Online]. Available: https://www.postgresql.org/about/featurematrix. [Accessed: 30-Sep-2021].

[8] "EDB. power to postgres," EDB. [Online]. Available: https://www.enterprisedb.com/. [Accessed: 06-Oct-2021].