Team C-Lion's Assignment 2 Report

The Concrete Architecture of PostgreSQL

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

Lassonde, York University

EECS 4314: Advanced Software Engineering

Professor Zhen Ming (Jack) Jiang

November 08, 2021

Table of contents

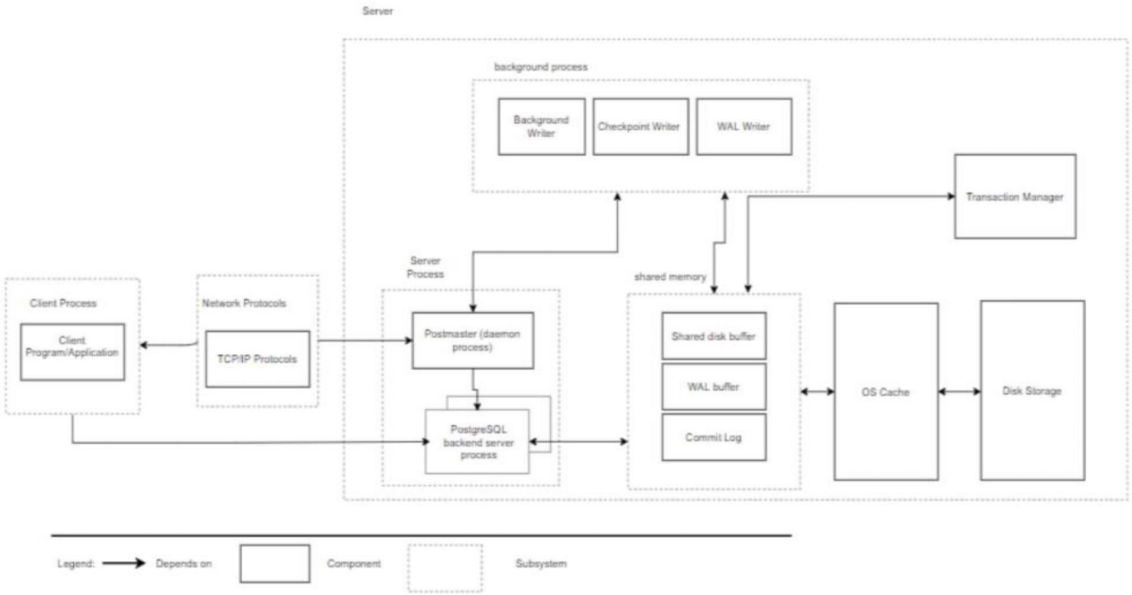
Abstract

In this project the concrete architecture of PostgreSQL was derived. Through sorting source files to fit within the conceptual architecture that had already been formed (see Assignment One) it was realized that the conceptual architecture needed to be remodeled slightly and all the source files were sorted into corresponding subsystems and shown in screenshot using LSEdit. In this report we will discuss the overview of the architecture with detailed description, subsystems and their interactions Important Architecture Style and Design Patterns and the conceptual vs concrete architecture of shared memory.

Introduction and Overview

PostgreSQL is a database editing software system divided into four major subsystems: client communications manager, server processes, database control, and shared utilities. The initial conceptual architecture showed the first three systems as three layers of a layered architecture. Under this view, shared utilities become a layered independent system. After concrete dependencies were shown and justified, it became apparent that the four major systems Interacted as independent objects in an object-oriented architecture.

The server processes consist of a postmaster and backend instances. Backend instances in turn contain five subsystems: postgres, parser, rewriter, planner/optimizer, and executor. The backend instances were initially thought to run as a pipe and filter, but analysis of the concrete architecture, as it was created, showed that these subsystems acted as objects in an object-oriented architecture. The Postgres subsystem often acts as a mediator between the backend instance and outside interaction but is occasionally bypassed by subsystems which access the parser directly.



Architecture

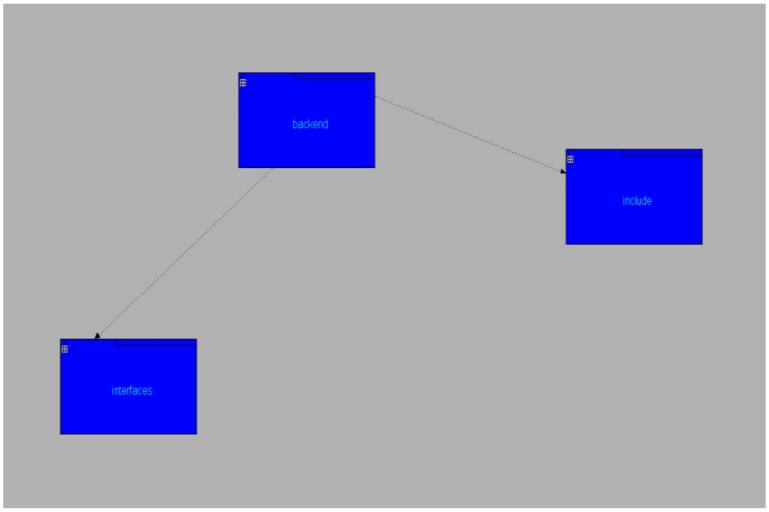


Figure 1 - Conceptual and Concrete architecture of the Client-Server Subsystem

Detail Descriptions of high-level architecture

The client program (e.g psql) initially interacts with interface libraries (libpq, ecpg). Libpq is a C application that has a set of library functions that permits the ability for a user to pass queries to PostgreSQL backend server and retrieve the results of the inputted queries. Initially, libpq makes a connection to the PostgreSQL backend and the libpq application can have several backend connections open at a time. To make a connection, libpq takes parameters such as host, port, username, and password. When a client undergoes successful authentication for a database, the inputted SQL query is processed through the ECPG preprocessor where SQL code is replaced with the ECPG library which replaces the SQL into C code to be processed by the C compiler. For the entire above process to occur, within PostgreSQL there is an interface component where header files are used throughout the source code in .c files which means that the header files are added in the .c files. Ultimately, this creates a chain of dependencies in the subsystems shown above in concrete high level architecture diagrams. Overall, libpq is very paramount for the TCP/IP communication with the database server establishing the architectural style of the client-server model. Once a request is authenticated the query is sent to the backend subsystem. The backend has various subsystems and some of the important ones include postmaster, access, storage etc.

From the perspective of an SQL query (e.g., a SELECT statement), once authentication is done as indicated above the interface library passes the query to the backend. The Postmaster manages events such as system shutdown; however, Postmaster fork a subprocess called Postgres which manages the events within the server. For instance, within Postgres backend server process, there is a module called tcop which dispatches requests to other components such as the query processor: optimizer, executor, parser, and rewriter. The next sequential subsystem occurs from the query processor to the buffer manager. If the results of this query are found in the shared buffer then the results are returned to the client via the backend process through the interface library. If the data is not found in a shared buffer cache, then OS cache can next be queried to check if the relevant data/file lies in OS cache. If so the results are again returned to the client via the backend process. If

data/file isn't present in either of these caches then this will result in an IO operation from the physical disk and will take longer as well in this case. In terms of the effectiveness, the ability for PostgreSQL to establish running multiple servers is pertinent for its use. The subsystem involved with scalability uses the architectural style of publish and subscribe model to where in PostgreSQL, the subscriber is the read-only application that has a single subscription; however, the application can go to other publishers to access data.

Subsystems and their interactions

Backend

In the client-server model, the backend is an important subsystem as it contains additional subsystems which are of high importance as can be seen from the concrete architecture diagram screenshots. Most of the subsystems in the backend are related to processes which are done once a client is authenticated. Some notable subsystems in backend include the client interface libraries, postmaster, query processing logic, the logic that deals with front end requests, log/recovery mechanisms and storage mechanisms.

Interfaces

The interface subsystem contains the two front end libraries libpq and ecpg. The purpose of these libraries is to do client authentication. Once the authentication is done the query can be sent to the backend part.

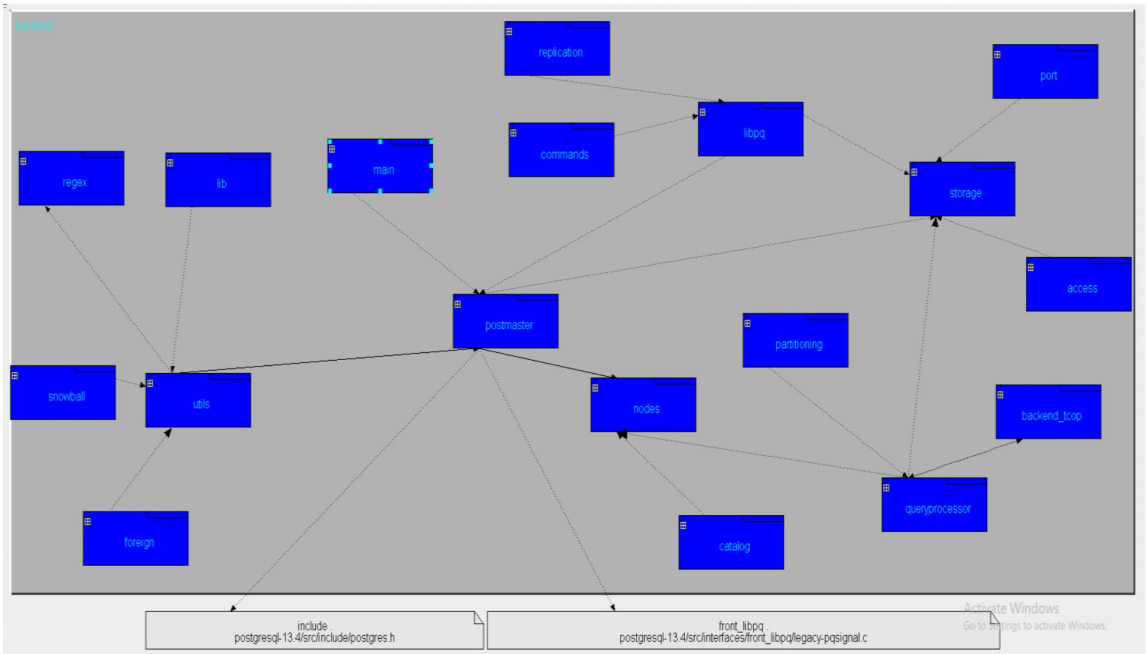


Figure 2 - High level concrete architecture of backend server

Libpq

The libpq subsystem is contained in the backend subsystem and contains the logic for lower level communication details between the backend and frontend. This subsystem also includes ways to format/parse frontend and backend messages. Libpq is dependent on the files in the tcop subsystem.

Postmaster

This subsystem is a very crucial subsystem in the backend. The requests come from the libpq which communicates over the TCP/IP network with the postmaster thus spawning a backend process. The postmaster process is responsible for initializing semaphore pools and shared memory. In the case of an individual backend crash, shared memory can be reset by the postmaster. Check pointer and WalWriter are two important processes that are also managed by the postmaster process (i.e., startup if necessary and termination). The postmaster initiates a dispatch to check pointer after a certain time whereby it checkpoints uncommitted data into a log file assisting in the recovery process.

Query Processor

The Query Processor subsystem as the name suggests is used to process a query that is passed to the backend. The Query Subsystem includes additional subsystems inside such as a parser which within has an analyzer component, optimizer, executor, and rewrite. The parser converts a plain text to parse trees. Analyzer transforms the raw parse tree into a query tree. Optimizer generates the cheapest path regarding the desired relation and passes it to the executor. The executor recursively performs the associated operations at each node of the plan tree. The plan tree is essentially a demand-pull pipeline of tuple processing operations [5].

Tcop

Tcop subsystem contains the logic to deal with function requests that are coming from the front end. This subsystem is a traffic related system where it can communicate with the front end and tell it that it is ready for a brand-new query if it is idle. The query string supplied by the client is passed to the parser component and upon completion the tcop subsystem further separates the utility commands such as ALTER, CREATE and DROP. Furthermore, the subsystem also includes logic related to handling errors (e.g., if client connection is lost etc) and includes the logic for query processing (rewrite, executor, optimizer).

Access

Access subsystem is responsible for accessing the tables of the database in an efficient manner. Various indexing methods/ways have been defined to make sure that indexing in the huge tables is fast. The insertion in tables also happens in the access subsystem. The WAL Log and Commit log are a very important part of the Access subsystem as WAL data can be used to perform recovery operations whereas Commit Log is a log of the transaction IDs of all transactions that are successfully completed.

Storage

PostgreSQL storage is a collection of modules that provide transaction management and access to database objects. It contains logic for buffer initialization, data structures for buffer management, shared buffers, locks, pinning/unpinning buffers, and local buffers. The design of these modules is guided by the goals of providing transaction management without specialized code, accommodating historical state of the database,

and taking advantage of hardware to improve performance. Following these points, the storage subsystem provides benefits including instantaneous recovery from crashes, ability to keep archival records, asynchronous housekeeping, and concurrency control.

Architecture Derivation Process

The derivation process for creating the concrete architecture required understanding the PostgreSQL source code for all the known subsystems from prior knowledge. After getting familiar with the source code, a python script was used to parse Postgres_UnderstandingFileDepedency.csv to find out all the dependencies regarding a specific subsystem. The figure below displays the dependencies regarding postmaster and determining the frequency of dependencies between a subsystem leads to further investigation into the architectural styles. Aside from the utility, access and command subsystems that have dependencies with many subsystems in the backend, as you can see that libpq has the second highest dependency with postmaster which can be explained between the relationship between the user application initializing the libpq to initially make a database connection.

```
('postgresql-13.4/src/backend/utils/hash', 5)
('postgresql-13.4/src/backend/access/heap', 2)
('postgresql-13.4/src/backend/libpq', 13)
('postgresql-13.4/src/backend/utils/adt', 8)
('postgresql-13.4/src/backend/bootstrap', 3)
('postgresql-13.4/src/backend/replication', 2)
('postgresql-13.4/src/backend/utils/time', 2)
('postgresql-13.4/src/backend/replication/logical', 3)
('postgresql-13.4/src/backend/access/table', 3)
('postgresql-13.4/src/backend/utils/misc', 22)
('postgresql-13.4/src/backend/utils', 5)
('postgresql-13.4/src/backend/storage/lmgr', 19)
('postgresql-13.4/src/backend/utils/fmgr', 2)
('postgresql-13.4/src/backend/utils/init', 20)
('postgresql-13.4/src/backend/utils/resowner', 4)
('postgresql-13.4/src/backend/commands', 2)
```

Figure 3 - diagram of the high-level concrete architecture

Furthermore, after carefully picking the important dependencies between subsystems, the files are initialized in the. contain file and raw.ta file (specifically showing the cLinks between the files) which is inputted into a createContainment.sh outputting the CustomizedFileDependencies.ls.ta. Lastly, the runLSEdit.sh is executed using the. ta file executing the LSEdit application of PostgreSQL. The editor is an interactive tool that allows the user to see the relationships between subsystem files. Once the assumptions between subsystems were confirmed using LSEdit, the included files of the subsystems were updated in the .contain. Afterwards, a python script was used to identify the differences between the existing. contain and new subsystem dependencies to include (e.g., backend), identified in the Postgres_UnderstandFileDependancy.csv which is then added in the raw.ta file. Overall, the process of constructing the contain and raw.ta file was iterative and manual as at times we had to make judgements based on subsystems/files. To avoid making unclear dependencies in LSEdit we had to show only relevant dependencies and had to cut out on many dependencies as initially it couldn't be interpreted.

Important Architecture Style and Design Patterns

PostgreSQL uses a variety of architecture styles and we have identified this from extensive research while also working on concrete architecture. It uses the Client server model when the interface library(libpq,

ecpg) connects with the backend. It uses a pipe and filter architecture inside query processing as can be seen from the diagram above. Replication subsystem uses the Master and Slave Design pattern as the master server sends data to the other standby servers which act as receivers of this data.

A brief overview of some of the most notably used design patterns in shared memory architecture are:

Static Allocation pattern: For some of the systems, which exhibit simple, consistent, and highly predictable sharing of the memory resources, the Static Allocation Pattern seems to be the best choice. The main advantage of this design pattern is that it is very simple to design and maintain in the long run. However, one drawback to this design pattern is that dynamic allocation of resources is not allowed. All allocation of memory resources is done at the system initialization stage.

Pool Allocation pattern: For many systems, objects are created dynamically during the system's execution. This design pattern addresses the need for dynamic allocation by creating pools of objects at startup, which later are accessible to processes upon request. This helps the system's memory resources to not get overwhelmed by repeated create and destroy function calls.

Garbage Collection pattern: One of the most common issues we come across are memory leaks which happen mostly due to pointer errors. The problem with memory leaks and pointer errors is that they leave no trace as to where the defect may be even after the entire system gets crashed. This is where the Garbage Collection pattern comes into play by addressing these memory defects. These memory leaks occur primarily due to errors made by the programmer.

Layered Style Pattern: The Buffer manager used this pattern to handle data travelling between the database cluster and shared memory

Assigned Subsystem - Shared Memory

Shared Buffer

Shared Buffer is the largest part of shared memory and is an area of memory where the database server can quickly read/write data. These buffers are used for the background processes and are also used for user processes who connect to the database. There is a parameter shared_buffers in PostgreSQL.conf file. This parameter determines the amount of memory that is dedicated to the server for caching purposes. As we know that reading data from memory is way faster than reading data from physical disk location thus this increases the performance significantly thereby reducing unnecessary IO operations. A list-like data structure is used to keep track of free buffers. Buffers that are free will get utilized by operations/processes and will eventually become stale in the pool. There is a flag named IO_IN_PROGRESS that is used in the buffer descriptor and is set when an IO begins. The flag is cleared once the IO operation finishes. This flag makes sure no process uses the buffer when an IO is being executed. There are also access control mechanisms that are in place to ensure data consistency and availability. Two important access control mechanisms for shared buffers are Pin counts and Buffer Content Locks.

Pin counts

Pins are holding on the buffer before anything can be done with the buffer. Pins are there so that no other process can interfere as long as the buffer is pinned. Unpinned buffers can be claimed and used by processes. The methods Read Buffer() and Release Buffer() can be used to pin a buffer while the process is in use and later can be used to unpin a buffer.

Buffer Content Locks

Buffer Content Locks are useful when reading the actual content of buffers. This lock is helpful in reading pointers to specific tuples. There are two kinds of locks i.e., shared and exclusive. As the name suggests multiple processes can be used to hold shared locks on the exact same buffer. In contrast an exclusive lock can be used to acquire a lock on a buffer and thus prevents anyone from holding exclusive and shared lock to this buffer. Exclusive lock can be used particularly when read/write happens. These locks shouldn't be held for very long because other processes might need to acquire the lock on a buffer as well for read/write purposes. The method Lock Buffer () can be used to acquire/release buffer content locks. If Pins are held, the tuples content can be accessed by a process even if the content locks are dropped after initially being held.

Write-Ahead Log Buffer

Write-Ahead Log Buffer aka as WAL buffer is an amount of memory allocation given to store WAL data [1]. This data contains the metadata info about the changes that take place in the actual storage disks. This data is very useful when doing database recovery. WAL buffer in terms of size is way smaller than the shared buffer and is not connected with the shared buffer directly. The way WAL buffers work is that first data is modified on these buffers. The modified data is called dirty data (data that is not written on disk). This dirty data is then written to a physical disk as WAL segments by the buffer manager. Hence as said before the biggest advantage of WAL is that if the operating system crashes or there is hardware failure the database can be recovered and reconstructed through these transactions that are recorded as log. There are two important subsystems that we need to talk about when it comes to WAL. These are check pointer and Background writer

Check pointer

Check pointer as the name suggests creates checkpoints in WAL [2]. A checkpoint means that all the info before this checkpoint has been updated [2]. At each checkpoint the dirty data is written to the disk storage and checkpoint information is written to the log file. Checkpoints are useful when performing a recovery operation. In the event if recovery is needed, the procedure of recovery looks at the latest checkpoint information to determine where the REDO operation needs to be performed [3].In terms of performance, check pointer can cause many I/O operations as data needs to be written to the disk which is resource intensive task. Due to the performance issue, check pointer is restricted in the sense that I/O operation happens at the start of a checkpoint and these operations complete before the start of next checkpoint to avoid overlap of intense operations at the same time. If no data is written in the WAL buffer since the last checkpoint, new checkpoints will be skipped. Two variables which are important for Checkpointer are checkpoint_timeout and max_wal_size. Checkpoint begins every checkpoint_timeout seconds or if max_wal_size exceeds its limit [3]. There is also an SQL command CHECKPOINT which can be used to do a checkpoint.

Background Writer

Background writer is another subsystem that helps the check pointer by writing data to the disk thus reducing the burden on the check pointer. The data that is written to the disk is the so-called "dirty" shared buffer. Once the data is written to the disk the buffers are marked as being clean. When a SQL query comes to a backend process it is best if clean buffers are found otherwise the process must also write dirty buffers to disk. One of the problematic cases is when data is being dirtied repeatedly in these shared buffers and Checkpointer, Background Writer are both writing this data to the disk which increases the net I/O operations.

Commit Log

The commit log is a log of transaction IDs of all transactions that successfully completed. Once a transaction has written its transaction ID to the commit log, that transaction is considered to have successfully completed. If a transaction fails for any reason (manually rolled back, power outage, etc.), its transaction ID is never written to the commit log. If a transaction is no longer running, Postgres can determine whether the query succeeded or failed by looking for its transaction ID in the commit log. If the transaction ID of the transaction is in the commit log, the transaction succeeded, otherwise the transaction failed.

Buffer Manager

The buffer manager follows a *layered style architecture* and is responsible for data travelling between the storage cluster and the shared memory processes. It is made up of three layers: the **buffer pool, the buffer descriptors layer,** and the **buffer table.** Each page of data is assigned a unique tag called the *buffer tag*, which is stored in the Buffer Table. Associated to each tag is a *buffer id*, which indicates the position of the buffer descriptors list in which the page's metadata is stored. Metadata is important as it not only identifies the page, but useful statistics like *refcount* (the number of times the page is actively referenced by the database), *usage count* (the number of times the page has been accessed). There are also useful *locks* that control access to that page, and *flags* that indicate whether the stored page is *dirty or valid*. The Buffer pool is an adjacent array in which the *buffer id* can also be used to find the page's storage location.[6]

Conceptual vs Concrete in PostgreSQL

In the concrete architecture, the relationships between subsystems are not as separated as the several main subsystems. For instance, utils provides utility applications to other components such as lib. Aside from the noticeable high-level dependency between the subsystems, the architectural styles of some of the highlighted subsystems are consistent with the styles referred to in conceptual architecture: in the below illustration is the dependency between all the subsystems in the query processor. Initially, tcop (which is managing the query processor) manages the dependency between parser and rewriter where SQL keywords are reserved for the optimizer to make the command efficient and eventually used as an input for the executor. Overall, the concrete architecture for the query processor defines a pipe and filter architecture style. Now, with regards to the storage subsystem in the concrete architecture of the backend it shows the clear high-level dependency between storage and other subsystems such as access and query processor. Initially using LsEditor, there are many dependencies that could not be shown in the report but the frequent one-way dependency to storage depicts the repository style elucidated in the conceptual arch

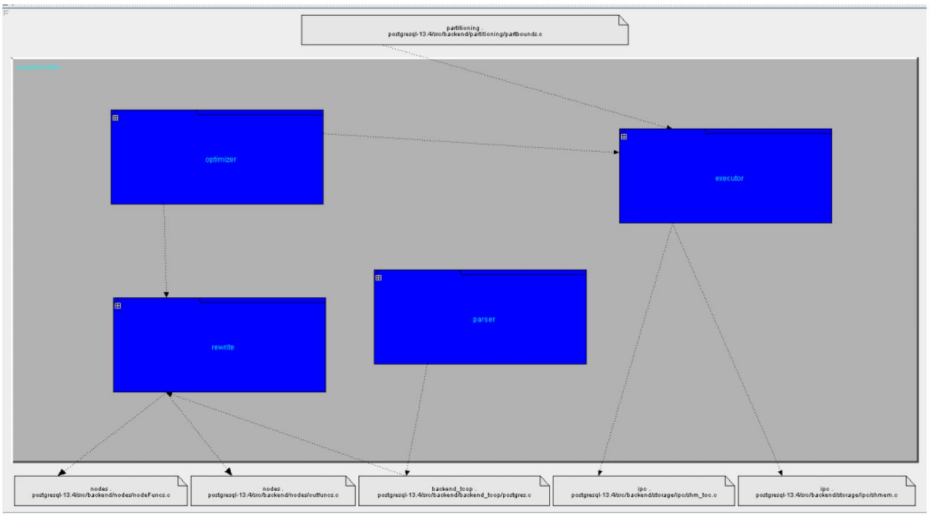


Figure 4 - Concrete Architecture of query processor

Shared Memory: Conceptual vs Concrete Viewpoint

From our conceptual architectural standpoint, the shared memory subsystem was considered as an additional option of storage, for our PSQL backends server, along with local allocated server memory. The local allocated memory sub-system is mainly tasked with the storage and buffer for query processing commands, while the shared memory subsystem was responsible for handling shared data between multiple clients.
In our conceptual architecture, the shared memory sub-system was only investigated briefly through Write Ahead Logging (WAL) and Shared Buffer Pool, both of which were seen as abstract areas of the shared memory serving as small functional components in the overall goal of the database system. Both WAL and Shared Buffer Pool from our conceptual architecture, alongside additional processes, were studied further in detail as these serve as core spaces with the shared memory functionality.

Role of Postmaster in shared memory architecture

The responsibility of the postmaster backend subsystem is explored further in context to shared memory. Even though the postmaster itself is not responsible for carrying out tasks related to shared memory, it plays a crucial role in forking new sub-processes which may be dynamically needed by the shared memory. In the concrete architecture, we explore how postmaster created the shared memory and semaphore pools during startup, but do not get involved in the interactions of these processes.

Exploration of the transactional nature of shared memory architecture

Transactions are a central part of the core responsibilities of the shared memory subsystem, and this very transactional nature is explored further in our concrete architecture. We explored the concrete application of common disk operations and the purpose of having commit logs to provide storage of records between each shared transaction in memory.

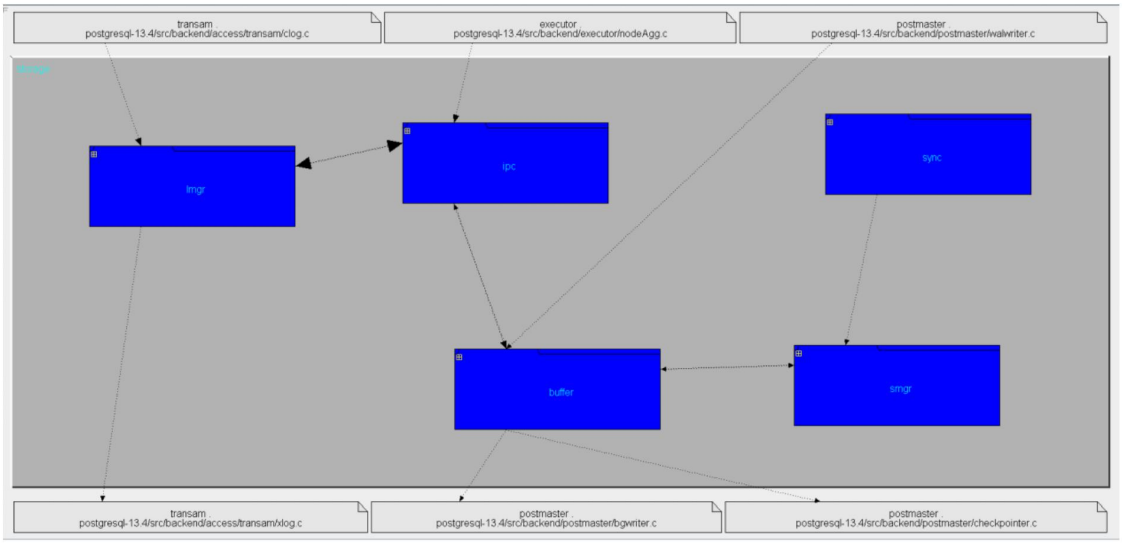


Figure 5 - Concrete Architecture of Storage Subsystem

Above diagram shows the concrete architecture of the storage subsystem. The subsystems such as ipc, sync was missed when we made the conceptual architecture in assignment 1. The buffer subsystem was also not fully explained properly in assignment 1 as well. Buffer inside has a reasonable number of dependencies which were not shown again in assignment 1 diagrams.

Diagrams:

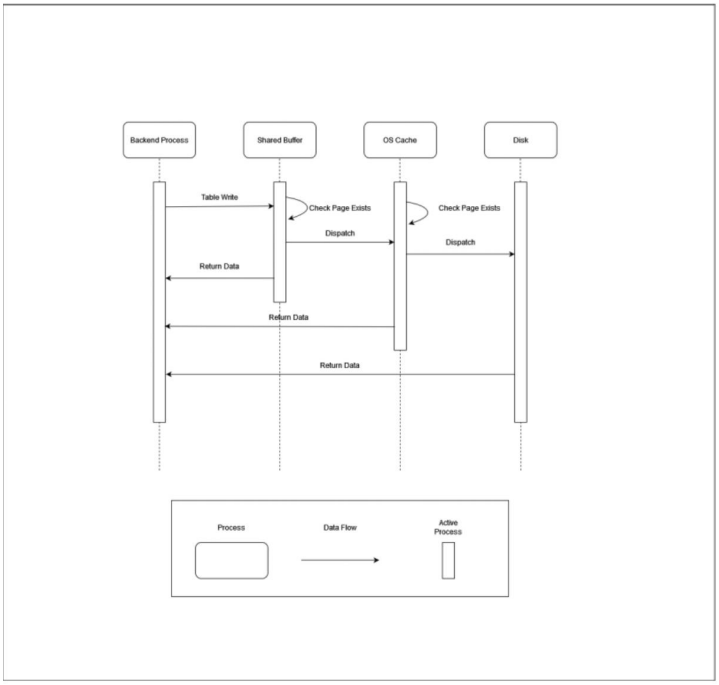


Figure 6 - Sequence of the shared buffer mechanism

The backend process makes a request to update a table. The Shared buffer unit looks for the requested table in its pages. If the page exists, the corresponding table is returned to the process. If the page does not exist, the same information is passed to the cache of the OS, where a similar search is performed. If the page exists, the table is returned to the background process, otherwise the request is passed on to the disk.

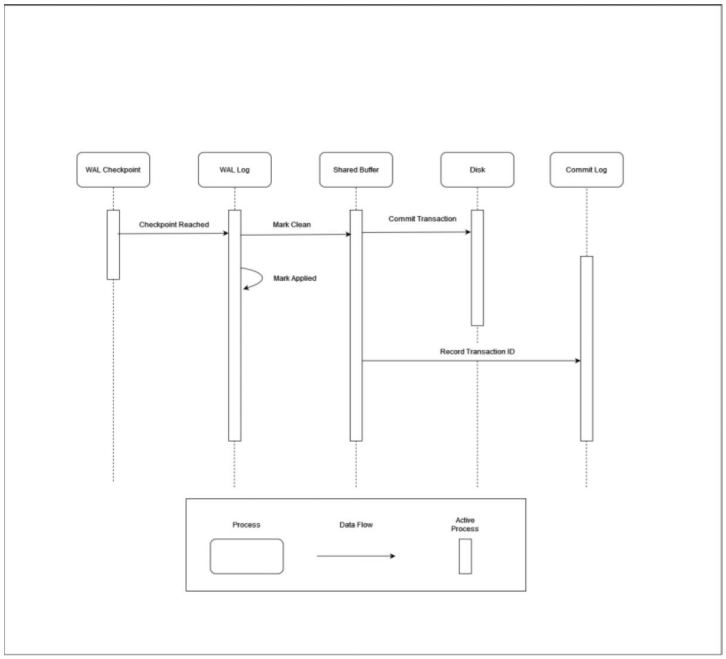


Figure 7 - Sequence of the WAL component

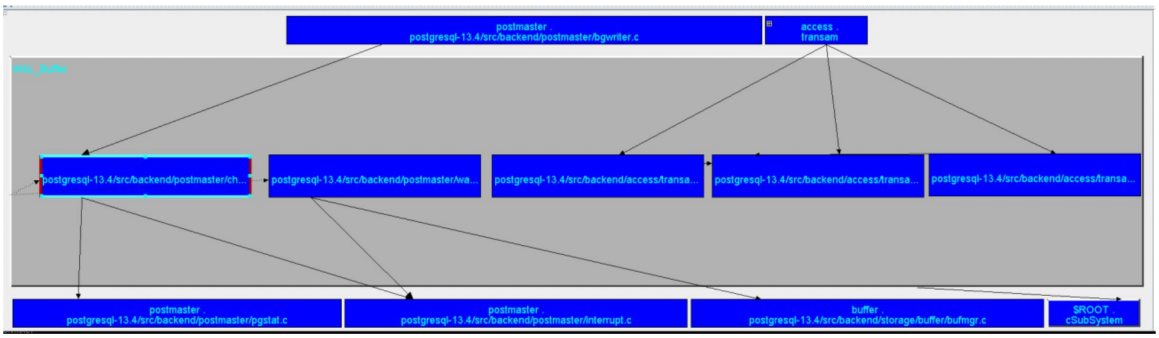


Figure 8 - Concrete Architecture of WAL component

The sequence begins with the check pointer creating a checkpoint, either when the checkpoint timeout time has reached, as indicated by checkpoint_timeout, or when the WAL buffer load has reached the predetermined threshold, as indicated by max_wal_size. Either way, the WAL log sends a signal to the shared buffer, instructing it to apply the dirty data, that is modified data not yet written to the disk, to commit the change. It also sends information to the transaction manager, which in turn instructs the commit log to record the committed transactions.

Use Cases:

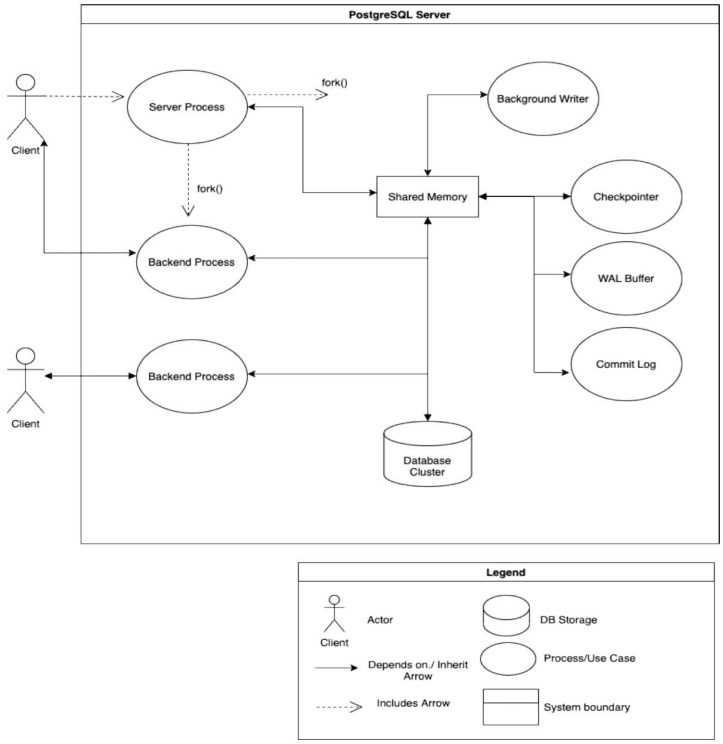


Figure 9 - Use Case

Data Dictionary:

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.
ECPG	Embedded SQL C preprocessor
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.
Write-Ahead Logging(WAL)	Method to ensure consistency and integrity of data

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

Conclusions

In the end, the concrete architecture did not match up exactly with the new conceptual architecture that had been formed. There were many unexpected dependencies, some which were logical and some that were not. It was learned that unjustified dependencies were usually the effect of convenience or legacy reasons. However, the new conceptual architecture that was formed matched the concrete a lot better than the old conceptual (see Assignment One). One idea to enhance PostgreSQL is to separate server processes from data storage to facilitate scaling-out. The pqsignal protocol could be rewritten and moved to minimize repeated code and add additional cohesiveness to the subsystems.

Lessons Learned:

Because of the cyclical nature of reflection analysis, it was important to realize that being completely correct on the early interactions is unrealistic and time consuming. Using each iteration to bring the concrete and conceptual architectures closer quickly became essential to the process of deriving a coherent conceptual architecture. Sometimes, good commenting in the source code made the task of understanding subsystems much easier. Other times, the lack of comments became a major hindrance. This stressed the importance of good commenting practices while coding. An important lesson is that a system's concrete architecture will not always strictly follow a particular architecture style. Certain dependencies are counter to the architecture because they are done for convenience, or to optimize performance. Because of the way that open-source projects evolve, there are legacy interactions that should have been updated, but haven't looked into yet. A project like this one is constantly looking to improve its functionality, and therefore sometimes the organizational architecture is overlooked. Using a reference architecture certainly helps the creation of a reliable conceptual architecture

References:

Ahmed, N. (2018). Let's go to basics- PostgreSQL memory components. *Fujitsu.* https://www.PostgreSQL.fastware.com/blog/back-to-basics-with-PostgreSQL-memory-components[2]

Integration, N. A. on D., Integration, A. M. on D., & Asana, A. J. on. (2021, November 3). *Working with postgres wal made easy - learn*. Hevo. Retrieved November 8, 2021, from https://hevodata.com/learn/working-with-postgres-wal/.

PostgreSQL. (n.d.). Retrieved November 8, 2021, from https://www.PostgreSQL.org/files/documentation/pdf/13/PostgreSQL-13-US.pdf

Informit. InformIT. (n.d.). Retrieved November 8, 2021, from https://www.informit.com/articles/article.aspx?p=30309&seqNum=2

File browser. PostgreSQL. (n.d.). Retrieved November 8, 2021, from https://www.PostgreSQL.org/ftp/source/v13.4/

Hironobu SUZUKI, https://www.interdb.jp/pg/. (n.d.). *The internals of postgresqlfor database administrators and system developers*. The Internals of PostgreSQL : Chapter 8 Buffer Manager. Retrieved November 8, 2021, from https://www.interdb.jp/pg/pgsql08.html.