PostgreSQL Report

Wendy-Wenxi Hu, Meng Wei, Yunhao Mao

March 25th, 2018

1 Introduction

PostgreSQL is an open-source object-relational database management system with powerful functionality and extensibility[1]. PostgreSQL is released under its own *PostgreSQL License* which means it can be freely used, copied, modified and distributed for any purpose, similar to softwares that are under BSD or MIT licenses[2]. According to db-engines.com, PostgreSQL is currently the 4th most popular DBMS falls behind Microsoft SQL server and is slight ahead of MongoDB.[3].

PostgreSQL started off as POSTGRES project in 1986 at University of California, Berkeley led by Professor Michael Stonebraker, the initial design goal was to create a relational DBMS that successes INGRES project(one of the earliest relational DBMS from 70s) but with better support for complex objects, user extendibility for data types, and other features[4]. New features such as support for multiple storage managers, an improved query executor, and a rewritten rule system were added in subsequent versions in few years. In 1994, Andrew Yu and Jolly Chen added a SQL language interpreter along with many more modern DBMS features to POSTGRES and renamed it to Postgres95. By 1996, the name PostgreSQL was finally given to the project and the new PostgreSQL DBMS was reborn with version 6.0 inheriting POSTGES's version number[1]. The latest major release is PostgreSQL 10 in October, 2017.

PostgreSQL supports a variety of architectures and operating systems. The latest PostgreSQL 10 supports the following CPU architectures: vx86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, ARM, MIPS, MIPSEL, and PA-RISC. It also supports the following operating systems: Linux (all recent distributions), Windows (Win2000 SP4 and later), FreeBSD, OpenBSD, NetBSD, macOS, AIX, HP/UX, and Solaris[1].

Comparing to other relational DBMS such as MySQL and SQLite, PostgreSQL has several strong advantages that make it stand out from the rest. PostgreSQL has complete supports for ACID(Atomicity, Consistency, Isolation, and Durability) which makes it favoured over other RDBMS when reliability and data integrity are essential requirements. It is also highly programmable so that it can be extended with custom procedures which can be used to simplify

the execution of complex database operations. Moreover, PostgreSQL supports numerous complex geometric data types such as `circle`(circle on a plane), `line`(infinite line on a plane), and `path`(geometric path on a plane)[5]. This allows PostgreSQL to provide functionality to manage complex database designs such as a Geoinformatics database.

This report will mainly be investigating PostgreSQL version 10.3; however since the general architecture of PostgreSQL wasn't changed much since the early versions, information from older sources will be used.

2 Potential Applications

PostgreSQL was used by companies and corporations of various sizes due to its distinguishing features of free, easy to use and flexible. Some of the most famous users include Skype, Uber(previously), IMDB.com, many universities such as UC Berkeley and government agencies such as U.S. State Department[6].

Using PostgreSQL as data storage solution can be much beneficial for projects or applications with following characteristics:

- GIS applications, PostgreSQL supports multiple geometric types and has some powerful geometric and spatial addons such as PostGIS[7].
- Small companies or open source projects, PostgreSQL's license permits free usage and modification even in commercial projects to reduce the cost of purchasing a commercial DBMS.
- Academic and teaching uses, PostgreSQL has a deep academic background and an active community which are helpful for learning technologies behind databases[10].

However, not all applications are suitable of using PostgreSQL. If a company just wants to purchase a full data storage solution, using a commercial database is likely easier to develop and maintain. Otherwise dedicate database specialists are needed to utilize most of the powerful PostgreSQL functions. Historically, PostgreSQL suffered issues such as lack of features, bad admin tools and inefficient architecture for writes[10] [11]. DBMS is something that developers do not change much since database migration is a hard and tedious process, therefore many legacy systems that were initially built with other DBMS tends to stay with that database unless migrating provides a much greater advantage and this prevents PostgreSQL from becoming more widely used. However, with PostgreSQL becomes more and more polished and popular, it can be expected that more developers will learn the power of PostgreSQL and switch to it.

3 System Architecture

3.1 Architecture Overview

PostgreSQL has a client-Server architecture, client makes DB requests through client APIs such as libpq, JDBC or even HTTP REST APIs. The PostgreSQL back-end server will then process the requests and return the result to the client[8].

PostgreSQL's client-server architecture has the advantage of having good security/reliability and the ability to work in a network environment.

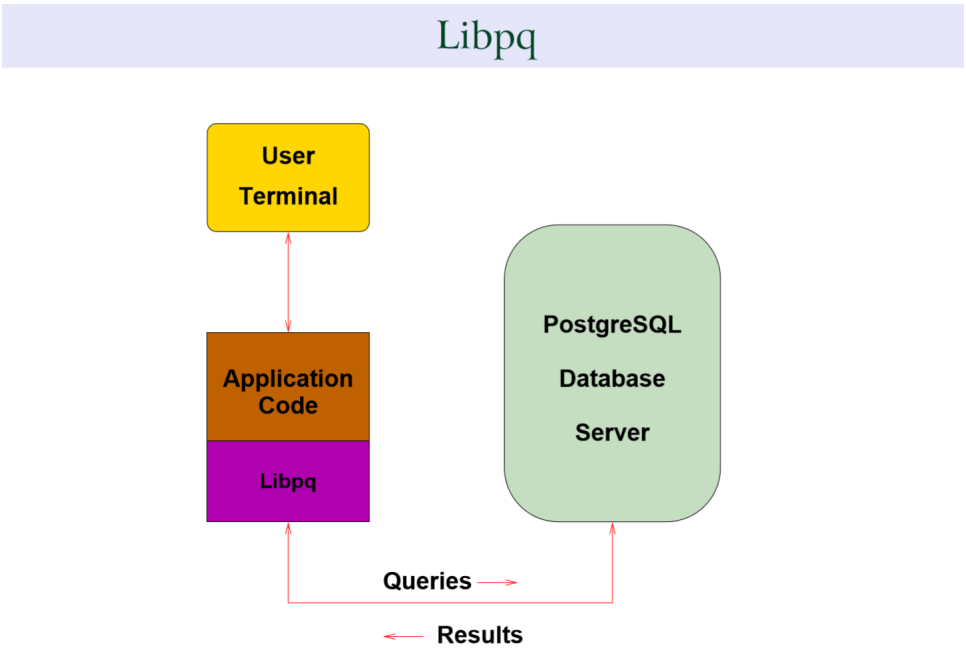

Figure 1: Picture shows the overview of PostgreSQL operation by Momjian, Bruce."PostgreSQL internals through pictures" Software Research Associates (2001).

Other than the client process and the back-end server process, there is the postmaster(daemon) process. The postmaster is the initialization process of a PostgreSQL server. On start-up, it will allocate shared memory space for disk buffer and tables as well as spawning server processors based on number of requests coming from the client[12].

The PostgreSQL server processes handles queries and perform I/O toward tables in memory where the OS's file system takes care of storing data into physical disks.

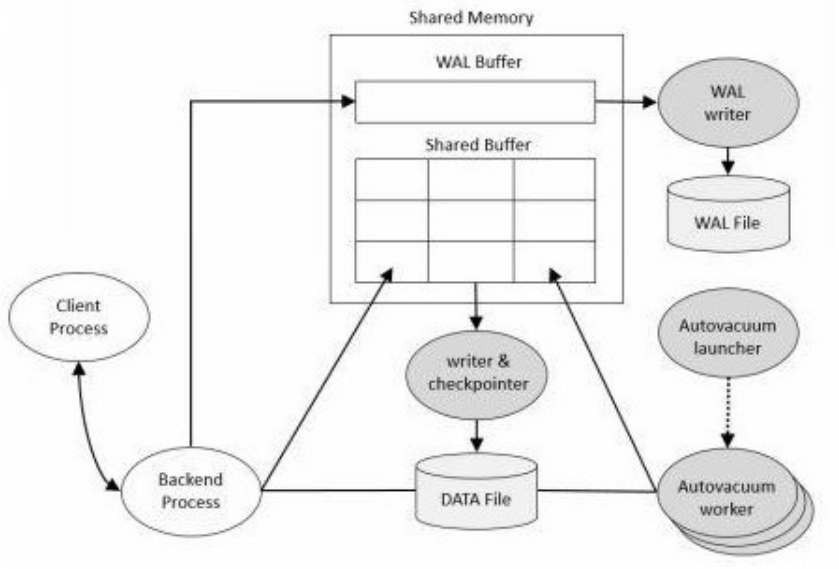

Figure 2: Picture shows the internal structure of PostgreSQL[10]

3.2 System Catalogs

PostgreSQL uses system catalogs to store metadata objects such as tables, columns and indexes. Some basic catalogs and functions are `pg_class` has table information such as name for each table, `pg_attribute` has information such as data type for each column in a table has and pg_index that relates a table to its indexes. There are also `pg_proc` to define functions and `pg_language` to define language used for functions. Catalogs themselves are essentially tables and PostgreSQL supports extension of catalogs[8]. Catalogs are maintained automatically by the system, for example, CREATE DATABASE statement will create a database on disk as well as adding a row to `pg_database` catalog[1].

3.3 Query Processor

PostgreSQL has a four-stage query processor, they are the Parser, the Rewriter, the Planner and the Executor.

- Parser is the first stage of query processing when a plain text query is requested from the client. A lexer generates tokens for each of key word or identifier then passes to the parser. The parser will generate a parsed tree using a set of grammar rules and actions but there is no system catalog lookups, the entire process is only syntactic analysis, similar to what a complier's parser would do. Then a transformation process will take the parsed tree and identify which tables, functions, and operators are used in the query to generate a query tree[1].

- The query tree is sent to the Rewriter as the second stage of processing. The rewriter is used by PostgreSQL's rule system to implement features such as views. The rules are stored in the catalog pg_rewrite and the rewriter will rewrite query tree based on the rules the query applied to. For example,

  ```
  CREATE VIEW myview AS SELECT * FROM mytab;
  ```

  can be rewritten as

  ```
  CREATE TABLE myview (same attribute list as for mytab);
  CREATE RULE "_RETmyview" AS ON SELECT TO myview DO INSTEAD
  SELECT * FROM mytab;
  ```

  [15]. Rules can also be defined and extended by users.

- Query tree then goes to the query evaluation stage, it is sent to the Planner(Optimizer). The planner will first create the plan of scanning tables that the query uses, and based on types of index used in the relation, different plans will be generated. Then it will plan for different join methods: nested loop, merge or hash join. Finally, the planner will decide which plan should be using by calculate an estimated cost based on statistic of the database and predefined cost variables[1].

- Finally, the Executor will take the plan created by the planner and run each node in the query tree recursively. Executor will compute each node and return the result rows to its top node. Nodes are also responsible for applying any selection or projection expressions that were assigned to them by the planner[1].

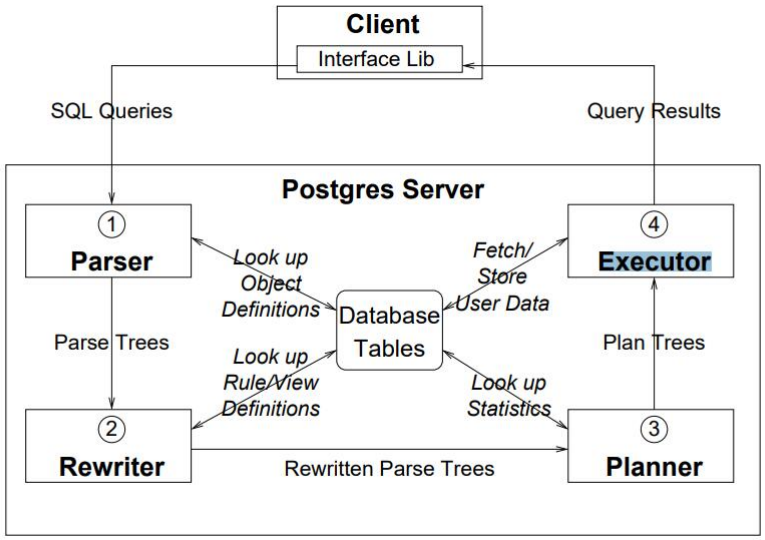

Figure 3: Steps of query processing: overview by Lane, Tom. "A Tour of PostgreSQL Internals" Open Source Database conference. 2000.

The above diagram shows the interaction among each part of the query processor and tables.

3.4 Concurrency Control

In PostgreSQL, data consistency is maintained by a model called Multiversion Concurrency Control(MVCC). The basic concept is that each SQL query will only see a snapshot of historical data instead of the current state of the data, thus viewing inconsistent data produced by concurrent transactions performing updates on the same row can be prevented[1].

PostgreSQL provides 4 kinds of isolation levels: read uncommitted, read committed, repeatable read and serializable which prohibits or allows behaviors such as dirty read, nonrepeatable Read, phantom read or serialization anomaly. It also provides 3 levels of locks: Table-level Locks, Row-level Locks and Page-level Locks. There is also an Advisory Lock which the DBSM does not enforce anything and it is left to the application to define the behavior[1].

3.5 Write Ahead Logging

WAL is used in PostgreSQL like many other DBMS to provide crash recovery and improve reliability. WAL logs are stored in the directory `pg_wal` under the data directory in segments files of 16 MB each by default, and each segment is split into pages of 8KB. A data type of `pg_lsn` is provided to be used as the Log Sequence Numbers[1]. PostgreSQL has a feature called Asynchronous Commit, which means the in memory WAL is flushed to the disk asynchronously by a background process instead of blocking other transactions. Doing this speeds up the transaction time, however, it does increase the risk of losing data between transaction returns successful and before data written to the disk[1].

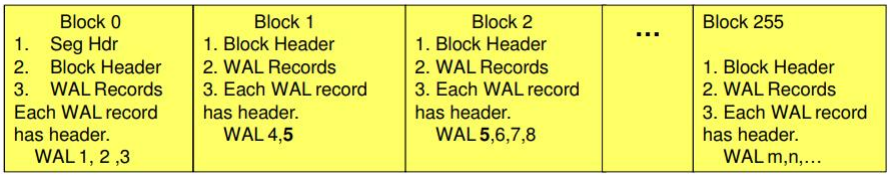

Figure 4: Layout of WAL segment file, WAL Internals Of PostgreSQL, PostgreSQL www.pgcon.org/2012/schedule/attachments/258_212_Internals%20Of%20PostgreSQL%20Wal.pdf

3.6 Routine Vacuuming

Vacuuming is a maintenance process in PostgreSQL, it can free up unused disk space and update statistics used by the query planner. Vacuuming can be done manually by issuing VACUUM command, or let the autovacuum daemon to perform VACUUM and ANALYZE commands automatically[1].

There are VACUUM and FULL VACUUM where VACUMM simply reclaims space and makes it

available for re-use while normal DB operations can conduct concurrently. The latter one rewrites the entire contents of the table into a new disk file with no extra space, allowing unused space to be returned to the operating system. This operation takes longer and the tables need to be locked[1].

4 Query Language and Interface

4.1 Background

SQL, Structured Query Language, became a standard of the ANSI, American National Standards Institute in 1986 and of the ISO, International Organization for Standardization in 1987[1]. Although ANSI SQL successfully specifies how a system should manage, manipulate and query data, some of its features are too complex to be totally portable. Moreover, since some specifications of ANSI SQL are ambiguous to be implemented and ANSI SQL didn't consider some practical issues including but not limited to indexing and file storing, lots of database management systems changed or extended the standard ANSI SQL to meet their requirements. Up to 2018, the latest version of PostgreSQL extends ANSI SQL and implements Procedural Language/PostgreSQL Structured Query Language, known as PL/pgSQL. Latest PL/pgSQL was based on the ANSI-SQL:2008 standard.

4.2 Characteristics

PL/pgSQL implements SQL in multiple aspects, including multiversion concurrency control, data types, subqueries, foreign keys, etc. Beyond ANSI SQL standard, PostgreSQL also provides unique, partial, compound, and functional indexes which can support any of B-tree, R-tree, GiST or hash.[1] Moreover, PostgreSQL also extends ANSI SQL standard through features including Single or Multiple Table Inheritance, Rule Systems and Database Events. Table Inheritance helps database managers to create new tables form existing ones; Rules system, sometimes named the Query Rewrite System as mentioned in the last section, helps database managers to restrict or transform specified query operations on some tables or views; The Events System helps database customers to exchange information at a peer-to-peer or advanced level using LISTEN and NOTIFY commands.[1]

Furthermore, PostgreSQL serves flexibility to its users. Customers or Managers can add new aggregate functions, operators, data types, index methods and procedural languages. This is feasible because PostgreSQL's catalog stores not only tables or columns information but also details related to data types, functions, indexing, etc.[1]

4.3 Procedural Languages

PostgreSQL also supports other programming languages other than SQL and C, which includes Java, Perl, Tcl, Python, JavaScript, etc.[1] These programming languages, known as Procedural Languages, are not directly parsed and executed by PostgreSQL, but by corresponding handlers, which help PostgreSQL interpret or execute Procedural Languages. PostgreSQL's core distribution includes four Procedural Languages, including PL/pgSQL, PL/Python, PL/Tcl, PL/Perl. [1] There are also some helpful Procedural Languages developed and maintained by groups other than PostgreSQL core distribution. PostgreSQL documentation lists several of them:

Name	Language	Website
PL/Java	Java	https://tada.github.io/pljava/
PL/Lua	Lua	https://github.com/pllua/pllua
PL/R	R	http://www.joeconway.com/plr.html
PL/sh	Unix shell	https://github.com/petere/plsh
PL/v8	JavaScript	https://github.com/plv8/plv8

Figure 5: Procedural Languages
https://www.postgresql.org/docs/10/static/external-pl.html

4.4 Interface

Despite Procedural Languages, PostgreSQL also specifies library interfaces as well. PostgreSQL core distribution includes two client interfaces: "libpg" and "ECPG". "libpg" is a primary C language interface which other interfaces are built on. It mainly functions as a communication channel between PostgreSQL backend server and programs. Other interfaces based on "libpg" should include the header file "libpq-fe.h" and must link with the libpq library.[1] "ECPG" is an embeded SQL package for PostgreSQL. It was written by Linus Tolke (<linus@epact.se>) and Michael Meskes (<meskes@postgresql.org>).[1] Programs written in embedded SQL are passed through PostgreSQL's embedded SQL preprocessor and converted to a basic "C" code. After that, it is passed through C compiler and get executed. Besides "libpg" and "ECPG", there are also interfaces in other languages maintained by groups other than PostgreSQL:

Name	Language	Comments	Website
DBD::Pg	Perl	Perl DBI driver	http://search.cpan.org/dist/DBD-Pg/
JDBC	Java	Type 4 JDBC driver	https://jdbc.postgresql.org/
libpqxx	C++	New-style C++ interface	http://pqxx.org/
node-postgres	JavaScript	Node.js driver	https://node-postgres.com/
Npgsql	.NET	.NET data provider	http://www.npgsql.org/
pgtcl	Tcl		https://github.com/flightaware/Pgtcl
pgtclng	Tcl		http://sourceforge.net/projects/pgtclng/
pq	Go	Pure Go driver for Go's database/sql	https://github.com/lib/pq
psqlODBC	ODBC	ODBC driver	https://odbc.postgresql.org/
psycopg	Python	DB API 2.0-compliant	http://initd.org/psycopg/

Figure 6: Client Interfaces

https://www.postgresql.org/docs/10/static/external-interfaces.html

5 Storage Details

5.1 Database Cluster and Database Layout

The storage area on disk is called a database cluster. It is a collection of databases and is managed by a single PostgreSQL server. The configuration files and data files for the database cluster are stored under the cluster's data directory. This directory is created upon the execution of `initdb` which initializes a new database cluster. The location of this directory varies depending on the installation or the user's operating system, and is usually set to the environment variable PGDATA.

The PGDATA directory contains several control files and subdirectories such as `base`, `global`, and `pg_wal`. The `base` subdirectory consists of subdirectories which contain the database's file for each database in the cluster. These per database subdirectories in `base` are named after the OID(Object Identifier) of their associated database. The OID is an unique unsigned 4-byte integer assigned to each database object such as tables, views, and databases themselves, and it is used by PostgreSQL internal management[1]. The mappings of the database objects and their OIDs are stored in system catalogs depending on the object types. For instance, the OIDs for each database in the cluster is stored in `pg_database`. The `pg_global` contains the system catalogs that are physically share across the databases in the cluster, such as `pg_database`.

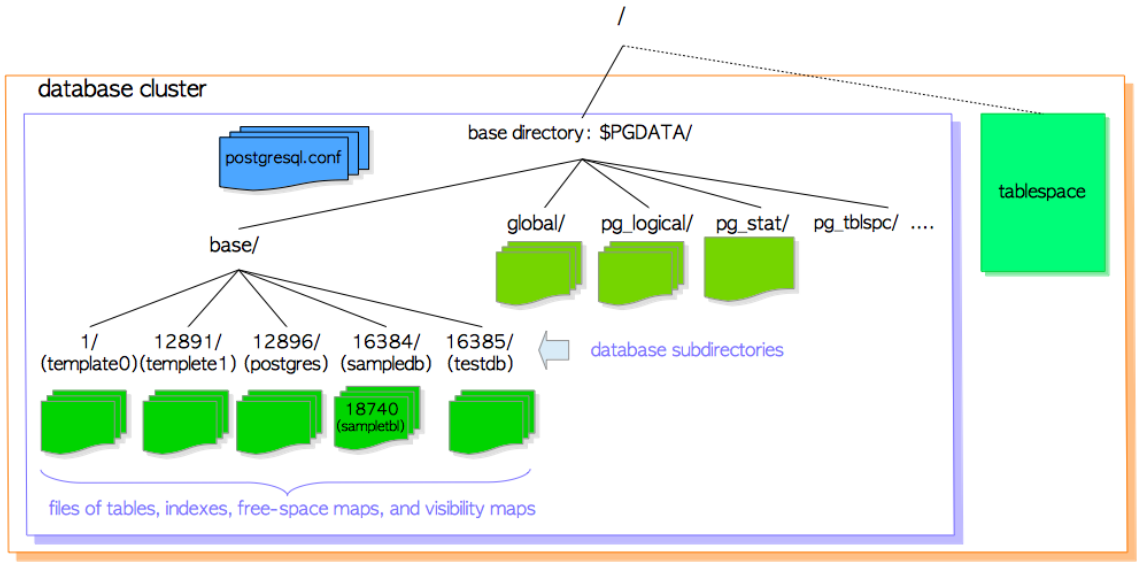

Figure 7: An example of database cluster file layout
http://www.interdb.jp/pg/pgsql01.html

5.2 Tables and Indexes

The data files for the tables and indexes can be found in the subdirectory of the belonging database. These files are named with the table or index's filenode number. The filenode number of a table or an index usually matches with its OID, but is not always the case. The filenode number is used to manage the data files of the table/index while the OID is used to manage the table/index object internally that represents the data files. When the size of the table/index exceed 1GB, PostgreSQL creates additional file with name of form [filenode.1], [filenode.2] and so on to store the excess data.

For each table and index, there is a free space map that keeps track of the free space available to use. It is stored in a file with name of form [filenode_fsm]. The free space map is a tree of free space map pages where the bottom level of the tree stores the information about the free space in each table or index and the upper level of the tree stores the aggregated information of the bottom level. The data in the free space map is updated by the VACUUM in order to achieve MVCC.

In addition, each table also has a visibility map which determines whether each page has dead tuples. It is stored in a file with name of form [filenode_vm]. Visibility map reduces the cost of the process of VACUUM[16].

5.3 Page Layout

Tables and indexes are stored as an array of pages, each page has a fixed length and is usually 8KB. When PostgreSQL needs to read a tuple, it reads the entire page which contains the desired

tuple; When PostgreSQL needs to write to a tuple in a page, it writes a new copy of the entire page to the disk[1]. An ordinary page consists of 4 parts: page header, row pointers, free space, and the data tuples. The header is 24 bytes long and contains meta informations such as the checksum (`pd_checksum`), the pointer indicating the beginning of the free space (`pd_lower`), the pointer indicating the end of the free space (`pd_upper`) and other data necessary for the WAL (`pd_lsn`). The row pointers consists of an array of pointers indicating the location of the beginning of the tuple they are referring to, with Nth pointer referring to the Nth tuple. The tuples are the record data stored in the table. They start from the back of the page with the first tuple stored at the end, thus the row pointers and the tuples both grow toward the middle of the page. The gap between the row pointers and the tuples is referred as free space. Some pages of index also contain a special space used as a utility section that access methods can store relevant data of the index structure[1].

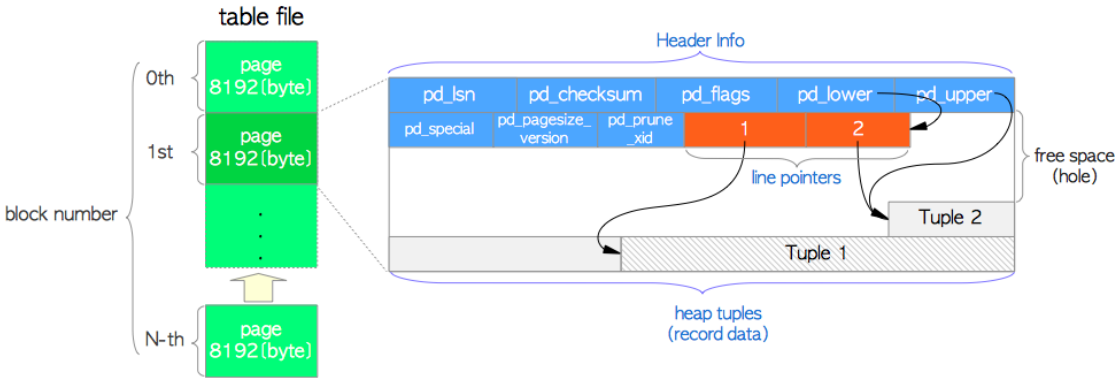

Figure 8: page layout of a table file
http://www.interdb.jp/pg/pgsql01.html

Each tuple in the page has a tuple number for identification called CTID(tuple identifier). CTID consists of the page number of the page that contains the tuple and the index of the row pointer of the tuple[1].

5.4 The Oversized Attribute Storage Technique

Since PostgreSQL uses fixed sized pages which are usually 8KB and tuples cannot span multiple pages, the table is impossible to store a tuple with very long field values. Then the oversized attribute storage technique(TOAST) is employed to solve this problem. When there is a tuple that exceed the size that is allowed to store in the page, TOAST breaks the tuple of large field into multiple smaller parts and stores those parts as separate rows in a TOAST table. A pointer to the TOAST table will be created by the TOAST management code and stored at the original space where the oversized data was supposed to be stored. Not all data types support TOAST since

it is unnecessary to introduce overhead as some data types cannot have large field values such as INT. Thus, only data types with variable length support TOAST. TOAST is more efficient than allowing the oversized tuple to span over multiple pages since the out-of-line value will only be pulled out when the field is selected by the user to be displayed. Moreover, the TOAST table can be stored either on disk or in memory depending on the need[1].

6 Evaluation

6.1 Introduction

PostgreSQL, as a mature database management system, could query data quickly and maintain data efficiently. As for query optimization, PostgreSQL's query Planner calculates costs of possible plan trees and chooses the optimal one[1]. We tested different query plans that PostgreSQL can create and then we analyzed the result in this section.

All the tests were conducted on PostgreSQL 10.3 server on a Macbook Air(Early 2015 model) with an Intel Core i5 Processor, 8GB of memory and a 128GB SSD.

6.2 Logic Behind PostgreSQL Planner

Query optimization could be influenced by several facts. The most important one is indexing. PostgreSQL supports various indexes including B+ Tree, Hash, Sequential Scan, Bitmap Index, R-tree, Generalized Inverted Index and Generalized Search Tree. PostgreSQL handles indexes by separating them from the relation they describe, storing them as their own physical tables. These indexes could be accessed using PostgreSQL's `pg_class` catalog[1]. For a single relation, PostgreSQL also provides command '\d' to view its corresponding index information.

Different indexes have their specific pros and cons. Generally speaking, hash index is good at fetching a single tuple in a relation, but cannot perform range selection and "IS NULL" checking. B-tree is the default index PostgreSQL chooses when it executes 'CREATE INDEX'. Since B-tree is a balanced tree, equality and range queries could be performed efficiently. Moreover, B-tree could be advanced using clustered data. Bitmap Index creates a bitmap for possible values of column. It works like partitioning tuples into different categories and thus provides compact representation. Bitmap Index usually comes with other index methods and functions like logical operators upon different index filters[1]. Some other indexes like R-tree and GiST supports more generalized case like coordinate values and geometric data type, and works similar to B-tree.

With these properties of Indexes, the planner could estimate the cost of alternative plan trees for a query. Specifically, since JOIN sequence is hard to optimize and perform due to exponential

growth with respect to the number of table, we will discuss this hardest case in details. PostgreSQL has a Genetic Query Optimization (GEQO) process which generates plans for queries performed on individual relations using the randomized algorithm[1]. It generates these possible join sequences at random like solving Traveling Sales Man problem, where each query statement and relation stands for a single node. After getting these join sequences, PostgreSQL's Planner will estimate the cost of performing the query using each join sequence. Conceptually, the sequence with lower estimated cost is more likely to run faster than others. The planner will drop those with higher cost, and then repeat this procedure several times. Due the the characteristics of randomized algorithm, the expectation results are near optimal. JOIN queries could also be optimized through applying various join methods including nested loop, hash join and merge join based on the which indexes the joined tables are using.

6.3 Query Evaluation Strategies

We measured performance of a query with different plans. We tested difference scanning methods and joining methods by forcing the PostgreSQL planner to use a certain strategy. This was done by setting the corresponding planner configuration variables to disable or enable certain planning methods. The following settings were used: `enable_mergejoin, enable_hashjoin, enable_seqscan, enable_indexscan`.

The performance of each query planning is measured in databases with different number of records. The data records are generated by a tool called `fake2db`[18] with the following command:

```
fake2db.py --rows num_row --db postgresql
```

This command generates 5 tables of size num_row, each with several fields of type varchar(30). We only uses 2 tables, `company` and `customer`, with the assumption of the field `email` in relation `customer` refers to `email` in `company`.

To create index in database, CREATE INDEX statement was used to give columns hash or B+ tree indexes. We also used EXPLAIN and ANALYZE statement to display the planning details and execution time of a query. Execution time from the result EXPLAIN ANALYZE returned was used to measure the performance.

6.3.1 Comparisons Between Indexes

We run both single selection and range selection on field `email` of relation `company` in databases with different sizes.

1. Performance of Single Selection between Sequential scan, Hash index scan and B+ tree index scan(unclustered).

The following SQL query was executed:

```
EXPLAIN ANALYZE select *
from company where email = 'elizabethbrown@mitchell.com';
```

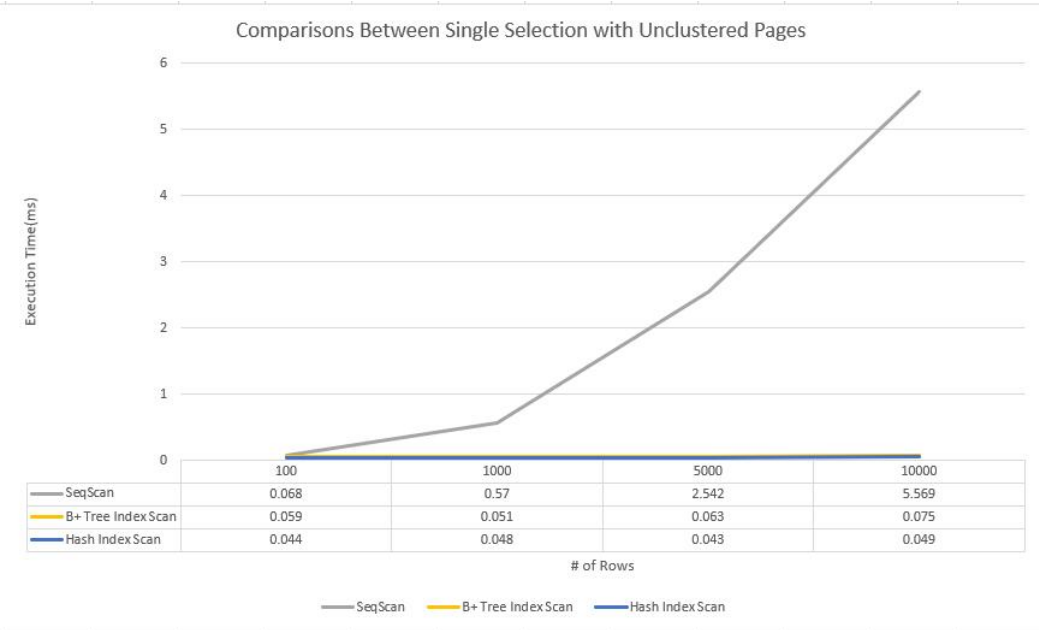

Figure 9

As the size of the database grow, the execution time of sequential scan increases while the performance of index scan of both type remains highly efficient. The performance of hash index is slightly better than B+ tree index when performing single selection.

2. Performance of Range Selection between sequential scan, Hash index scan and Btree index scan(Unclustered)

   The following SQL query was executed:

   ```
   EXPLAIN ANALYZE select * from company
   where email < 'elizabethbrown@mitchell.com';
   ```

Figure 10

For range selection, both hash index and B+ tree index cost more time comparing to their single selection performance, and the overall performance of Hash index is very close to the performance of normal sequential scan. Moreover, since data are unclustered, B+ tree index doesn't perform well, but it is still better than Hash index when performing range selection.

3. Performance of Single Selection between Sequential scan, Hash index scan and B+ tree index scan(Clustered)

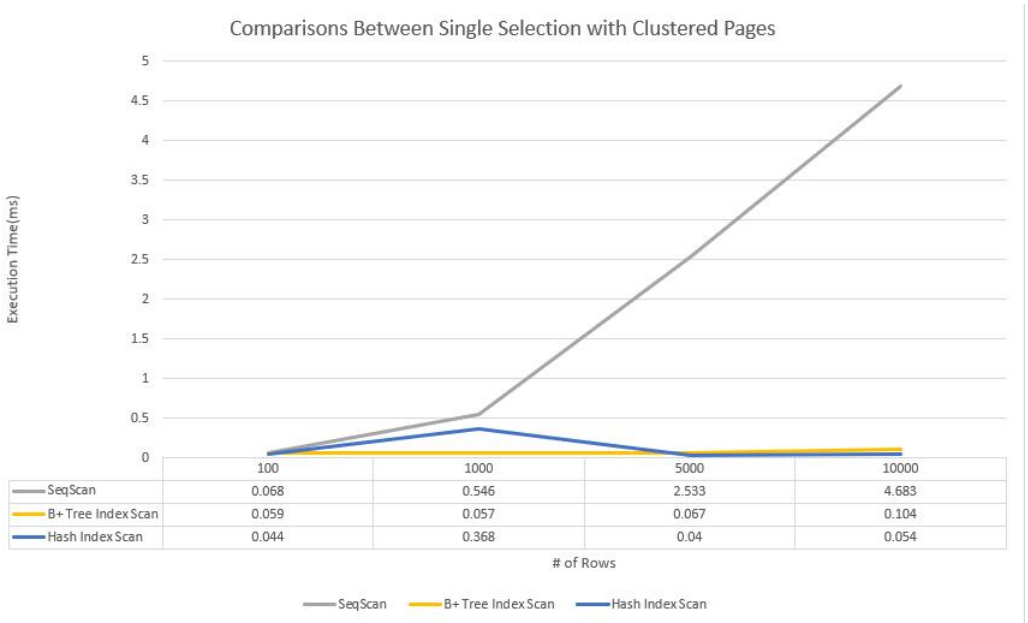

Figure 11

The performance of the 3 scanning methods with clustered index does not change much

comparing to the unclustered performance since sorting the data does not affect the process of single selection.

4. Performance of Range Selection between Sequential scan, Hash index scan and B+ tree index scan(Clustered)

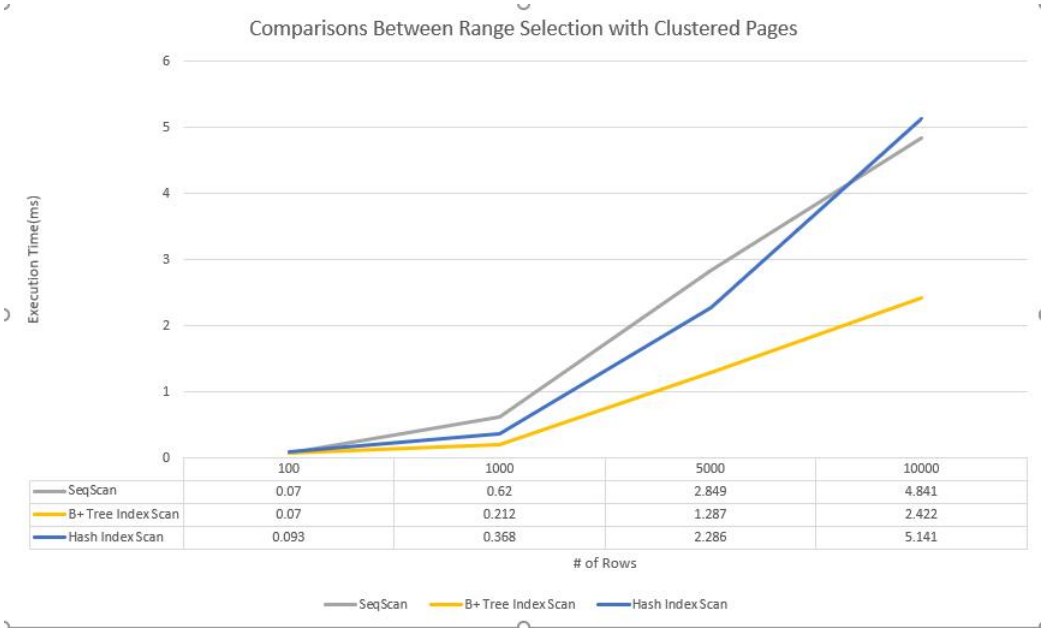

Figure 12

The performance of B+ tree index improved significantly when the index is clustered.

6.3.2 Comparison Between Different Join methods

We run a range selection on inner join of relation `company` and `customer` on the field `email`.

1. Comparisons Between Range Selection

```
EXPLAIN ANALYZE select *
from company join customer on company.email = customer.email
where company.email < 'elizabethbrown@mitchell.com';
```

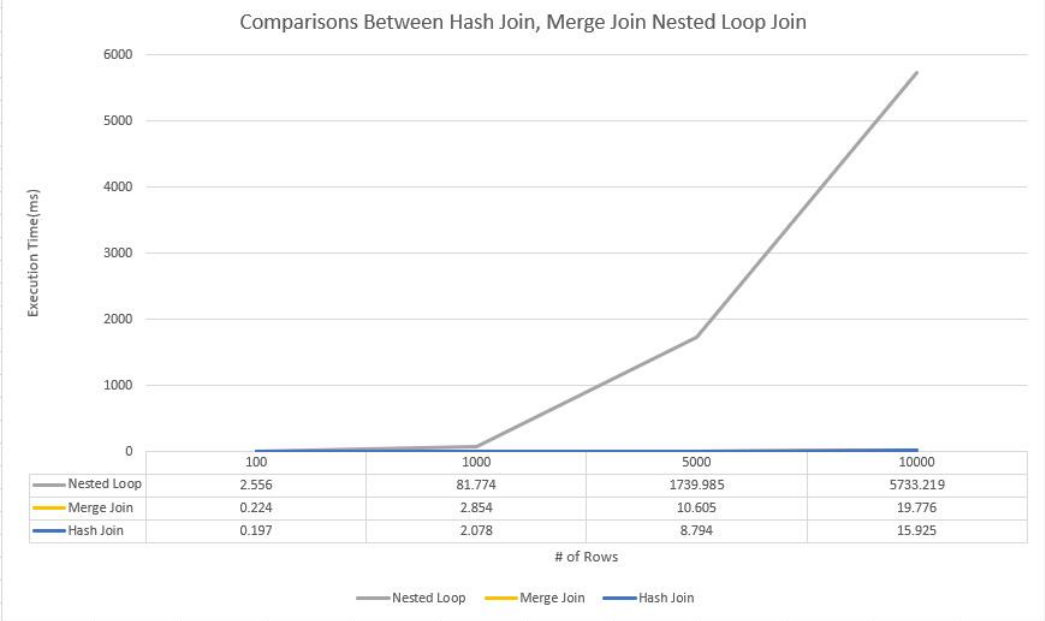

Figure 13

According to the test result, Hash Join and Merge Join is highly efficient comparing to Nested loops join. We notice that as the database size grows larger, the difference between execution time of Hash Join and Merge Join becomes more obvious. This result implies that Hash Join might have a significantly better performance over Merge Join when the database size is very large.

6.4 Crash Recovery

Crash Recovery was tested by inspecting WAL logs. Test was conducted by updating the database then inspecting the WAL data file using `pg_waldump`.

Two transactions were performed with the following SQL statements:

```
UPDATE company set email = 'abc@mail.com' where id = 900;
DELETE from company where id = 4999;
```

Then the WAL was inspected with the following command:

```
sudo bin/pg_waldump data/pg_wal/000000010000000000000005 -d
```

The following output was obtained:

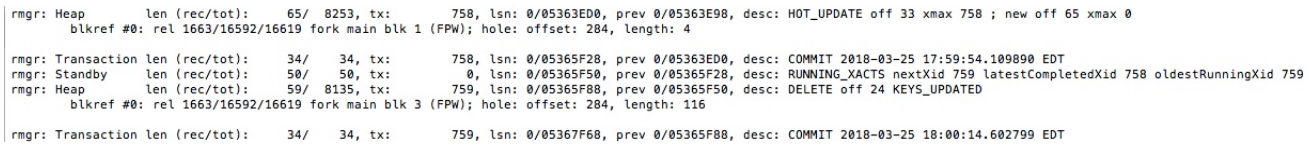

Figure 14

It can be observered that UPDATE had WAL log entry with LSN 0/05363ED0 and the transaction number is 758. Old off for the tuple in page is 33 and new tuple is in page offset 65. The length of tuple is 65. blkref is the file that the page is in.

The next log showed the the transaction UPDATE belongs had committed since it was completed.

DELETE had similar logs in the WAL.

Unfortunately, due to the limitation of our personal computers, the performance of crash recovery. However we can conclude that writing to the tables indeed logged in the WAL and it can be useful in crash recovery.

7 Conclusion

PostgreSQL is a continuously evolving database management system. By researching different aspects, we discover that it is powerful, efficient, and flexible. Its client-server architecture and query process system enable it to be a reliable and efficient system. It not only implements ANSI SQL standard 2008 but also extends it as PL/pgSQL which provides extra features and functionalities. It also contributes flexibility by serving extra procedural languages and interfaces. PostgreSQL managed to store the data efficiently as it utilizes TOAST to deal with oversized data. The overall query performance is descent as the planner chooses the minimum cost query plan. In addition, the source code is highly organized and readable that helped us to understand PostgreSQL better. We believe that PostgreSQL will be more popular and achieve a high market share in the future.

References

[1] “PostgreSQL 10.3 Documentation.” PostgreSQL, www.postgresql.org/docs/10/static/index.html.

[2] “License.” PostgreSQL, www.postgresql.org/about/licence/

[3] “DB-Engines Ranking.” Popularity Ranking of Database Management Systems, db-engines.com/en/ranking

[4] Stonebraker, Michael, and Lawrence A. Rowe. *"The design of Postgres"*. Vol. 15. No. 2. ACM, 1986.

[5] Tezer, O.S. “Contents.” SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems | DigitalOcean, DigitalOcean, 18 July 2017, https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of

[6] “Featured Users.” PostgreSQL, www.postgresql.org/about/users/

[7] http://postgis.net/

[8] Lane, Tom. *"A Tour of PostgreSQL Internals"*. Open Source Database conference. 2000.

[9] Momjian, Bruce. *"PostgreSQL internals through pictures"*. Software Research Associates (2001).

[10] Conrad, Tim. *"PostgreSQL vs. MySQL vs. Commercial Databases: It's All About What You Need."*

[11] “Why Uber Engineering Switched from Postgres to MySQL.” Uber Engineering Blog, 26 July 2016, eng.uber.com/mysql-migration/.

[12] Chauhan, Chitij. “Become a PostgreSQL DBA: Understanding the Architecture.” Severalnines, Severalnines, 20 Oct. 2017, severalnines.com/blog/become-postgresql-dba-understanding-architecture.

[13] "ISO/IEC 9075-1:2016: Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)"

[14] PostgreSQL contributors (2011). "PostgreSQL server programming". PostgreSQL 9.1 official documentation. postgresql.org. 03 September 2012.

[15] PostgreSQL Programmer's Guide, www.cs.helsinki.fi/u/laine/postgresql/programmer/programmer.htm.

[16] SUZUKI, Hironobu. “The Internals of PostgreSQLfor Database Administrators and System Developers.” The Internals of PostgreSQL : Chapter 1 Database Cluster, Databases, and Tables http://www.interdb.jp/pg/pgsql01.html

[17] "PostgreSQL Indexes." Leopard
https://leopard.in.ua/2015/04/13/postgresql-indexes#.WrfNctMhKb9

[18] fake2db
github.com/emirozer/fake2db