PostgreSQL Analysis

Sistemas de Bases de Dados

NOVA School of Science and Technology

Abstract. The present document is the paper for the course Sistemas de Bases de Dados, that will present our study and analyse of the Data Base System PostgreSQL. We will discuss topics such as file structure and storage, indexing and hashing, query optimization, control of concurrency and support for distributed database, while demonstrating the execution of the system's functionalities.

Keywords: PostgreSQL · Database System

1 Introduction

This document will be composed of a rigorous analysis and exploration of the Data Base System PostgreSQL. As mentioned in the abstract, the paper will be divided into 6 topics: how does the Data Base System store files, what types of indexing it supports, how does work the query processing and optimization, in what way does the system controls concurrency and if it has support for distributed database, and if yes, how does it work. We will also reference some particular features about PostgreSQL.

In each chapter, we will provide also a comparison with the system that we used during the classes, Oracle 18c. We will analyse differences and similarities with each other and understand for which tasks one is better than the other.

But before analysing the referenced topics we first need to understand what is PostgreSQL and why is it used. PostgreSQL is a relational database system that suports both SQL and JSON querying and is used as a primary database for multiple web mobile and analytics applications. Some common uses of PostgreSQL are: a robust database in the LAPP stack (stands for Linux, Apache, PostgreSQL and PHP) that powers many

robust and dynamic web applications, used by many large companies as well as start ups as primary database and also as a support for geospatial databases for geographic information systems (GIS) by using it with the PostGIS extension[1].

Understanding now how the PostgreSQL was developed, first we need to mention the POSTGRES package where the PostgreSQL derives from, and that was developed at the University of California Berkeley. Led by Professor Michael Stonebraker, the implementation of POSTGRES began in 1986, and during its first years it suffered some changes on the rule system and some additions, in particular, support for multiple storage managers and an improved query executor. During this time, POSTGRES was used in multiple research and production applications, like financial data analysis system, asteroid tracking database, between many others. In 1993, the amount of users that used this prototype almost doubled, and the amount of time that was needed for the maintenance and support of it took an enoumous amount of time, so Berkeley POSTGRES project stopped in Version 4.2[2].

In the year 1994, Andrew Yu and Jolly Chen created Postgres95 by adding an SQL language interpreter to POSTGRES. It was released as an open-source based on the original POSTGRES Berkeley code. Postgres95 showed clear internal improvements running 30% to 50% faster compared to POSTGRES, and apart from this it also received some addition, changes and removals, in particular: the addition of the SQL language, support for the GROUP BY clause and a new program that used GNU Readline (psql) in order for interactive SQL queries to be used and the removal of inversion file system and the instance-level rule system. It also was provided a short tutorial in SQL features as well as Postgre95[2].

In 1996, the name was changed to PostgreSQL in order to give to the Data Base System a name that would survive future times. The priority of PostgreSQL is not solving problems from the source code like in Postgre95 but is now more focused on the augmentation of features and capabilities, although all fields of work still receive constant look and understanding[2].

With more than three decades of constant developing and understanding of problems and the source code, PostgreSQL can be considered the most advanced open-source Data Base System available in the world.

2 File Structure and Storage

To better understand how PostgreSQL stores and manages files, we first need to understand how the databases and tables are created as well as the architecture of the process and memory of this Data Base System.

2.1 Logical and Physical Structure of a Database Cluster

A PostgreSQL server manages a group of databases. This is called a database cluster. This does not mean that PostgreSQL has a group of database servers, since it runs on a single host and is managed by a single database cluster. To better understand this term of database cluster, we will explain the logical structure of it. A database is a group of database objects, and a database object is no more than a data structure that can be used either to store or reference to data. All database object belongs to their own database, be them tables, indexes, etc. All this objects are managed inside PostgreSQL using their respective object identifiers (OIDs). The relations between this OIDs and database objects are stored in a system catalog. In summary, a database cluster is one directory, that is usually refered as the base directory, that contains some subdirectories that can also have several files. Also, while PostgreSQL can have table-spaces, this term is different from other Data Base Systems, since a tablespace in PostgreSQL is one directory that holds some data outside the base directory[4].

2.2 Layouts

As we can see in Fig.1, a database is a subdirectory below the base subdirectory, and all the OIDs will coincide with the respective database directory names. When a table, for example, has a size less than 1GB, it is stored under the base directory where it belongs. This database objects

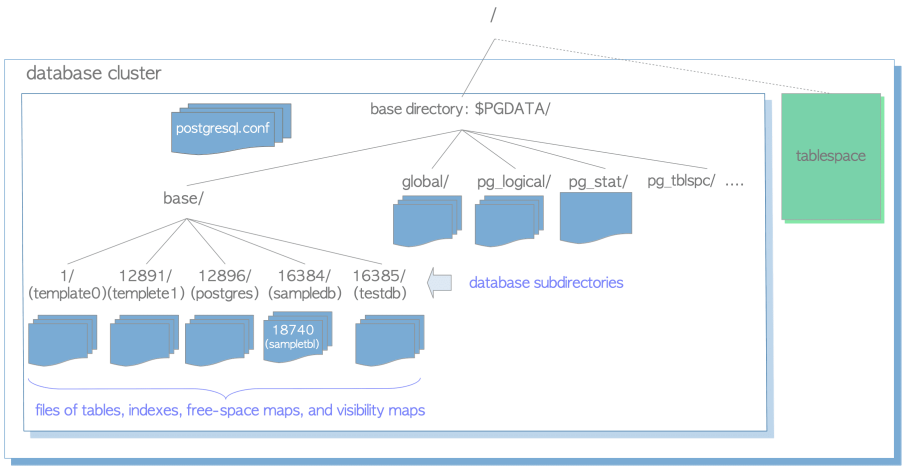

Fig. 1. Example of database cluster[4]

are internally controlled by their own individual OIDs, while the files are managed by a variable called refilnode, although the refilnode value does not have to match always the respective OID. When a file is bigger than 1GB, PostgreSQL creates a new file named refilnode.1 for example. If the new file still is exceeded, PostgreSQL creates a new refilnode file, and this continues until the initial file is saved. It is also important to note that the PostgreSQL files are not locked to 1GB, this size can be changed in the configurations of it. In each database subdirectory, each table has two suffixes that refer to free space map and visibility map, that save information about the free space capacity and visibility of each page on the table file. Indexes, that will be explored in the next section, only have individual free space maps but not visibility maps[4].

In Fig.1 we can also see an area that we still did not talked about, the tablespace. This is an area outside of the base directory that stores the structure of tables, indexes, large objects and long data. This is used to organize the data in a database[6].

2.3 Heap Table File

A data file is divided into pages (or blocks) of fixed length that usually is 8KB. These pages are ordered sequentially, starting in 0. When the file has reached its full capacity, PostgreSQL adds at the end of the file a new empty block and increases the file size. A page of a table has three kinds of data: a heap tuple, that is the data itself, a line pointer, that work as indexes to the tuples and header data, that is allocated at the beginning of a page and has information about the page. The space between the end of line pointers and the most recent tuple is called free space.

2.4 Buffer Manager

We will now talk about the buffer manager of PostgreSQL. A buffer manager is responsible for managing the data transfers between shared memory and persistent storage.

Buffer Manager Structure The buffer manager of PostgreSQL consists of a buffer table, buffer descriptions and a buffer pool. The buffer pool is responsible for the storage of blocks, and is usually an array where each slot stores one page of data file[7].

Buffer Tag PostgreSQL has for each block of all data files a unique tag, that when the buffer manager receives a task, PostgreSQL uses this buffer tag of the desired page. It has three values: the RelFileNode, the fork number of the belonging page and the block number of the desired page. The fork has the number 0, 1 and 2 that assigns to tables, freespace maps and visibility maps respectively[7].

How pages are read from buffer manager? When a table, for example, is read, the backend sends via request that contains the page's buffer tag and buffer manager. Then, the buffer manager returns the buffer ID of the place where the requested page is stored. If the page that was requested is not stored in the buffer, the buffer manager loads it from

persistent memory to the buffer and returns its buffer ID. To end, the backend process uses the buffer ID in order to read the page that was requested. When a process modifies a page in the buffer, the page that still was not removed is referred as dirty page[7].

How does the page replacement algorithm works? When the buffer is full and the requested page is not in it, the buffer manager needs to select one page to be replaced. PostgreSQL for this task uses an algorithm called clock sweep, since it has been proven to be more simpler and efficient that the usual LRU. What this algorithm does is, it keeps all pages on a circular list in the form of a clock, and a pointer is placed in the oldest page. When a page fault happens, the page where the pointer is placed is inspected. If its R is 0, then the page is removed and the new page is inserted in that place. If it is 1, then the page is good and the iterator moves to the next page. This process continues until a page with R equal to 0 is found. Although it is very similar to the second chance algorithm, it changes in the implementation[8].

How does the buffer manager flushes dirty pages? Dirty pages need to be removed from the buffer, but in order to the buffer manager to perform this, it needs the help of other two processes, the checkpoint and background writer, that will be later analysed in this section[7].

Buffer Manager Structure PostgreSQL buffer manager has three different layers to it: buffer table, buffer descriptions and buffer pool. The buffer pool is essentially an array where a data file page is stored in each slot. The buffer description is an array of buffer descriptions. The way it works is that each description is linked to one slot in the buffer pool where the data is stored. The buffer table is a hash table where all the relations between buffer tags and buffer ids are hold[7].

Buffer Manager Locks Buffer manager also has integrated many locks in order to mainly keep the integrity of the data when is being accessed or changed[7].

Flushing Dirty Pages As mentioned previously, the buffer manager in PostgreSQL has two process to remove dirty pages: checkpointer and background writer. What the checkpoint process does is, it writes a checkpoint record in the WAL (Write ahead block) segment file and flushes the dirty pages right when the checkpointing begins. On the other hand, the background writer role is to reduce the influence of eventual intensive writing of the checkpoint process. It continues to delete dirty pages but in small amounts in order to maintain or minimize the impact on the database activity[7].

2.5 Differences and similarities of PostgreSQL and Oracle 18c on this topic

Before taking a look in the differences and similarities of both Database Management Systems, we have to remember that PostgreSQL is Open-Sourced and Oracle is Proprietary. This has an impact on their differences since PostgreSQL is free to download and any developer can help on the software's functionality and correction of bugs, while having a vast comunity-based development resources that can be useful for unique businesses while Oracle is licensed, maybe offering less flexibility in resources but compensating in other aspects such as premium support, stability, documentation resources, enterprise-grade security, etc[11].

But apart from that, this two DBMS have a lot of similarities. One of them is the fact that each one of them has their own buffer manager, not relying because of that on the operating system to access and maintain order when requests into pages are done.

The main advantage in using Oracle instead of PostgreSQL for storing files is its security. This is because when files are stored in the database, they are backed up, synchronized to the disaster recovery site using Data Guard. But PostgreSQL, since it is open source as mentioned above, offers multiple extensions that when are currently used, can outperform Oracle in many ways. But again, it is important to reinforce that they still have many similarities, and that may be the reason why they are two of the most used Database Management Systems in the world.

3 Indexing and Hashing

PostgreSQL indexes are a unique database object where its main function is to speed up the data access. There are some auxiliary structures such as the fact that each index can be removed and rebuilt just from the information in the table. Although multiple indexes have differences, all of them will be associated to a key with table rows that relate to this key. Each row will have its own tuple id in order for it to be identified[13].

Similar to Oracle, PostgreSQL uses the B-tree index type by default. To use other index types, the keyword USING followed by the index type name needs to be placed in the query.

It is possible also to define an index on more than one table. This is also called Multicolumn Indexes. With the current version, only the B-tree, GiST, GIN and BRIN index types support this feature.

PostgreSQl also has the ability to combine multiple indexes in order to handle cases that cannot be done by a single index scan. This can be done by adding the forms AND and OR across several indexes. To do this, the system scans each index and prepares a bitmap in memory that gives the locations of the table rows that match the index conditions. Another property of indexes in PostgreSQL is the unique indexes, that is still only available with B-tree indexes. What this does is that it does not allow table rows with equal indexed values.

All the indexes in PostgreSQL are secondary indexes. What this means is that each index is stored separately from where the table's main data is and in an ordinary index scan, each row returned will require a fetch of data from both the index and the heap. Other aspect about this is the fact that index entries that match a given condition will usually be close together in the index, while the table rows it references might be anywhere in the heap and the heap access of an index scan involves many random access, and this can be really slow. To solve this, PostgreSQL also implements index-only scans, which can get the results of queries from an index without the need of any heap access. The rules for this to be done are: The index type must support this methodology such as B-trees, and queries must reference only columns that are indexed. If this clauses are

met, then an index-only scan is physically possible. In order for this to be effective, creating covering indexes might help. A covering index is an index that includes the columns needed by a type of query that you run very frequently.

3.1 Specific Types of Indexes

Hash Indices Before analysing how does Hash Indices work in PostgreSQL, lets look at the idea of hashing. The main idea of hashing is to associate a small identifier with any type of data. We do this association by using an hash function. The number returned from the hash function can be used as an index of an array where tuple ids will be stored. The components of this array are called hash table buckets and one bucket can store more than one tuple ids if the same index is given to more that one row. So its easy to understand that the better an hash function distributes the rows, the better it will be to access it.

Now that we have the main idea of how hashing works, lets focus on how hash indices work in PostgreSQL. So when we insert into the index, the hash function will compute it for the key. In PostgreSQL, hash functions always return a "integer" type with range of around 4 billion values. So the number of buckets that are available in the beginning is equal and is dynamically increased if the data size is increased. But this is not sufficient since tuple ids that match different tuples can be placed in the same bucket, along with the tuple id. This would increase the index size by some margin, so what is done is, instead of the key, the hash code of the key is stored in the bucket. While searching the index, first the hash function is computed in order to get the bucket number. Now that we have the bucket number, the only thing remaining is going through all the contents of the bucket to return the tuple id, but this is done in a very efficient way since all the hash codes with the tuple ids are stored in an ordered way. However, it is still possible for two different keys to be in the same bucket with the same hash code. This is solved with the accessing method asking the general indexing engine to look into each tuple id while rechecking the condition on the table row[16].

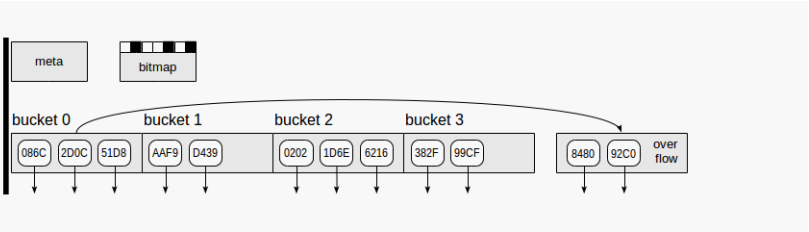

Fig. 2. Hash index [16]

Looking from the perspective of the buffer cache manager to the indices, it turns out that the information and the index rows look packed in pages. This index pages are stored in the buffer cache and removed from there the same way it is done with pages. In Fig.2, we can see that Hash Index shows four different types of pages: Meta Pages(it is the page with the number 0 that contains all the information of what if in the index), Bucket Pages(The pages where the hash code with the tuple ids are stored), Overflow Pages(Same type of the Bucket Pages but are only used when one page is insufficient for a bucket) and Bitmap Pages(That keep order of the overflow pages, knowing which ones are free to be used or the ones that can be reused by other buckets). The down arrows that are in each index page element represent tuple ids. When an index is increased, PostgreSQL creates at the same time the double amount of the current buckets, and to avoid allocation of a large number of pages at the same time, in version 10 was implemented that this increase in size is done in a more smooth way[16].

B-tree Indices PostgreSQL also has implemented the B-tree and this method is good for data that can be sorted. As in the Hash Index, rows in the B-tree are also packed into pages. The leaf pages contain the data that is indexed and the references to the tuple ids. The internal pages have references to a child page of the index and also contain the minimal value of this page. Some important aspects of the B-tree are:

– They are balanced, which means that the leaf pages are all separated from the root by the same number of internal nodes, therefore, search time is the same for all values;
– B-trees are multi-branched, which means that each page, that is usually of size 8KB, contains multiple tuple ids. Because of this, the depth of a B-tree is always small, usually of depth 4-5 to bigger tables;
– All the data in the index is sorted, and pages on the same depth value are connected in a bidirectional way. This way, it is possible to get an ordered data set just by walking from one direction to another without needing to return to the root node[17];

GiST Indices Generalized Search Tree, or GiST, is a balanced search tree like B-tree. The difference between them is that B-tree indices are connected to the comparison semantics. But in the days we live in, databases got to a point where the data and these operators do not make much sense (data like geodata, text documents, images, etc.).

The GiST index comes in to take care of these types of data. It enables the definition of rules to distribute data of an arbitrary type into a balanced tree, and also a method to make use of this representation.

In GiST, each leaf node contains some boolean expression and a tuple id, and the indexed data must meet this predicate. Each internal node also contains a boolean expression and a reference to a child node, and all the data of the child must also meet this boolean expression. This last feature of GiST index is what changes comparing to the simple ordering of the B-tree.

Search in GiST is done by using a specialized consistency function that is defined by the interface and implemented in its own proper way for each of the supported operators family. This function is called for an index and is the function that determines if the whether the boolean expression of that row is consistent with the search predicate. In an internal row, is also this function that determines if it should go to the corresponding subtree or leaf row. The search algorithm starts in the root node, and computes the consistency function to find out which child nodes make sense to enter

and which ones it does not. The algorithm will probably find more than one node that makes sense to enter so the algorithm is repeated for each one of them, and if the node is a leaf node, then the row is selected by the consistency function and it is returned as one of the results. This search is depth-first, so the algorithm always tries to reach the leaf node first, and this is good since the goal is to return results as soon as possible. It might be important to note again that the consistency functions does not guide itself by operators like "equal" or "bigger", so the order of the returned results is not important.

Operations like insertions and deletions are done by other interface functions, but whenever a new value is inserted into the index, its position is chosen in a way that the boolean expression of its parent is extended as little as possible. However, when a value is removed, the parent row is not reduced. This only happens in specific cases such as when a page is divided into two, or the index is recreated from scratch. We will not go further into the indexes of various types, but it is possible to create indexes to data such as points and other geometric entities, intervals and exclusion constraints and full-text search[19].

SP-GiST Indices SP-Gist may have the same name as GiST, and that is since both of them are generalized search trees that the goal is to provide with a framework for building various access methods. The "SP" part stands for space partitioning, and what it means is exactly as the name suggests, space, like for example, in a two-dimensional plane. This index is useful especially for structures where the space can be recursively split into non-intersecting areas. So in summary, the idea of it is to split the value into non-overlapping subdomains each of which can also be split, and partitioning like this leads to non-balanced trees.

The fact that it is non-intersecting helps in the decision making during the procession of insertion and search. A rule that is also important to note is that the trees induced are low branching. A good example is the quadtrees that usually have 4 children and larger depth. Trees like this example suit well the work of the RAM, but the index is stored in the disk

and because of that, in order to reduce the number of I/O operations, the nodes will need to be packed into pages and this is not easy to do in an efficient way. Apart from that, the time that it takes to get a different value of the index may be different for other branch depths.

The internal node of the SP-GiST tree saves references to a child node by defining a label for each reference. Besides, an internal node can also store a value that can be regarded as an arbitrary predicate that all child nodes can meet.

Leaf nodes of this index contain the values of the indexed type to a reference to the tuple id. The search key of it can also be used as the value, but not obligatory. Adding to that, leaf nodes can also be grouped into lists, so internal nodes not only can reference to a single leaf node, but to a list of nodes.

Similar to GiST, the main search function is the consistency function. The way it works is very similar with the consistency function to GiST. At the physical level of it, index nodes are grouped into pages to make the work with nodes more efficient from the point of view of I/O operations. It may also important to note that one page can contain internal nodes or leaf nodes, but never both[21].

GIN Indices Generalized Inverted Index, or in abbreviation, Gin uses this so-called inverted index that manipulates data types of values that are not atomic, but instead, consist of elements. During this study, we will call these types compound. To note that these are not the values that we index. We index the individual elements, and each element is related to a value when it occurs.

To better understand this concept, we will make an analogy. Imagine the index is placed at the end of a book, and for each term, there is provided a list with the pages where that term appears. The access must be fast, so for that, these elements are stored in a B-tree in an sorted set of references to table rows that will contain the compound values of each element linked to other elements. Order here is not essential for the return value, but is important to keep the internal structure of the index.

In GIN index, elements are never deleted. The values that can contain an element can disappear or vary, but the set of elements that is built is more or less the same. This algorithm makes it easier algorithms for concurrent work of several processes with the index.

If the list of tuple ids would be short, it would be possible to fit into the same page of the element. But when the list is big, a more efficient data structure would be better, and that is why it is implemented using a B-tree.

In summary, GIN is a B-tree of elements, and B-trees or flat lists of tuple ids are connected to leaf rows of that B-tree. Just like GiST and SP-GiST indexes that were discussed above, GIN index provides an application developer with the interface in order to help with operations over compound data types[23].

RUM Indices RUM method extends the concepts of GIN by allowing us to perform text search in a more faster way. But what limitations does GIN have that RUM allows us to do? First thing is the fact that "tsvector" data type has not only lexemes, but also information in the it position inside he document. GIN index is not able to store this information. For this exact reason, operation that search for phrases, which only appeared with version 9.6 of PostgreSQL by GIN index, are not very efficient and to have the original data for recheck. Other aspect is the fact that search systems usually return results sorted by relevance. We can then use functions such as "ts rank" and "ts rank cd" to do this, although they need to be computed for each row of the result, which can be really slow.

In summary, RUM access method can be considered the same as GIN but that additionally stores position information and can give back results in a needed order[25].

BRIN Indices Unlike the indices we talked above, the idea around BRIN is to avoid looking through definitely unsuited rows rather than finding the results in a quick way. This is an inaccurate index since it does not contain tuple ids at all.

In a simple way, BRIN works good with columns where values have a correlation with their own physical location in the table. Saying in other words, if a query without ORDER BY clause returns the column values through a virtual way in increasing or decreasing order.

This method of accessing was created in scope of Axle, a European project that used huge analytical databases, with an eye on tables that are several terabyte large. An important feature about this method is that allows us to build indexes on such tables is a small size and minimal overhead costs of maintenance.

The way it works is that the table is divided into ranges that are many pages large and the index stores summary information on the data of each range. A rule is that this is the minimal and maximal values, but it happens to be different. Assuming a query that contains a condition for a column. If the sought values do not get into the interval, all the range is skipped, but if they get, all rows in all blocks will have to be looked through to decide the matching ones.

BRIN is then not an index, but an acceleration of a sequential scan. We can regard BRIN as an alternative to partitioning if we consider a range to be a virtual partition[26].

BLOOM Indices Bloom filter is a data structure that allows users to quickly check membership of an element in a set. This filter is compact, but allows false positives, which means that can mistakenly consider an element to be part of a set (false positive), but does not allow to consider an element of a set not to be a member (false negative).

Bloom is an array of m bits that in the beginning is filled with zeros. Various k hash functions are chosen to map any element of the set into k bits of the array. In order to add an element to the set, we have to set each bits into the array of m bits to one. Then, if all bits that relate to an element are set to one, the element is can be a member of the set, although if it is at least one bit equal to zero, then the element is not sure to be in the set.

We have N separate filters built for each row index and as a rule, many fields are included in the index. It's values of the mentioned fields are the ones that compose the set of element on each row.

When we choose the length of the array of m bits to be size m, we can find a trade-off between probability of the false positives and the index size. The are of application of Bloom index is quite big, considering wide tables to be queried using the filters on each field. This access method, just like BRIN, can be called an accelerator of the sequential scan, since all the matches fount in any index must be reviewed with the table, but the is for sure a chance to avoid it when we look at most rows at all[27].

3.2 Differences and similarities of PostgreSQL and Oracle 18c on this topic

Again, the two Database Management Systems have many similarities in this topic, sharing the implementation of many index types. But where PostgreSQL really shines is with the indexes that manage data such as points, images, maps, ect. The algorithms and extensions are so refined and frequently receiving updates, that for companies that have large data to store, or data types as mentioned previously, using PostgreSQL really have advantages since they have this indexes to work properly with this types of data.

4 Processing and Query Optimization

4.1 Query Processing

In PostgreSQL, to process a desired query, the query has to pass a sequence of stages to obtain a result. These stages are:

1. Connection to PostgreSQL The first stage of processing a query is to establish a connection from the application to the PostgreSQL server. This connection is essential, for the program to be able to transmit the query to the server and also receive the results from the server.

In PostgreSQL, each client process connects to precisely one back end process. This type of model is called "process per user" client/server and it is assured by the *postmaster*, which creates a new back end process every time a connection is requested. In order to guarantee data integrity, the back end processes communicate to other processes and with each other using semaphores and shared memory. If the connection is successfully established, the client can send the query to the back end process, that will parse the query, create an execution plan, execute the plan and retrieve to the client the desired output through the same connection.

2. Parser stage This stage can be divided into two different sub stages: the parser and the transformation process.

The parser receives the query as plain text and checks if the syntax is correct. If the syntax is not valid, an error is returned, otherwise, a parse tree is created and handed back. The parser is split into two components. To check the syntax, a lexer is used to generate tokens for every SQL key word of the query and it is defined in the file *scan.l*. To build the parse tree, a set of actions and rules, written in C language, are defined in the file *gram.y*. Both the files are transformed into C source files, and with a normal C compiler, the parser is created.

After the parser process is finished, the parse tree is handed to the transformation process. This process will do the semantic interpretation with the aim of understanding which tables, operators and functions are referenced by the query. The result of this activity is what it is called a query tree.

3. Rewrite system Using the query tree handed by the previous stage, this process looks for rules, stored in the system catalogs, and performs transformations to the tree based on these rules. This activity can be useful when the desired query involves a *view*, since it will rewrite the query to access the base table in the view instead.

4. Query Optimizer This system takes the rewritten query tree and builds a query plan to output to the executor. This finished plan consists

in the union of sequential or index scan of the relations, with any of the necessary join strategies (nested-loop, merge, or hash join) plus any supplementary steps needed, such as sort or aggregate-function nodes.

This stage of query processing is one of the most important, since this is the stage where the least time possible to execute the query is selected, and that is the type of information which is usually more relevant to the end users, so the information about it will be presented with more detail in section 4.2.

5. Executor The executor process uses the plan handed by the optimizer, and recursively processes it to output the required set of rows. This is done by a demand-pull pipeline mechanism, with every time a plan is called, it will deliver one more row or report that is finished outputting rows.

4.2 Query Planner/Optimizer

PostgreSQL database system uses a cost-based query optimizer to determine an optimal execution plan for all the different type of queries, examining the possible options and selecting the plan that is expected to execute the fastest while producing the same set of results.

To optimize a given query, the first thing done by the optimizer is to generate the different plans to scan each relation used in the query. The possible plans may consist of sequential scans, different types of available indexes scans, and unions of the previous scans. When the query makes use of join operators, the number of possible executing plans grows exponentially with the number of joins in the query, so it is indispensable for the database system to have high performing join algorithms available for distinct queries that produce distinct plans, and because of that, we will first discuss the join techniques used by this database system. The three available join strategies on PostgreSQL are:

– **Nested loop join**: The simplest join technique. Consists on scanning the entire right (inner) relation once for every row in the left (outer)

relation. Can consume a lot of time, but if there is an index available on the inner relation, it can be the most adequate strategy to perform under some queries;

- **Merge join**: Both relations need to be sorted on the join attribute prior to the join starts. The sorting can be attained by an explicit sort step or by a index scan on the join algorithm (if available). With both relations sorted, both are scanned in parallel and matching rows are combined. This strategy can be really effective since each relation is only scanned once;
- **Hash join**: The inner relation is scanned, partitioned using an hash function on the join attributes and loaded into an hash index. Then the outer relation is scanned and for every row of it, the attributes are hashed to determine matching rows in the relations. This algorithm may be used in cases the relations are not sorted and the sorted process would not be effective, therefore the merge join would not be effective either;

If the query involves a join of more than two relations, the planner examines different join sequences with different combinations of join algorithms to find the cheapest plan.

In PostgreSQL to specifically use a desired join order in a query, the query must be re-written to have the *JOIN* statement between each relation that is supposed to join. An example query for this is: select ... from x JOIN y ... JOIN z JOIN w JOIN c ...;

If the user desires to force one of the join strategies, it is necessary to disable the others available join strategies and enable the one desired. For example, if the user wants to force the use of the Hash join technique, he must execute the following statements:

```
SET enable_hashjoin on;
SET enable_nestloop off;
SET enable_mergejoin off;
```

This process is much simpler in Oracle 18c. In order to force a specific join technique it is only necessary to add a comment of the type /*+ use <join technique > */ after the SELECT statement.

For queries that do not use much relations, a near-exhaustive search is performed to find the best join order, but for queries involving more tables, that is not the case, because exhaustive searching on many tables would take too long to execute, often longer than the penalty of executing a sub-optimal plan. For this type of cases, PostgreSQL uses *GEQO*.

GEQO means Genetic Query Optimizer and it is an algorithm that makes use of heuristic searching on query planning to find the best join sequence possible. With *GEQO* it is possible to define the *geqo_threshold*, which defines a threshold for the number of joins on a query to use exhaustive search, and after that number, to start using *GEQO* instead. This threshold's default number is 12. To generate possible plans, *GEQO* generate plans for scans of individual relations, using the standard planner, and join plans are developed using the genetic approach, generating possible join orders at random. Then for each random join order, the standard planner code estimates the cost of performing the query in the specific order, considering the three possible join strategies. The chains of joins with the lower estimated cost are considered the more adequate, and after that, are combined with other random adequate sequences. This process is repeated until a preset number of join sequences have been examined and the best join order is used to generate the finished plan.

Despite the fact that *GEQO* is really useful since it reduces planning time for complex queries, the developers of PostgreSQL consider that there is some work left to be done to improve the genetic algorithm.

Although joins are the operation that needs more importance when it comes to optimizing, it is also crucial to discuss how, another relevant operation, the *SELECT* operation, is processed and optimized in PostgreSQL and how the different clauses that can be added to it are also processed.

SELECT retrieves rows from zero or more tables and its processing can generally be divided in the following stages[5]:

1. Firstly all the queries in the *WITH* are computed and will serve as temporary tables that can be referenced in the *FROM* list. Usually, a *WITH* query is materialized, unless specified otherwise with NOT MATERIALIZED. Materialize is the process of storing subselect operations that may need to be computed more than once;
2. All elements in the *FROM* list are computed, scanning the relations in case. The scan can be done using the Sequential Scan operator or the Index Scan according to the desired attributes and other operations of the query, specially the *WHERE* clause next described;
3. If the *WHERE* clause is specified, the rows that do not satisfy the desired conditions are eliminated. If using a Sequential scan to scan the relations, for each row of the table, the query constraints are evaluated, and if they are satisfied, the required columns are added to the result set. When using an Index Scan, an index structure must be available to transverse in order to retrieve the expected columns. An Index scan can be really convenient when the query provides a starting and/or ending value for the attribute of the index or when the result is desired to be ordered by the index order;
4. If the *GROUP BY* clause is specified or there are aggregate function calls present in the query, the output is combined into groups of rows that match on one or more values, and the results of aggregate functions are computed;
5. In this stage, the actual output rows are computed using the *SELECT* output expressions for each selected row or row group. The next stages are processed after the the output is produced;
6. If the the selection uses *DISTINCT*, the duplicate rows are eliminated from the results or if it uses *DISTINCT ON* rows that match a specific expression are eliminated as well;
7. When using operators such as *UNION*, *INTERSECT* OR *EXCEPT*, the output of more than one SELECT statement can be combined to form a single result set;
8. In the cases the *ORDER BY* clause is specified, the rows returned are sorted in the desired order. If this clause is not present in the query,

the rows are returned in whatever order the system finds fastest to produce;

9. If a *LIMIT* or *OFFSET* clause is present in the query, only a subset of the result rows are returned.

To handle complex expressions, PostgreSQL makes use of materialization, in fact, it is impossible to disable materialization on this system. However it is possible to disable the query planner's use of materialization, using the statement: SET enable_material on.

4.3 Execution plan

In order to visualize a query's execution plan made by the planner it is possible to run the command *EXPLAIN* before the desired query. This command displays the execution plan that the optimizer generates for the supplied statement, showing how the referenced table(s) will be scanned and the join algorithms that are being used in case there are multiple relations in the query, to bring together the required rows from each input table. The command mentioned does not execute the query, it only plans the execution. If it is desired to actual run the query it is possible to add the parameter *ANALYZE* between the *EXPLAIN* command and the query.

As explained in the last section, the most important part to display is the estimated query execution cost since this is the relevant information to the user perspective, but this information would not make sense to display if the structure of the plan would not be displayed too.

The structure of the query plan is a tree of plan nodes. Plan nodes at the lower level of the tree are scan nodes, and as mentioned previously there are three type of scans: sequential, index and bitmap index scans. If the query requires joining, aggregation, sorting or another type of operations that execute on the raw rows, additional nodes would also be displayed above the scan nodes to dispose how these nodes perform. For each operation node, generated by the optimizer, it will be displayed 4 different numbers. The first one is the estimated start-up cost and this is the time elapsed before the output stage can begin. This time can be the

time to do the sorting process in a sort node. The second is the estimated total cost which is the time estimated to complete retrieving rows from a query. The third one is the estimated number of rows output by the plan. And finally, the fourth is the estimated average bytes of the retrieved rows.

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

Fig. 3. Example query and respective execution plan

4.4 Statistics

As previously explained, the query optimizer needs to estimate the number of rows that the query will output in order to make the best choice for the execution plans and to do these estimations, the PostgreSQL system utilizes statistics.

The main and most important components for reliable statistics are the total number of entries in each table and index as well as the number of disk blocks occupied by each table and index. These components are store in *pg_class*, which is a system catalog table. It is possible to query this table to output the components mentioned, and the main columns of this table are the *relname*, *reltuples* and *relpages*, with are the name of the each relation, the tuples of each relation, and the pages occupied by each relation respectively. These are not automatically updated and for that purpose it is necessary to execute the *ANALYZE* operation, however the

optimizer will scale the values it finds in this table to match the current physical table size, so closer approximations are possible.

Another relevant system catalog table is the *pg_stats*, which stores the statistics about the queries that retrieve only a fraction of the rows in a table, due to WHERE clauses that restrict the rows to be examined. For this kind of selectivity, the optimizer is needed to estimate the fraction of the rows that match the desired condition. This table can be updated using the *ANALYZE* operation and the statistics are always up-to-date.

When queries involve multiple columns that are correlated, this is, that are not independent, the regular statistics are not enough to capture this type of cross-column correlation. For that cases, PostgreSQL uses Extend Statistics. These can compute the statistics automatically of very large column combinations. To create this type of statistic, that are often called statistics objects, the command *CREATE STATISTICS* creates a catalog entry expressing interest in the statistics and, when combined with the *ANALYZE* operation, the actual data collection is performed. Since the extended statistics sample size is increased by increasing the target for the table, this usually means that the statistics are more accurate but that comes with more time spent on calculating them.

5 Transaction Management and Control of Concurrency

5.1 Transactions

Transactions are a fundamental concept for any database system. A transaction is a unit of program execution that accesses and possibly updates various data items [10], where multiple steps are bundled into a single all-or-nothing operation. Transactions should follow some properties in order for the system to be able to preserve the database integrity. These properties are:

- **Atomicity**: either all operations of the transaction are properly executed or none are;
- **Consistency**: the database's consistency is preserved after the transaction;

– **Isolation**: each transaction and each step of it, must be unaware of other concurrently executing transactions;
– **Durability**: Changes made to the database by a successful transaction must persist on the database, even if there are system failures.

In PostgreSQL, it is possible to set up a transaction by surrounding the desired SQL commands of the transaction with *BEGIN* and *COMMIT*. Actually, in this database system, as well as in other major systems, every SQL statement is executed within a transaction, having an implicit BEGIN, and if successful, a COMMIT statement wrapped around. In this system, it is also possible to control the statements in a transaction with the use of *savepoints*. *Savepoints* are defined inside a transaction block with the command *SAVEPOINT <savepoint name>* and allow, not only the user to precisely discard steps of the transaction, while committing the rest, but also allow the user to roll back to a specific *savepoint*, priorly defined with the command *ROLLBACK TO SAVEPOINT <savepoint name>*. It also offers the option to the user to release a *savepoint* so the system can free some resources, when the user is sure that he will not be using the particular *savepoint* to roll back again. This is done with the command *RELEASE SAVEPOINT <savepoint name>*. In PostgreSQL, this mechanism is considered as support for nested transactions, but it differs from the convectional SQL, since it is more like nested partial points in transactions, whereas in Oracle 18c there is a more traditional support available for nested transactions.

In figure 4, an example is displayed from a transaction block with different commands.

5.2 Concurrency Control

In PostgreSQL data consistency is usually maintained by using a Multi version Concurrency Control (MVCC) since a proper use of it will generally provide the best performance, but there are also some other tools to manage concurrent access to data.

Using MVCC, each SQL statement only sees a database version as it was some time ago regardless of the current state of the information, this

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

Fig. 4. Example of transaction block with several statements [12]

is called a *snapshot*. This way, transaction isolation for each database session is performed, this is, statements are prevented from accessing inconsistent data produced by concurrent transactions effecting the same rows. The use of the MVCC model of concurrency usually brings more performance advantages than locking methodologies because MVCC locks acquired for reading data do not conflict with locks acquired for writing data, and this way writing locks never blocks reading and reading locks never blocks writing. Identical to Oracle 18c, PostgreSQL guarantees this by providing an innovative *Serializable Snapshot Isolation* level (SSI), the strictest level of transaction isolation.

To better understand concurrency control, it is crucial to understand the concept of transaction isolation in detail. The Serializable level was already mentioned as the most strict isolation level, but PostgreSQL defines three more levels: Read uncommitted, Read committed and Repeatable Read. The serializable level emulates serial transaction execution for all committed transactions, this means, that any concurrent execution of transactions is guarantee to produce the same effect as running them sequentially. The other defined in terms of phenomena happening between the interaction of concurrent transactions. These phenomenons must not happen in the serializable level but can use them as examples for studying purposes, and next we are describing them from the least to most strict:

– **Dirty read**: transaction reads data written by a concurrent transaction that was not committed.
– **Nonrepeatable read**: transaction re-reads data it has previously read and detects that data modified by another concurrent transaction.
– **Phantom read**: transaction re-executes a statement that returns a set of rows that satisfy a desired condition and detects that the returned rows have changed due to another recently committed transaction.
– **Serialization anomaly**: the result of succesfully commiting a group of transactions is inconsistent with all possible ordering of running those transactions sequentially.

The SQL standard and PostgreSQL transaction isolation levels are displayed in table 1, to briefly describe how they are implemented and their differences. One important thing to notice is that, in PostgreSQL, although you can request for any of the four isolation levels, the Read uncommitted and the Read Committed levels behave exactly the same way, in fact, internally they are implemented the same way, since the Dirty read phenomena is not possible in the first level opposite as it is in the SQL standard.

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

Table 1. Implementation and differences between SQL standard and PostgreSQL levels of transaction isolation

Focusing in the Serializable Snapshot Isolation level, it is implemented with the Serializable isolation level improved by adding checks for serialization anomalies.

In comparison, Oracle 18c only implements two of the four transaction isolation levels: the Read committed, which is the default level in Oracle and the Serializable level, that is used to implement Snapshot Isolation,

and it is more adequate for larger databases with only few updates that result in few conflicts.

To specify the transaction isolation level, in PostgreSQL, the command *SET TRANSACTION ¡isolation level¿* is available to set the isolation level for a specific transaction block. It is also possible to omit the SET statement, mentioning the isolation level on the BEGIN statement of a transaction block.

5.3 Explicit Locking

PostgreSQL also implements several lock modes to control concurrent access to data when MVCC does not achieve the desired behavior. In addition, most of this database system commands automatically acquire locks, with the adequate modes, to ensure the data is not modified while the process executes. Currently, the lock models, with different levels of granularity, supported by PostgreSQL are:

- Table-level Locks
- Row-level Locks
- Page-level Locks
- Advisory Locks

Table-Level Locks There are available several table-level locks and their differences are the set of lock modes that each conflicts with, and the granularity of each one, this is, the amount of data this is locked at one time. The different lock-modes are:

- **ACCESS SHARE**: Usually queries that only read data from a table acquire this lock mode;
- **ROW SHARE**: Commands like SELECT FOR UPDATE and SELECT FOR acquire this lock;
- **ROW EXCLUSIVE**: This lock mode is acquired for queries that modify a table's data, like INSERT, DELETE and UPDATE;
- **SHARE UPDATE EXCLUSIVE**: Used when performing ANALYZE, CREATE INDEX CONCURRENTLY or CREATE STATISTICS commands;

– **SHARE**: Acquired by CREATE INDEX command;
– **SHARE ROW EXCLUSIVE**: This is lock is acquired performing the CREATE TRIGGER command;
– **EXCLUSIVE**: Acquired by the command REFRESH MATERIALIZED VIEW CONCURRENTLY;
– **ACCESS EXCLUSIVE**: conflicts with all of the lock modes. It is the default lock mode for LOCK TABLE statements that do not specify a lock mode.

In figure 5 the conflicts between table-level locking modes are displayed.

	Existing Lock Mode							
Requested Lock Mode	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCL.					X	X	X	X
SHARE UPDATE EXCL.				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCL.			X	X	X	X	X	X
EXCL.		X	X	X	X	X	X	X
ACCESS EXCL.	X	X	X	X	X	X	X	X

Fig. 5. Conflicts between modes of table-level locks [15]

A table-level lock usually is held until the end of the transaction or can be release during a *savepoint* rollback.

In PostgreSQL it is possible to perform a specific lock on table-level, the command *LOCK TABLE <table name> IN <lockmode>*, where lockmode can be ROW EXCLUSIVE for example.

In Oracle 18c, the explicit locking of tables can only be done with *row share mode*, *row exclusive mode*, *share mode*, *share row exclusive mode* or *exclusive mode*.

Row-Level Locks Row-level locks are also possible in PostgreSQL. This type of locks do not affect data querying, they only block writers and lockers to the same row. Row-level locks are released the same way.

Row-level locking can be performed using 4 different modes in this database system. In table 2, the conflicts for row-level lock models are displayed.

Requested Lock Mode /Current 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 2. Conflicts between row-level lock modes

To specifically use this type of locking level on PostgreSQL, firstly it is necessary to lock the table(s) where the user wants to perform the row-level lock. This is done by locking the table as explained for table-level locks. Then the specific row-level lock statement is added to the end of the query that will return the desired row(s). For example, this can be done the following way: SELECT .... FROM .... WHERE ... FOR UPDATE (could be any other row-level lock model);

In Oracle 18c, row-level locking is the default locking technique.

Page-Level Locks Page-level share/exclusive locks are also possible in PostgreSQL. These are used to control the read and write access to table pages in the shared buffer pool and are released immediately after a row is fetched or updated.

Advisory Locks Advisory Locks are also provided by PostgreSQL, and these are useful to create locks that have application-defined meanings, this means that is up to the application to use them correctly. These can be acquire at session level or transaction level.

5.4 Deadlocks

It is not possible to discuss about lock modes without explaining deadlocks, since the use of locks can highly increase its likelihood. Deadlocks

happen for example, when one transaction acquires an exclusive lock on a table 1 and then tries to acquire another exclusive lock on another table 2 while another transaction already has an exclusive lock on table 2 and wants to acquire a lock on table 1, this way, none of the transactions is able to proceed and it would wait for ever.

PostgreSQL automatically detects deadlocks situations and resolves them by aborting one of the transactions involved, allowing the other(s) to complete. It is important to mention that deadlocks can occur even if explicit locking is used.

The best defense against deadlocks is generally to avoid them but there is no magic configuration to do that. To avoid them, it is necessary to be certain that all the applications using the database acquire locks on multiple objects in a consistent order but it this is not feasible to verify in advance, that is why deadlocks are handled dynamically by retrying transactions that abort due to deadlocks.

5.5 Index Locking

PostgreSQL provides nonblocking read and write access to table data using some of the index discussed previously in this document. These indexes and how they are handled is next described:

- **B-tree, GiST and SP-GiST**: handled with short-term share/exclusive page-level locks are used for read and write access.
- **Hash indexes**: share/exclusive hash-bucket-level locks are used for read and write access.
- **GIN indexes**: short-term share/exclusive page-level locks are used for read and write access.

Currently, B-tree indexes present the best performance for concurrent applications that need to index scalar data.

6 Support for Distributed Database

Distributed databases are group of databases distributed along different locations (often called nodes or sites) that communicate with each other

constructing a network able to store and respond to users' necessities. The main advantages of using this architecture are the higher availability and data stability (because it is possible to maintain replicate versions of the same relations), higher response times (due to the improved query parallelism using the distributed nodes) and geographic data storing choices (relations can be fragmented according to statistics that describe where it is more accessed). In contrast it can be more difficult to update data bases because it is necessary to comply with ACID properties, this is the data needs to be reliable all the time, e.g. updates on a given node should reflect at the same time on all nodes. As implied in the above sentences there are two possible ways to distribute the data by the sites: replication or fragmentation, but they are often used together to improved efficiency.

PostgreSQL is a Database Management System highly appropriated for distributed database architectures because its features are very useful to their construction. Some of these characteristics are [3]:

- **libpq** is a C application terminal that offers a library of functions to contact with a back end Postgre server asking queries and receiving their results. This is useful because it provides a tightly defined protocol on a connection to another data base
- **Transactions**
- **Streaming replication** and other solutions that will be better explained in the next subsections

6.1 Postgre approach

The main reason why Distributed Databases are important is to have a high availability even with loss of primary server and data balancing through the servers. The biggest obstacle is to achieve a good synchronization, this is be able to maintain all different servers consistent even with write requests being done.

There are some important definitions to understand the different solutions offered: *master server* is the server that can modify the data, *standby server* is the server that listen to changes on the master one, *warm standby server* is a standby server that can only be connected when is promoted

to primary and *hot standby server* which is a server for read-only requests. The varied approaches presented differ on if they are synchronous or asynchronous (all servers just confirm a request when is committed on all of them or not) or granularity, in all of them it is important to define a balance between performance/speed, functionality and reliability.

Some of the solutions presented in the PostgreSQL manual are:

Shared Disk Failover There is a single disk array that is shared by all the servers. This avoids losing time on synchronization because there is only one copy and in case of failure the standby servers can mount the database but if the disk array fails the database becomes nonfunctional.

File System (Block Device) Replication The file system is replicated in all servers and it is important that all of them are equal anytime. This is achieved by maintaining consistency in the mirroring process by making all the writes in the same order.

Synchronous Multimaster Replication Writes requests are accepted on all servers and the data is transmitted from the master to the others before it is committed. Read request can be sent to any server. The biggest advantage is that the writes can be done in any server but big writes can cause locks and delays leading to low speed and consequently bad performance. "PostgreSQL does not offer this type of replication, though PostgreSQL two-phase commit (PREPARE TRANSACTION and COMMIT PREPARED) can be used to implement this in application code or middleware." [Notes] (Since Postgres 8.1 that it is available the two-phase commit feature. This allows one of the database involved in the transaction to be the coordinator and send prepare, ready and commit messages, keeping all the relations updated at the same time in all servers [20]).

6.2 Streaming replication

Streaming replication is the best process to keep the standby servers more updated (there is another technique used that will not be discussed

that is file-based log shipping). The method is simple, the master just send/stream WAL (Write-Ahead log) to the slaves servers without the need for them to be processed. This process is asynchronous so there is possible delays on keeping data consistency between master and standby servers (but the delay is shorter than FBLS). The biggest disadvantage is that if a WAL segment is recycled too early there is the need for the standby server to be reset, however this can be avoided by setting the wal_keep _size bigger or using replications slots.

To achieve a properly functional replication process is important that all the servers that form the database are well organized and named. The primary server needs to use authentication (a feature also available on Postgre servers) to enable connections with the standby servers. The standby servers needs to have a file named *standby.signal*. Libpq is, as mentioned early, the favorite library to use to define the connection protocols.

6.3 Built in extensions

dblink is a built in extension from PostgreSQL to query other local or remote servers. In Figure 6 and 7 it can be observed respectively the use of the main method *dblink* querying a local and a remote server. There is another useful methods to be used like: *dblink open* to open a cursor on a remote database, *dblink get connections* that returns an array with all the open db links connections (can be important to control all the servers local and remotely connected) and *dblink send query* that sends an asynchronous query to a remote database.

All functionalities offered are very similar to Oracle DbLink but it is not yet as rich. Some techniques that can improve it are pipelining (could improve speed on synchronized joins) and introspection (avoiding the need to specify the all output structure) [9].

6.4 Third-party extensions

citus is a third party extension to be used on Postgres to scale out databases distributing data along the cluster, improving the balance be-

```
SELECT towns.*
FROM dblink('dbname=somedb','SELECT town, pop1980 FROM towns')
    AS towns(town varchar(21), pop1980 integer);
```

Fig. 6. Querie another local PostgresSQL DB with DBlink [9]

```
SELECT blockgroups.*
    INTO temp_blockgroups
    FROM dblink('dbname=somedb port=5432
            host=someserver user=someuser password=somepwd',
    'SELECT gid, area, perimeter, state, county,
            tract, blockgroup, block,
            the_geom
            FROM massgis.cens2000blocks')
            AS blockgroups(gid int,area numeric(12,3),
                perimeter numeric(12,3), state char(2),
                county char(3), tract char(6),
                blockgroup char(1),
                block char(4), the_geom geometry);
```

Fig. 7. Querie another remote PostgresSQL DB with DBlink [9]

tween all the nodes and shards. Like Postgres, Citus is a open source tool and is available via GitHub. It delivers a query speed improvement of 20x to 300x using parallelism, more data in memory and data compression and uses a single database for transactional workload reducing infrastructure needs [14]. The choice of using a distributed database always depends on the amount of workload existent,this is, only it is a good one if there are a lot of concurrent accesses and a big amount of data.

To distribute the data with Citus you need to use *create distributed table* with the name of table as an argument, and it will distribute the data table across the nodes in the cluster. To insert data on the nodes you just insert directly on Postgres, the routing for the nodes is made by Citus, that will use internal hash functions to make the shard choice [18].

When querying it becomes more difficult because there some situations to consider: data can live on a single shard or aggregated across them. If the table of the query is not yet distributed it uses the Postgre coordinator to do it, if it targets a single shard, runs with the router executor, otherwise with the real-time executor. Router executor targets the right node and works like a normal PostgresSQL instance. The real time executor parallelize to have a better performance across the nodes with the information needed. An example of it can be seen on Figures 8 and 9 where it is observed respectively the normal query done by the user and the parallel work done by Citus to ask the question for all nodes. Normally the select is done for all the rows and the final aggregation (count in this case) is done by the the coordinator. This is just a simple case of what Citus can handle, and why is is more fast than going through each

tuple of a non distributed table and also a lot more simple to handle for the developer [18].

Fig. 8. Normal query [18]

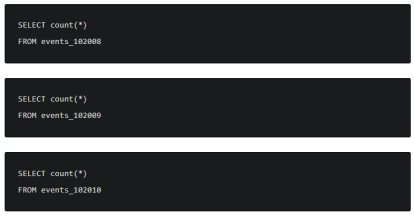

Fig. 9. Citus' behavior [18]

7 Interesting System Characteristics

XML and JSON queries XML is a well known data structure by the companies because of its ease to stream between devices and it can be used to query the databases in Postgre [22]. The advantage of saving values in XML format is because it has a well-formed structure and there are a lot of functions to perform type-safe operations. Difficulties on handling the XML format are the type of encoding used (if it is the same for everything is gonna be a lot faster, often if UTF-8) and the comparison between values (it is necessary to use some tools like text-search or Xpath indexes).

JSON is another data type accepted in Postgre. It offers *json* that stores a text copy and parses it every time when it is needed and *jsonb* that stores in binary format adding a slight overhead on encoding but makes up for it when it is time to process it (a lot faster than parsing a text file). JSON can be used to store values in the database as a field to be inserted or even on the query, e.g. on the where clause. Another main advantage is that jsonb documents can be indexed using GIN indexes.

Spatial Data is the type of data that after analysed is a guide to a geographical location. PostGis is a library that is possible to install on a PostgreSQL database and it provides a lot of functions to work with

geographical data. Some data types it offers are point, line, circle, path (to define a polygon), and box. The point data type is one of the most important because it can be used to define coordinates [24]. A big advantage is the spatial indexing used for spatial queries.

8 Conclusion

PostgreSQL is often called as the world's most advanced Open Source Relational Database and there are many reasons to it: support to varied data types from geometry structures to non-relational data like json or XML, a vast number of extensions that gives the developers a lot of new functionalities to work with, a sophisticated feature of concurrency control (Multi version Concurrency Control) that enables a great boost of performance, it supports full-text search that speeds up searching processes and the most important when using a database a great tool of reliability like streaming replication, a specific feature of replication, that allows standby servers to reconstruct the database if the master server fails [28].

In summary, PostgreSQL is a highly robust database management system that highly complies with the SQL standards, open-source with a lot of support in the community and that makes it highly attractive to new developers. ACID compliant features makes it appealing to businesses that work with transactions a lot, allied with the twelve types of authentication offered. PostgreSQL has been growing at a good pace in the last 20 years with a lot of rich features and a powerful data management and tries to go beyond with the characteristics of other database management systems and that makes it one of the top choices when handling extensively populated databases [29].

Notes

A big portion of the information exposed in this document is based on the documentation available on PostgreSQL official website. All the other used sources are correctly described in the references section.

References

1. What is PostgreSQL? https://www.postgresqltutorial.com/postgresql-getting-started/what-is-postgresql/. Last acessed 1 June 2022
2. A Brief History of PostgreSQL https://www.postgresql.org/docs/current/history.html Last acessed 1 June 2022
3. Conference Sessions 2017 PostgreSQL https://www.postgresql.eu/events/pgconfeu2017/sessions/ Last acessed 2 June 2022
4. Database Cluster, Databases, and Tables https://www.interdb.jp/pg/pgsql01.html Last acessed 2 June 2022
5. Understanding How PostgreSQL Executes a Query http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+4.+Performance/Understand Last accessed 2 June 2022
6. Table spaces https://www.ibm.com/docs/en/db2/11.5?topic=databases-table-spaces Last accessed 2 June 2022
7. Buffer Manager https://www.interdb.jp/pg/pgsql08.html Last accessed 2 June 2022
8. Operating System - Clock Page Replacement Algorithm Long Question Answers https://examradar.com/clock-page-replacement-algorithm-questions-answers/ Last accessed 2 June 2022
9. USING DBLINK TO ACCESS OTHER POSTGRESQL DATABASES AND SERVERS https://www.postgresonline.com/journal/archives/44-Using-DbLink-to-access-other-PostgreSQL-Databases-and-Servers.html Last accessed 2 June 2022
10. Provided Database systems lecture slides Last accessed 2 June 2022
11. PostgreSQL vs Oracle: 6 Critical Differences https://hevodata.com/learn/postgresql-vs-oracle/#introp Last accessed 3 June 2022
12. 3.4. Transactions https://www.postgresql.org/docs/current/tutorial-transactions.html Last accessed 3 June 2022
13. Indexing in PostgreSQL - 1 https://postgrespro.com/blog/pgsql/3994098 Last accessed 3 June 2022
14. Citus - Product https://www.citusdata.com/product Last accessed 3 June 2022
15. 13.3. Explicit Locking https://www.postgresql.org/docs/current/explicit-locking.html Last accessed 3 June 2022
16. Indexes in PostgreSQL - 3(Hash) https://postgrespro.com/blog/pgsql/4161321 Last accessed 3 June 2022
17. Indexes in PostgreSQL - 4(Btree) https://postgrespro.com/blog/pgsql/4161516 Last accessed 3 June 2022
18. How Citus works? https://www.citusdata.com/blog/2017/09/15/how-citus-works/ Last accessed 3 June 2022

19. Indexes in PostgreSQL - 5(GiST) https://postgrespro.com/blog/pgsql/4175817 Last accessed 3 June 2022
20. PostgreSQL Supports Two-Phase Commit https://www.endpointdev.com/blog/2006/05/postgresql-supports-two-phase-commit/ Last accessed 3 June 2022
21. Indexes in PostgreSQL — 6 (SP-GiST) https://habr.com/en/company/postgrespro/blog/446624/ Last accessed 3 June 2022
22. 5 Things You Didn't Know About Postgres https://www.technology.org/2019/07/05/5-things-you-didnt-know-about-postgres/ Last accessed 3 June 2022
23. Indexes in PostgreSQL — 7 (GIN) https://habr.com/en/company/postgrespro/blog/448746/ Last accessed 3 June 2022
24. Spatial Data in PostgreSQL https://www.sqlshack.com/getting-started-with-spatial-data-in-postgresql/ Last accessed 3 June 2022
25. Indexes in PostgreSQL — 8 (RUM) https://postgrespro.com/blog/pgsql/4262305 Last accessed 3 June 2022
26. Indexes in PostgreSQL — 9 (BRIN) https://habr.com/en/company/postgrespro/blog/452900/ Last accessed 3 June 2022
27. Indexes in PostgreSQL — 10 (BLOOM) https://postgrespro.com/blog/pgsql/5967832 Last accessed 3 June 2022
28. Why Postgres? https://fulcrum.rocks/blog/why-use-postgresql-database/ Last acessed 4 June 2022
29. Advatagens of using Postgres https://www.keitaro.com/2021/09/14/advantages-of-using-postgresql/ Last acessed 4 June 2022