



The University of Western Ontario
Department of Computer Science

PostgreSQL and Performance Tuning

Group Members:

Jun Ming Shao-251258566

Yiming Hu-251367779

Jianwu Li-251025696

Alexander Shipley-251113572

Erxun Zhang-251397678

Databases II (Advanced Databases)

Project Report

December 23, 2023

Contents

1	Significance and Objectives	4
2	Methodologies	4
3	Design Principles	4
3.1	Ensure complex objects are supported	4
3.2	Ensure it is easy to extend the DBMS so that it can be used in new application domains	5
3.3	Ensure both rules and active databases are supported	5
3.4	Lessen the size of crash recovery code	5
3.5	Adapt new technologies when available	5
3.6	Try to avoid changing the relational model	5
4	Data and Information Flow	5
4.1	Establishing a Connection	5
4.2	Architecture Overview	6
4.3	Query Evaluation Engine	7
4.3.1	Parsing a Query	7
4.3.2	Rewriting	8
4.3.3	Creating an Execution Plan	8
4.3.4	Executing a Query	10
4.4	Parallel Querying	11
4.5	Storage Access	12
4.5.1	Reading Records	12
4.5.2	Writing Records	12
4.5.3	Buffer Manager	13
4.5.4	Clock Sweep	14
4.5.5	Ring Buffer	15
4.6	Data Partitioning and Foreign Data	15
4.7	Multi-Version Concurrency Control	16
4.8	Background Server Processes	17
4.8.1	Vacuuming	17
4.8.2	Reclaiming Space	18
4.8.3	Updating Statistics	18

4.8.4	Updating the Visibility Map	18
4.8.5	Transaction ID Wraparound	19
4.8.6	Background Writer	19
4.8.7	Write-ahead logging	19
5	Index	20
5.1	B-tree Index	20
5.2	Hash Indexes	20
5.3	GiST and SP-GiST Indexes	21
5.4	BRIN Indexes	21
6	Security	21
6.1	User Authentication	22
6.2	User Names and Groups	23
6.3	Access Control	24
6.4	Functions	24
6.5	Security Case Study	25
7	Query Evaluation	32
7.1	Query Rewriting	32
7.2	Using Index	33
7.2.1	B Tree Index	34
7.2.2	Hash Index	35
7.2.3	Clustered B Tree Index	35
8	Performance Tuning	36
8.1	PostgreSQL Parameters for Tuning	37
8.1.1	Query Evaluation Engine Memory	37
8.1.2	Buffer Size	37
8.1.3	Parallel Query Parameters	37
8.1.4	WAL Configuration	37
8.2	pg_stat	38
8.2.1	pg_stat_statements	38
8.2.2	pg_stat_activity	40
8.2.3	pg_stat_user_functions	40
8.3	pgMustard	40

8.3.1	Annotated Plans	40
8.3.2	Query Examples Analysis	42
8.3.2.1	Inaccurate Row Estimate	42
8.3.2.2	High Evolution Cost	43
8.3.2.3	Poor Disk and Index Usage	43
8.3.2.4	Increasing Parallelism	44
8.3.2.5	Identifying Cache Misses	45
8.3.3	pgMustard Case Study	46
8.4	dbForge Studio for PostgreSQL	49
8.4.1	Query Profiler	49
8.4.1.1	Execution Plan Diagram	50
8.4.1.2	Plan Tree	50
8.4.1.3	Comparison for the Query Profiling Results	51
9	Conclusion	53

1 Significance and Objectives

Michael Stonebraker began a post-Ingres project to address the problems with contemporary database systems during the early 1980s. The project aimed to tackle the complexity of hierarchical and network models at the time, their inefficient disk search mechanisms, and to take advantage of the increasing popularity of SQL and relational models. PostgreSQL itself is a free and open-source O-RDBMS which emphasizes extensibility and SQL compliance. Moreover, it allows for more complex data and types at the expense of deployment complexity and higher maintenance requirements [1].

This report aims to provide a comprehensive analysis of the PostgreSQL system. As well, the report includes practical applications of tuning tools by using the analysis of performance factors in a practical manner. Analysis of the PostgreSQL system and its performance tuning tools is critical for several reasons. A complete understanding of the database architecture, functionalities, and features will aid database administrators in making the optimal decisions. Without a solid understanding of the system, database administrators are unable to leverage the capabilities PostgreSQL has to offer. Performance tuning aims to improve the speed, efficiency, scalability, and overall performance of the system by improving potential faults in processes or data. An under-performing database leads to poor application response times which causes bad user experiences, loss of valuable productivity, and minimizes business revenue.

The report references several tools and techniques to facilitate performance tuning. The features of the tools include identifying productivity bottlenecks, analyzing the effectiveness of existing index configurations, providing actionable tuning recommendations and advice, and generating comprehensive performance reports.

2 Methodologies

The report is comprised of two main components. The first part conducts an analysis of the PostgreSQL architecture and functionality. It studies how data and information flows from disk, memory, and processes when a query is evaluated. From this, the functionalities and features of PostgreSQL will be explored to highlight where performance optimizations are considered. Finally, the last part contains the practical application of the analysis through performance tuning tools.

3 Design Principles

3.1 Ensure complex objects are supported

1. Engineering data has more complexness than business data [2].
2. A relational system may be able to simulate the various data types, but will result in a poor performance [2].

3.2 Ensure it is easy to extend the DBMS so that it can be used in new application domains

1. DBMS should allow for new versions of access methods, operators, and data types [2].
2. Even beginners should be able to implement these new features [2].

3.3 Ensure both rules and active databases are supported

1. Alerters and triggers easily used in applications [2].
2. Alerters can send messages about problems [2].
3. Triggers ensure consistency by forcing updates in a database [2].

3.4 Lessen the size of crash recovery code

1. Disadvantages such as the code being challenging to create, test, and debug, along with special cases being hard to deal with [2].
2. Solution could be to imagine logs as regular data to ease recovery and support accessing historical data [2].

3.5 Adapt new technologies when available

1. Examples include optical disks and large multi-CPU processors [2]:
2. VLSI chips could be made given special hardware is used properly [2].

3.6 Try to avoid changing the relational model

1. Those on the business data side will become accustomed to relational concepts and it is advised to keep it constant [2].
2. Lots of models exist, but not one small model exists that can solve the problems of everyone [2].

4 Data and Information Flow

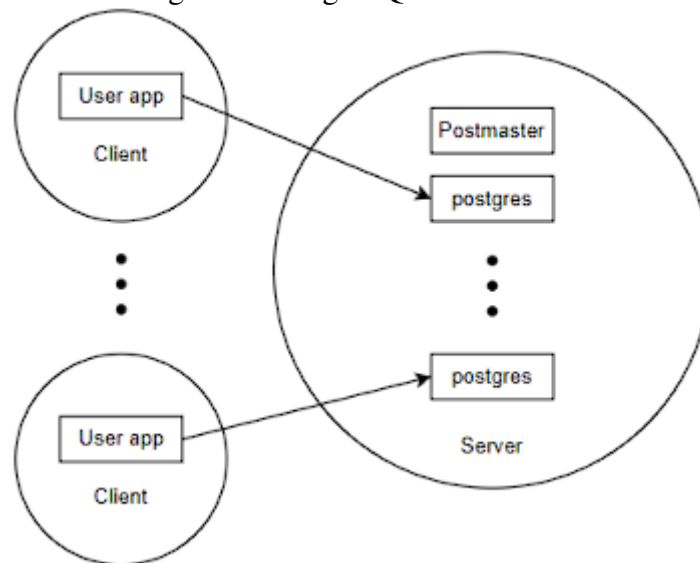
4.1 Establishing a Connection

Any data transfer mechanism related to query evaluation begins with a client process establishing a connection to a PostgreSQL server process. PostgreSQL employs a client-server model where each client has a dedicated server, and communication between the

two processes is facilitated through a message-based protocol. The client, which could be a web application, mobile app, or any system interacting with the database, is the origin or the requester of the data [3].

The connection system is demonstrated in Figure 1. A master process, known as the postmaster, listens for incoming connections at a specific TCP/IP port. Each time a connection is requested, the postmaster creates a new server process called 'postgres'. Each client process is connected to exactly one server process [3].

Figure 1: PostgreSQL connections



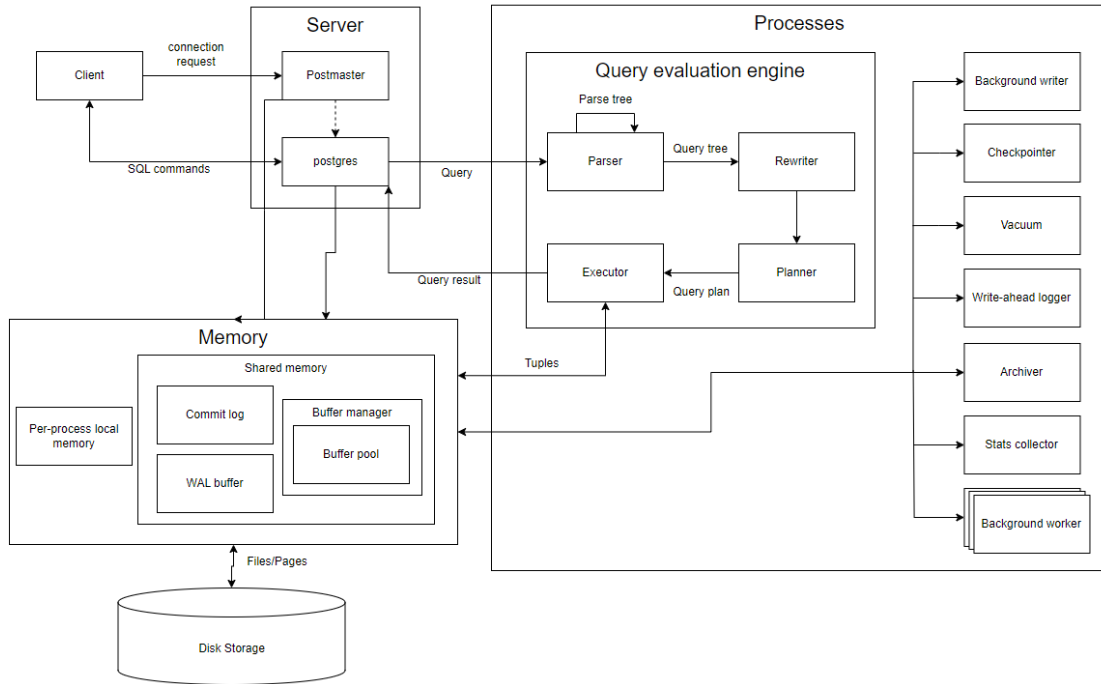
To ensure data integrity during concurrent data access, the postgres instances communicate with each other using semaphores and shared memory. This robust communication and process management system underpins the reliable operation of PostgreSQL databases [3].

After a connection is established, the client can send a query to the server it is connected to. The server parses the query to find any syntactical errors. The operations of the server involve handling all queries and statements issued by a connected client, such as requesting the query evaluation engine to retrieve the desired tuples. Finally, it returns the retrieved tuples to the client by transmitting them over the established connection [4].

4.2 Architecture Overview

Along with the server, the process and memory architecture make up the PostgreSQL system. A simplified general overview of the architecture can be seen in Figure 2. The process architecture consists of the query evaluation engine integrated as part of the postgres instances and various background processes which perform tasks for database management unrelated to query handling such as logging. There are additional background worker processes that are implemented by users which can also be used for parallel operations [5].

Figure 2: PostgreSQL architecture



Memory is categorized into local or shared memory. Local memory is allocated by each postgres instances for query processing like storing temporary tables. Meanwhile, shared memory is used by all remaining background processes such as transaction control [5]. The flow of information and data between the components and their tasks will be discussed in the following sections.

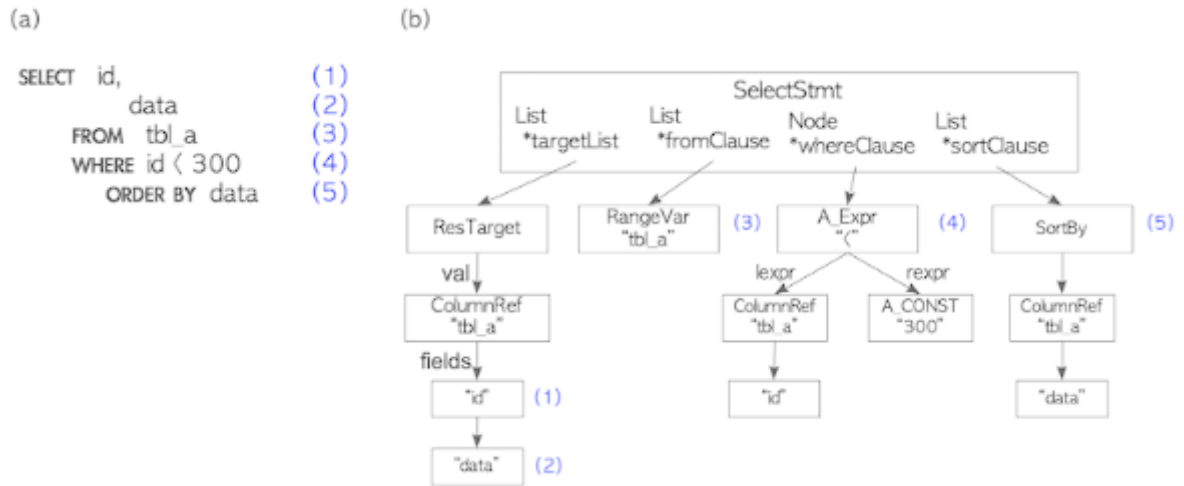
4.3 Query Evaluation Engine

4.3.1 Parsing a Query

All queries require parse analysis conducted by the parser. It checks the query string from the client as plain text for syntactic validity. If the syntax is correct then a parse tree is generated; otherwise an error is returned. Notably, the parser does not check the semantics of the query string. The sole function of the parser is to verify the syntax of the query in order to generate a parse tree. The reason being that the parse tree is created based on fixed rules about the syntactic structure of SQL without any reference to the system catalogs. Consequently, the detailed semantics of a requested query is unknown. Therefore, the parser will only return an error in the event of a syntax error [6]. The root node of the parse tree is the `SelectStmt` structure, with the query's relations and restrictions forming the child nodes [5]. Figure 3 shows an example of a parse tree generated as output by the parser given a selection query.

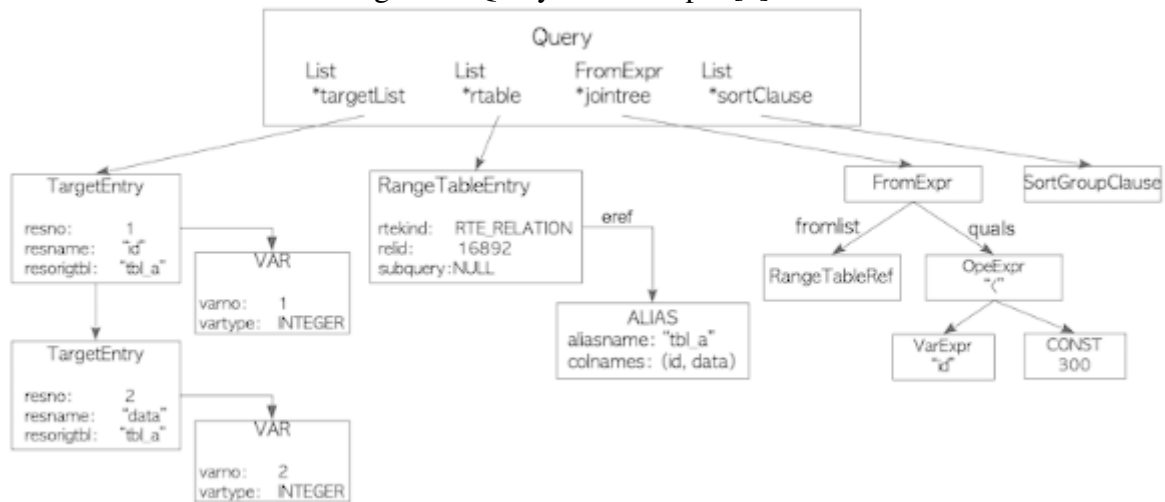
Upon completion of the parsing stage, the transformation function receives the tree returned by the parser as input. It performs the required semantic interpretation needed to identify which tables, functions, and operators are referenced by the query. The final data structure that is built to encapsulate the semantic information is called the query tree [6]. An example of this structure can be seen in Figure 4. The structure is annotated

Figure 3: Parse tree example [5]



with metadata pertaining to the query and its specific clauses [5].

Figure 4: Query tree example [5]



4.3.2 Rewriting

PostgreSQL includes an intricate and robust rule system for the specification of views and ambiguous view updates. The rule system is implemented through a technique called query rewriting. The rewrite system is a module that exists between the parser stage and the planner. With respect to data flow, both the input and the output of the re-writer are query trees. The representation and level of semantic detail in these trees remain unchanged during the rewriting process, thus it can be conceptualized as a form of macro expansion [7].

4.3.3 Creating an Execution Plan

The planner's role is to construct an optimal query execution plan, or simply a query plan, from the query tree. This involves generating all alternative plans that are equiv-

alent. The query plan is represented as a tree structure with each node being a specific relational algebra operation or sub-plan. Each node is additionally annotated with access methods for tables and the operators' implementations. Although it is usually infeasible, the query planner will ideally consider every possible query plan and choose the one with the least estimated cost [8].

PostgreSQL's planner utilizes data structures called "paths" as part of its decision-making. Paths are abstracted representations of complete plans. Paths contain only the necessary information for the planner to make optimization decisions. The planner's ability to adapt to changing data distributions and statistics is crucial for maintaining performance as the database grows [8].

The planner starts by generating sub-plans from scanning each single-relation used in the query. The potential plans are dictated by the available indexes on each relation. However, a sequential scan plan is always generated as any relation could potentially require a sequential scan. An example of a sequential scan plan is shown in Figure 5. Additional plans are created if there are indexes that match the relations in the query that need to be scanned. Index scan plans are also generated for indexes that have a sort ordering that can align with the query's ORDER BY clause, or a sort ordering that might be useful for merge joining [8].

If the query necessitates joining two or more relations, join plans are considered after all feasible single-relation scan plans have been identified. The three available join strategies are nested loop join, merge join, and hash join. In nested loop join, the inner relation is scanned once for every row found in the outer relation. For merge join, each relation is sorted on the join attributes before the join starts. The two relations are then scanned in parallel and matching rows are joined. Lastly, for hash join, the inner relation is first scanned and loaded into a hash table using its join attributes as hash keys. The outer relation is scanned afterwards and the correct values of every row found are used as hash keys to locate the matching rows in the hash table [8].

When the query involves more than two relations, the final result must be constructed by a tree of join steps, each with two inputs. The planner evaluates different possible join sequences between a pair of relations to find the least cost sequence. If the query is simple (i.e., there are fewer relations than required for a genetic query optimization), a near-exhaustive search is conducted to find the best join sequence. All possible plans are generated for each pair considered by the planner, and the one that is likely the lowest cost is chosen. When the `geqo_threshold` parameter set by the user is exceeded, the join sequences considered are determined by the genetic query optimizer [8].

The completed plan tree comprises sequential or index scans of the base relations, as well as any necessary nested-loop, merge, or hash join nodes. Any additional steps required are also included in the tree as nodes, such as sort nodes or aggregate-function calculation nodes [8].

Figure 5: A simple query plan from a sequential file scan

```

QUERY PLAN
-----
[
  {
    "Plan": {
      "Node Type": "Seq Scan",
      "Parallel Aware": false,
      "Async Capable": false,
      "Relation Name": "counties",
      "Alias": "counties",
      "Startup Cost": 0.00,
      "Total Cost": 16013.60,
      "Plan Rows": 196203,
      "Plan Width": 92,
      "Actual Startup Time": 0.778,
      "Actual Total Time": 156.024,
      "Actual Rows": 239233,
      "Actual Loops": 1,
      "Filter": "(cases > 1000)",
      "Rows Removed by Filter": 674794
    },
    "Planning Time": 0.063,
    "Triggers": [
    ],
    "Execution Time": 161.005
  }
]
1 row)

```

4.3.4 Executing a Query

The executor in PostgreSQL operates on the query plan generated by the planner. It processes the plan recursively, extracting the necessary rows in a demand-pull pipeline fashion. This means that rows are only pushed forward when the executor requests them. Each time a plan node is invoked, it delivers one more tuples or returns NULL if it has finished delivering rows [9].

For example, given a MergeJoin node at the root of the query plan then two rows must be fetched with one from each sub-plan. Therefore, the executor recursively calls itself to process the sub-plans starting with the left sub-plan as the new root. Assume that the new root is a SeqScan node, representing retrieval of a table from storage. Execution of this node causes the executor to fetch a row from the table and return it up to the calling node. When the SeqScan node returns a row, the MergeJoin node continues in the same fashion with the right tree and compares the rows to see if they can be joined. If so, it returns a join row to its caller. On the next call, or immediately if it cannot join the current pair of inputs, it advances to the next row of one table or the other, and checks for a match again. Eventually, one sub-plan or the other returns NULL, as does the MergeJoin node, indicating that no more join rows can be formed.

Complex queries may involve multiple levels of sub-plans. Regardless of the complexity, the data flow remains consistent. Each node computes and returns its next output row each time it is called. Each node is also responsible for applying any selection or projection expressions assigned to it by the planner [9]. The executor is the first and only stage that makes use of the storage system and it will issue requests for disk access. Once the query is completed, the results are sent back to the client application that initiated the query [9].

4.4 Parallel Querying

PostgreSQL is capable of using parallel query evaluation that leverages multiple CPU's to expedite query response times. The performance improvement can be substantial for queries that are suitable for parallel execution. Many queries can be evaluated more than twice as fast, and some queries can even run four times faster or more. Complex queries that scan a large amount of data but return only a few rows to the user will typically benefit most [10]. For example, it would improve equality selections on increasingly large data sets.

Parallel execution of operations in an execution plan involves the evaluation of different operations in parallel. PostgreSQL incorporates data-partitioned parallel evaluation: it partitions the retrieved rows, works on each partition in parallel, and combines the operation result. Every parallel query plan in PostgreSQL includes either a Gather or Gather Merge node. These nodes have exactly one child plan, which represents the portion of the plan that will be executed in parallel. The execution of a parallel query is initiated by the “leader” process (the process that originally requests the query). Other processes executing the same query are referred to as “worker” processes [10].

When the Gather node is expanded during query execution, the planner requests a certain number of background worker processes. The optimal plan may depend on the number of available workers, which can be set with user configurations. The user can also set the number of workers to zero or run PostgreSQL in single-user mode so that parallel querying is never used. Every background worker process that is successfully requested for a given parallel query will execute the parallel portion of the plan. The leader will also execute a portion of the plan, but it has an additional responsibility that takes priority: it must read all of the tuples generated by the workers [10].

Since each worker executes the parallel portion of the plan to completion, it is not possible to take a query plan and run it using multiple workers. Each worker would produce a full copy of the output result set. Instead, a partial plan is created so that each process that executes the partial plan will generate only a subset of the output rows in such a way that each required output row is guaranteed to be generated by exactly one of the cooperating processes [10]. Execution of parallel plans involve four operations named parallel-aware scans which satisfy these conditions [10]:

1. Parallel Scans: divides table blocks among processes to ensure each process generates a subset of the output rows.
2. Parallel Joins: the look ups of the outer relations are divided among workers in nested loop, merge, and hash joins, while the inner relation remains non-parallel except for certain join types.
3. Parallel Aggregation: each process first produces partial results, followed by a final aggregation by the leader.
4. Parallel Append: distributes processes across child plans to execute multiple plans simultaneously, avoiding contention and startup costs.

Even when it is possible to create parallel query plans, the planner will not generate them for a given query if any of the following are true: the query writes any data or

locks any database rows, the query might be suspended during execution, the query uses any function marked `PARALLEL UNSAFE`, or if the query is running inside of another query that is already parallel [10].

4.5 Storage Access

4.5.1 Reading Records

One way PostgreSQL retrieves records is through a sequential scan, where it navigates blocks on the disk by reading pages in an order computed by PostgreSQL. Each page contains a pointer to the next and the previous logical page, forming a linked list. This structure allows for both forward and backward query plan execution. A line table on each page directs to the start of each record. Additionally, on each page there is a line table containing pointers to the starting byte of each anchor point record on that page [5].

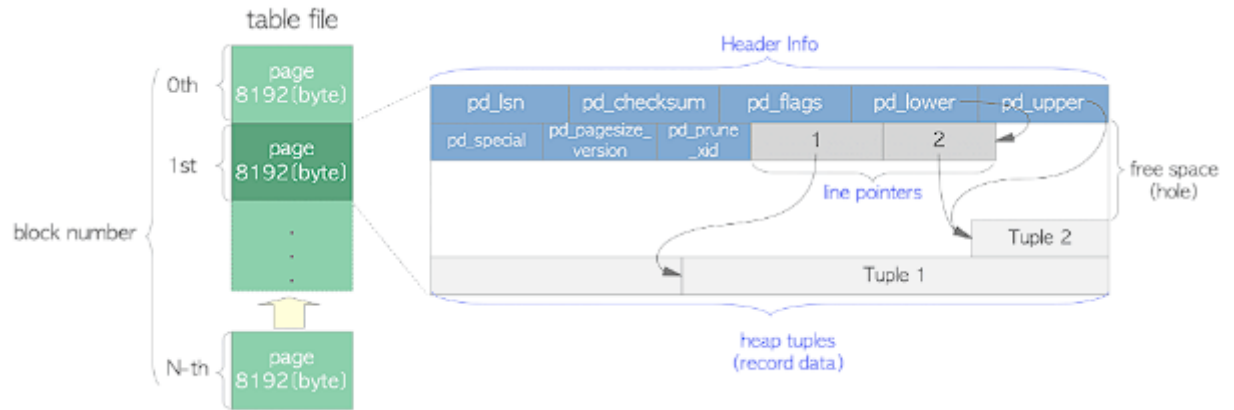
The second access method involves a B-tree index scan by default. Additional index access methods will be discussed. PostgreSQL consults an index file that contains index tuples, each of which is composed of an index key and a TID (composed of a block/page number and record offset within the page) that points to the target heap tuple. If the index tuple matches a request, PostgreSQL reads the desired heap tuple using the obtained TID value [5].

4.5.2 Writing Records

Inside a data file, whether it be a heap table, index, free space map, or visibility map, the file is divided into pages or blocks of fixed length, which is 8192 bytes (8 KB) by default. The pages within each file are numbered sequentially and are monotonically increasing. If the file is full, PostgreSQL adds a new empty page to the end of the file to increase the file size [5].

The page layout in PostgreSQL uses variable length records. Figure 6 shows the structure of a heap table file. The variable-length record structure includes `pd_lower` and `pd_upper` pointers, marking the free space boundaries on a page. When a new record is inserted, it is placed after the last one on the page. The pointer to the new record is appended to the last one. The `pd_lower` changes to point to the end of the new record pointer, and the `pd_upper` points to the new record. Other header data within this page (e.g., `pd_lsn`, `pg_checksum`, `pg_flag`) are also updated to appropriate values [5].

Figure 6: Page layout of a heap table file [5]



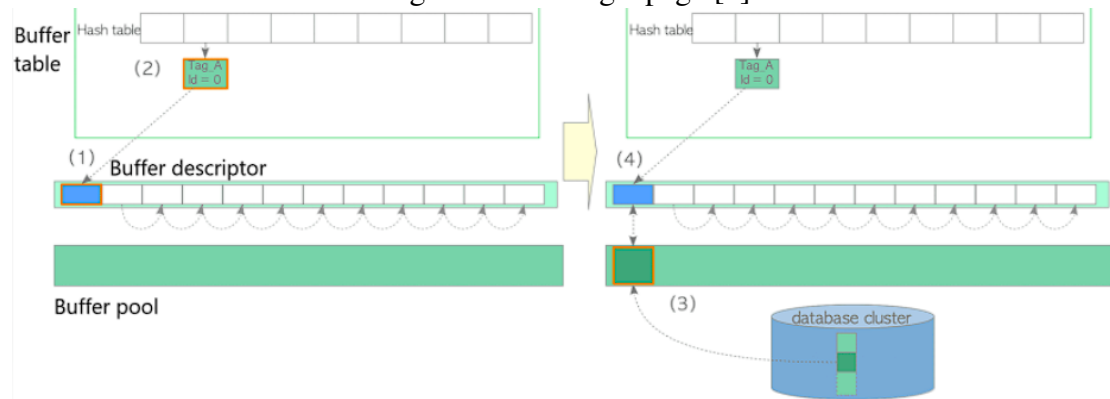
4.5.3 Buffer Manager

The buffer manager manages data transfers between shared memory and persistent storage. The buffer manager loads pages within tables and indexes from a persistent storage to the shared buffer pool, and PostgreSQL operates on them directly. When a query requires data from a table or index, the buffer manager checks if the required data can be served from memory before accessing the disk. The PostgreSQL buffer manager comprises a buffer table, buffer descriptors, and buffer pool. The buffer pool layer stores data file pages, such as tables and indexes, as well as free-space maps and visibility maps. The buffer pool is an array, where each slot stores one page of a data file. The indices of a buffer pool array are referred to as `buffer_IDs` [5].

The second component is the buffer descriptor. It is an array of descriptors which have a one-to-one correspondence to a buffer pool slot and holds the metadata of the stored page in the corresponding slot. Conceptually, the metadata can be defined as the pin count of the page. For PostgreSQL, the list of all empty or unpinned pages comprise the free list. The third component is the buffer table which is a hash table that stores the relations between the `buffer_tags` of stored pages and the `buffer_ids` of the descriptors that hold the stored pages' respective metadata. The buffer manager also uses a semaphore called the `BufMappingLock` to protect the data integrity of the entire buffer table [5]. Figure 7 demonstrates how the buffer manager operates:

1. Locate an empty descriptor from the free-list, and pin it.
2. Insert a new entry into the buffer table that maps the tag of the page to the `buffer_id` of the descriptor.
3. Load the new page from storage into the buffer pool slot.
4. Save the metadata of the new page to the retrieved descriptor.

Figure 7: Loading a page [5]



When reading a table or index page, a back-end process sends a request that includes the page's `buffer_tag` (unique ID of a data page) to the buffer manager. The buffer manager creates the `buffer_tag` of the page. It then hashes it to acquire the `buffer_id` and pins the buffer descriptor for this `buffer_id`. The appropriate slot is then accessed. If the requested page is not stored in the buffer pool, the buffer manager loads the page from persistent storage to one of the buffer pool slots and then returns the `buffer_ID` of the slot. The back-end process accesses the `buffer_ID`'s slot afterwards [5].

When the buffer is full and a new request is not found, the buffer manager must select one page in the buffer pool to be replaced by the requested page. PostgreSQL uses the clock sweep algorithm to perform this selection. After the buffer manager selects a "victim" buffer pool slot to replace, the slot is written to storage if it is dirty and it is deleted so that the new page can be inserted in its slot [5]. As previously mentioned, dirty pages are written to storage. However, this task is distributed to the check-pointer and background writer [5].

Thus, the buffer manager allows multiple clients to access frequently used data without repeatedly reading from disk. The buffer management of PostgreSQL is very efficient, ensuring improved performance and scalability by reducing disk I/O.

4.5.4 Clock Sweep

This algorithm is a variant of NFU (Not Frequently Used) with low overhead; it selects less frequently used pages efficiently. The buffer descriptors are implemented as a circular list. The `nextVictimBuffer` variable is always pointing to one of the buffer descriptors and rotates clockwise [5]. The algorithm is as follows [5]:

1. Obtain the potential victim buffer descriptor pointed to by `nextVictimBuffer`.
2. If the candidate buffer descriptor is unpinned, proceed to step (3). Otherwise, proceed to step (4).
3. If the `usage_count` of the candidate descriptor is 0, select the corresponding slot of this descriptor as a victim and proceed to step (5). Otherwise, decrease this descriptor's `usage_count` by 1 and proceed to step (4).

4. Advance the nextVictimBuffer to the next descriptor (if at the end, wrap around) and return to step (1). Repeat until a victim is found.
5. Return the buffer_id of the victim.

4.5.5 Ring Buffer

When reading or writing a huge table, PostgreSQL uses a ring buffer instead of the buffer pool. The ring buffer is a small and temporary buffer area used solely for a large table. The motivation is to prevent a back-end process from evicting all stored pages in the buffer pool when reading a large table, which decreases the cache hit ratio. The allocated ring buffer is released immediately after use [5]. When any of the following conditions is met, a ring buffer is allocated to shared memory [5]:

1. Bulk-reading: when scanning a relation whose size exceeds one-quarter of the buffer pool size, the ring buffer size is 256 KB.
2. Bulk-writing: when the SQL commands listed below are executed, the ring buffer size is 16 MB.
 - (a) COPY FROM command.
 - (b) CREATE TABLE AS command.
 - (c) CREATE MATERIALIZED VIEW or REFRESH MATERIALIZED VIEW command.
 - (d) ALTER TABLE command.
3. Vacuum-processing: when auto-vacuum performs vacuum processing the ring buffer size is 256 KB.

4.6 Data Partitioning and Foreign Data

PostgreSQL provides foreign data wrappers and declarative partitioning as options to mimic sharding. Database sharding is a type of horizontal fragmentation. It is the process of storing a large database across multiple machines by splitting data into smaller chunks, called shards, and storing them across several database servers. This improves query response time and allows organizations to support database scaling by allowing parallel processing of smaller shards and adding new shards without shutting down the client application [11].

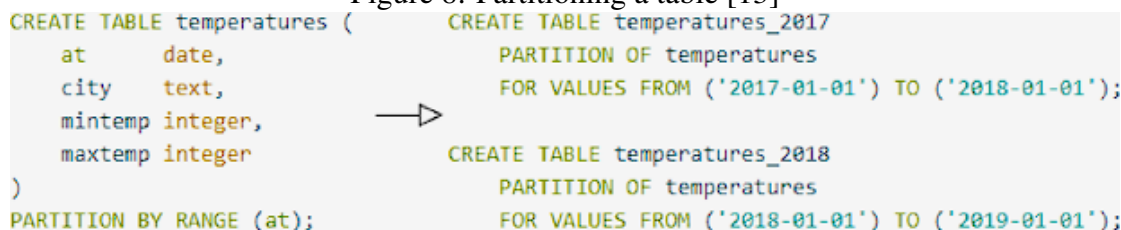
With regards to database performance, query response times can be improved dramatically when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning effectively substitutes for the upper tree index levels, making it more likely that the heavily-used parts of the indexes fit in memory. Moreover, when queries or updates access a large percentage of a single partition, performance can be improved by using a sequential scan of that partition instead of using an index, which would require random-access reads scattered across the whole table. Additionally, bulk loads and deletes can be accomplished by adding or removing

partitions, if the usage pattern is accounted for in the partitioning design. Moreover, infrequently accessed data can be migrated to cheaper and slower storage [12].

However, maintaining multiple, distinct tables implies that when an application has to access older data, it has to find out what tables are present in the schema, query each of them and combine the results from each table. PostgreSQL makes this possible without changing application code by partitioning tables. The partitioned table itself is a “virtual” table having no storage of its own. Instead, the storage belongs to partitions, which are otherwise-ordinary tables associated with the partitioned table. Each partition stores a subset of the data as defined by its partition bounds. All rows inserted into a partitioned table will be routed to the appropriate one of the partitions based on the values of the partition key columns. Updating the partition key of a row will cause it to be moved into a different partition if it no longer satisfies the partition bounds of its original partition [12]. In short, applications can interface with a top-level or master table as if querying from one large table, though PostgreSQL creates multiple partitioned tables that store non-overlapping data in the back-end.

In order to implement partitioning, a partitioning key and a partitioning method (hash, list, or range based) is chosen [13]. Figure 8 demonstrates how partitioning (using the range based method) is akin to creating separate tables yet the data distribution is transparent to the user.

Figure 8: Partitioning a table [13]



The diagram illustrates the process of partitioning a table. On the left, a single table definition is shown: `CREATE TABLE temperatures (at date, city text, mintemp integer, maxtemp integer) PARTITION BY RANGE (at);`. An arrow points to the right, where two partitioned table definitions are shown: `CREATE TABLE temperatures_2017 PARTITION OF temperatures FOR VALUES FROM ('2017-01-01') TO ('2018-01-01');` and `CREATE TABLE temperatures_2018 PARTITION OF temperatures FOR VALUES FROM ('2018-01-01') TO ('2019-01-01');`.

Standard PostgreSQL partitioning creates all partitions equal and on the same physical cluster. Therefore, partitioning is not a built-in way to distribute data across multiple physical clusters. Instead, it implements logical data organization and splitting but does not natively support distributing data [14]. PostgreSQL foreign data wrappers provide a way to access a foreign tables like regular tables in the local database (physical sharding). Users can create “foreign servers” to host separate databases. Afterwards, users can create a “foreign table” on a local server that serves as a proxy for accessing the table on the “foreign server”. Creating the foreign table as a partition on the local server will allow users to modify and query tables from their own server, effectively implementing the sharding design [13].

4.7 Multi-Version Concurrency Control

PostgreSQL maintains data consistency by using a multi-version model. While querying a database, each transaction accesses different versions table rows, regardless of the current state of the underlying data. Modified rows are marked with the transaction ID of the transaction that changed them. This protects the transaction from viewing inconsistent data that could be caused by concurrent transaction updates on the same

data rows, providing transaction isolation for each database session. This also allows several transactions to run simultaneously, and eliminates the need for locks or blocking operations to enhance the database's performance and responsiveness [15].

The main difference between multiversion and lock models is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data. Thus, reading never blocks writing and writing never blocks reading [12]. A snapshot in MVCC is a virtual read-only, point-in-time copy of the data recorded [16]. MVCC provides a consistent view of the data to each transaction even during other transactions which are making modifications. Each transaction has its own snapshot, that has data consistent with the beginning of its transaction. The snapshot comprises all information about all the tuples in the database and controls the tuples that are visible to the transaction at the time the snapshot is taken [16].

PostgreSQL will recognize when a transaction is trying to update a modified record by a different transaction based on transaction IDs. Modified rows also store the transaction ID of the transaction that changed them. The transaction ID of a snapshot is compared between requests to see which rows can be read without errors. Inconsistent transactions will create serialization errors from the former's snapshot being outdated. Transactions will have to refresh their snapshots to obtain consistent data and PostgreSQL will create new versions of tables after transactions have been committed [16].

MVCC can result in high concurrency and scalability and hence improve the performance of database applications in a multi-user environment. MVCC is often faster than other concurrency control techniques, especially in circumstances with high read-to-write ratios. However, when there are numerous transactions under processing, MVCC will require more memory as compared to other concurrency control techniques [16]. For example, MVCC generates outdated and unused row versions over time which eventually have to be reclaimed. This reclamation of old rows often hinders memory performance [17].

4.8 Background Server Processes

4.8.1 Vacuuming

PostgreSQL is unlike ARIES systems since it does not have a rollback or undo log. Instead it has a vacuum system and a commit log.

PostgreSQL databases require regular maintenance known as vacuuming on each table for several reasons. Those being to recover or reuse disk space occupied by updated or deleted rows, to update data statistics used by the PostgreSQL query planner, to update the visibility map which speeds up index-only scans, and to protect against loss of very old data due to transaction IDs reaching numerical limits (wraparound) [17]. The vacuum in PostgreSQL is synonymous with garbage collection. Therefore, this process optimizes performance by managing storage and to help accurately update the number of rows returned by a query so that the most efficient execution plan is chosen. Both contribute to its scalability by maintaining efficiency.

4.8.2 Reclaiming Space

The standard `VACUUM` command removes dead row versions in tables and indexes and marks the space available for future reuse, while `VACUUM FULL` actively compacts tables by writing a complete new version of the table file with no dead space. This minimizes the size of the table, but can take a long time and requires extra temporary disk space for the new copy of the table [17].

In PostgreSQL, an update or deletion of a row does not immediately remove the old version of the row. This is required for multiversion concurrency control. The row version must remain while it is still potentially visible to other transactions. Although, the space it occupies must be reclaimed for reuse by new rows when no transaction is interested to avoid unbounded growth of disk space requirements. However, the space is not returned to the operating system by the standard `VACUUM` (compared to `VACUUM FULL`) unless one or more pages at the end of a table become entirely free and an exclusive table lock can be easily obtained. To undo a transaction, the transaction is marked as aborted which then makes the old row version to automatically become the current row version again without a need for undoing any action [17].

The auto-vacuum daemon of PostgreSQL does standard `VACUUM` often enough to avoid needing `VACUUM FULL`. `VACUUM FULL` is generally avoided since tables will often grow from their minimum size. This approach maintains steady-state usage of disk space as each table occupies space equivalent to its minimum size as well as the additional space used for intermediate vacuum runs. The daemon also schedules vacuuming dynamically in response to update activity. In that, it will perform vacuuming when load is low [17].

4.8.3 Updating Statistics

The query planner relies on statistical information about table contents in order to generate optimal plans. The statistics are gathered by the `ANALYZE` command which is issued as an optional step in `VACUUM`. It is important to have reasonably accurate statistics to maintain database performance which implies frequent updates of statistics [17].

The auto-vacuum daemon, if enabled, will automatically issue `ANALYZE` commands whenever the content of a table has changed sufficiently. The daemon schedules `ANALYZE` strictly as a function of the number of rows inserted or updated. Though, it has no knowledge of whether that will lead to meaningful statistical changes [17].

4.8.4 Updating the Visibility Map

Vacuum maintains a visibility map for each table to identify which pages contain only tuples that are known to be visible to all active transactions and all future transactions, until the page is modified. This allows a vacuum to skip the active pages on the next run since there is no row to reclaim. More importantly, it allows PostgreSQL to perform index-only evaluation to confirm if the index entry should be seen by the current transaction. An index-only scan checks on the visibility map first. If it's known that all tuples on the page are visible, the heap fetch can be skipped to prevent disk accesses [17].

4.8.5 Transaction ID Wraparound

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction IDs. However, transaction IDs are limited to 32 bits and only increase in value, which makes IDs prone to wraparound when a database runs for enough time. An ID counter eventually wraps around to zero, making all transactions that were in the past appear to be in the future which makes their output become invisible. This causes catastrophic data loss, thus it is necessary to vacuum every table in every database at least once every two billion transactions. Periodic vacuuming will mark rows as frozen, indicating that they were inserted by a transaction that committed sufficiently far in the past that the effects of the inserting transaction are certain to be visible to all current and future transactions [17].

4.8.6 Background Writer

The purpose of the background writer is to issue writes of “dirty” (new or modified) shared buffers. When there is no longer a sufficient amount of clean shared buffers, the background writer writes dirty data to the file system and marks them as clean. Flushing dirty data reduces queries from being blocked while executing. Writing out dirty buffers in advance lowers the chance that server processes handling user queries will have to write dirty buffers themselves during a checkpoint (a database recovery technique) which improves overhead performance [18]. However, the background writer does cause a net overall increase in I/O load. It operates asynchronously to checkpoints which flushes dirty data. While a repeatedly-dirtied page might be written only once per checkpoint interval, the background writer might write it several times as it is dirtied [18]. The aim is to distribute the I/O load more evenly over time by independently writing dirty pages to disk, which can prevent sudden spikes in I/O during checkpoint operations.

4.8.7 Write-ahead logging

Write-Ahead Logging (WAL) is a no-force, steal mechanism used in PostgreSQL to ensure data integrity. The core principle of WAL is that changes to data files must be logged before they are written to the data files. This approach reduces the number of disk writes, as only the WAL file needs to be flushed to disk to commit a transaction, not every data file changed by the transaction. Moreover, the WAL file is written sequentially so the cost of flushing the WAL is much less than the cost of flushing the data pages [19].

WAL records are appended to the WAL files as each new record is written, with the insert position described by a Log Sequence Number (LSN) which are unique and increasing monotonically. WAL files are stored in the `pg_wal` directory under the data directory, as a set of segment files. Each segment is divided into pages, and the WAL record headers are described in `accessxlogrecord.h`. It's beneficial to store the WAL on a different disk from the main database files to improve performance and reliability [19].

Checkpoints are specific points in transaction sequences that ensure all data before them have been updated in the disk files. At these checkpoints, all uncommitted data pages

are saved to disk, and a unique checkpoint record is written to the WAL file. In case of a system crash, the recovery procedure uses the latest checkpoint record to identify the starting point (the redo record) for the REDO operation in the WAL. All changes to data files before this point are guaranteed to be saved on disk. Therefore, after a checkpoint, WAL segments before the redo record are unnecessary and can be recycled or deleted. However, if WAL archiving is in progress, these segments must be archived before they can be recycled or removed [19].

5 Index

Indexing in PostgreSQL involves the creation of optimized data structures to efficiently search and retrieve data from tables [20]. An index is essentially a subset of a table organized in a way that allows PostgreSQL to rapidly locate and retrieve rows matching specific query conditions. When executing a query, PostgreSQL examines available indexes to determine if any can be used to fulfill the query condition. If a relevant index is found, PostgreSQL utilizes it to swiftly identify corresponding rows in the table. This results in significantly faster queries, especially when dealing with large tables or complex conditions. PostgreSQL offers support for various index types, including B-tree, hash, GiST, SP-GiST, and BRIN. Each index type is tailored for different query types and data access patterns. In addition to standard index types, PostgreSQL allows users to define custom indexes using user-defined functions. It's important to note that creating an index requires additional disk space and may impact the performance of write operations, such as INSERT, UPDATE, and DELETE. Therefore, careful consideration is essential when choosing which columns to index based on frequently executed queries and data access patterns.

5.1 B-tree Index

The B-tree index stands out as the most frequently utilized index type in PostgreSQL, offering efficient storage and retrieval of data. It serves as the default index type, meaning that when the CREATE INDEX command is employed without specifying a particular index type [20], PostgreSQL automatically generates a B-tree index for the specified table or column. The structure of a B-tree index resembles that of a tree. The index commences with a root node, featuring pointers to child nodes. Each node within the tree typically accommodates multiple key-value pairs, where the keys are utilized for indexing, and the values indicate the corresponding data location in the table.

5.2 Hash Indexes

Hash indexes are crafted for swift key-value lookups. They excel in scenarios where query conditions demand equality checks on indexed columns, offering exceptionally rapid retrieval. This is achieved by the hash function directly determining the location of the desired data. Hash indexes prove most effective for equality comparisons, such as = or IN operations. Similar to other index types, hash indexes require maintenance during data modifications, encompassing inserts, updates, and deletes, to guar-

antee data consistency. However, it's worth noting that hash index maintenance can be more resource-intensive compared to B-tree indexes. This is primarily attributed to the necessity of resolving collisions and rehashing data [21].

5.3 GiST and SP-GiST Indexes

GiST (Generalized Search Tree) and SP-GiST (Space-Partitioned Generalized Search Tree) indexes represent sophisticated index types in PostgreSQL, offering versatile support for a diverse array of data types and search operations. These indexes are particularly valuable when dealing with intricate data structures and spatial data. GiST indexes, in particular, are instrumental for optimizing full-text searches. They provide an effective means to enhance the speed and efficiency of searches involving complex data types. Both GiST and SP-GiST are optimized for nearest-neighbor searches such as selecting the closest 5 locations from point (113,621) [20].

5.4 BRIN Indexes

BRIN, short for Block Range Index, represents a specific index type in PostgreSQL designed to deliver efficient indexing for large tables with sorted data. The BRIN index encompasses the minimum and maximum values within a group of database pages [20].

Key features of BRIN indexes include:

1. **Optimization for Speed:** BRIN indexes provide a straightforward method for optimizing speed, making them well-suited for large datasets with sorted characteristics.
2. **Suitability for Sequential Data:** BRIN indexes are particularly valuable for data displaying sequential or sorted attributes, such as time series data or data with a natural ordering.
3. **Logical Block Division:** BRIN indexes divide the table into logical blocks and store summary information about each block.
4. **Range of Values:** Each block contains a range of values, and the index stores the minimum and maximum values within each block.
5. **Space Efficiency:** Instead of storing individual index entries for each row, BRIN indexes store block-level summaries, resulting in smaller sizes compared to other index types.
6. **Optimal Performance for Sequential Scans:** BRIN indexes excel when data is sorted or when sequential scans prove more efficient than index scans.

6 Security

For a PostgreSQL database, multiple layers of security are enforced. Any client-server connections can only be made by using a Unix socket by default, so TCP sockets are

not allowed unless manually enforced. In addition, PostgreSQL allows the option to limit allowed IP addresses and force the authentication of connections. All Postgres users will have their own username and have the option to set up a password, thus this security is optional to them. However, depending on the authentication method used for the Unix or TCP/IP connections, it may be required to access the server. In regards to accessing the databases, users will not initially have the ability to write to databases created by someone else, this access must be given to them. These are basically the main ways database security is handled, but there are different categories of security that are in play to ensure these methods are handled properly. There is user authentication, usernames and groups, and access control. There are also very special considerations with functions and other object write methods which are also mentioned [22].

6.1 User Authentication

Authentication is used to ensure that the one accessing the database is the person who is supposed to be accessing it. There are a variety of different ways of accessing the databases, but one common way is using the shell. The shell, also known as `psql`, is a terminal that allows users to log in and connect to a database of their choosing. They are free to use queries, write scripts, and output results as they would like once they have been verified [23]. The network is another method to access the databases, and this is only applicable with a distributed Postgres system. Our case study will not focus on this version and will focus primarily on the shell but it is still something to be mentioned. In both cases, a certain authentication protocol will be used based on the host and the database that is attempting to be reached to allow or reject access. Unfortunately, this form of authentication has its downfalls as attackers can mask as the host and intrude depending on the strength of the authentication method. It is known as Host-Based Access Control [22].

Host-Based Access Control is the main way that PostgreSQL determines who can access the databases and how these individuals must validate who they are. All clients will be authenticated or not based on the `pg_hba.conf` file stored in the data directory. This file is essentially a list of authentication records where each record spans one line and is made up of a number of fields separated by spaces. All records have the type of connection, a list of allowed IP addresses, a database and user name, and the specified authentication method used for connections. Of course the authentication method is only for the specific records with the same other parameters. Once a record is found in the file that matches the parameters given (IP address, database, user name) by the client attempting to access the server, that record is used to authenticate the client. One downfall is that if a record is selected and the authentication is not successful, authentication is finished and the user is denied access. It does not matter if there is another record that would allow authentication, as it is a one-and-done approach. In the event zero matches are found, the client is also denied access [24].

Connections can be made via two types of sockets: Unix and TCP/IP. The `pg_hba` file allows connections of both types but it depends on different fields in the records. An example record format that would be used for Unix connections is as follows: “local database user auth-method.” It is the local keyword that tells the server that it is a Unix based connection that is attempting to be made. Database and user are pretty self-

explanatory keywords, while auth-method is one of many authentication methods to be used to verify the client. An example record that would satisfy TCP/IP connections is as follows: “host database user address auth-method.” It is the host keyword that signifies that a TCP/IP connection is attempting to be made, and the address keyword which was absent in Unix connections signifies the IP address of the client attempting to make the connection. For the TCP/IP connections, it can be specified if SSL is used to make the connections (hostssl) or not (hostnossl) [24].

As of PostgreSQL 16, there are eleven different authentication methods available [25]. Trust authentication is the first, which is simply saying that PostgreSQL believes that anyone who accesses the server and attempts to connect to databases has the authorization to, thus it trusts the clients. This is ideally not good for machines that span multiple users and is preferred for single users only [26]. Next there is password authentication, which comes with three separate methods. There is scram-sha-256 and md5, which both attempt to encrypt the passwords stored on the server, and password, which has the password sent in bare and is therefore vulnerable to certain attacks. This form of authentication is important as it will be used later on in the case study analysis of this section [27]. There is also Ident authentication and peer authentication. Ident authentication is only supported for TCP/IP connections while peer authentication is only supported on Unix connections. Ident authentication takes note of the operating system attempting to connect to the server and uses it as an allowed username. Peer authentication does the exact same thing but it is only for Unix connections. Other forms of authentication include GSSAPI which uses another server known as Kerberos for verification, SSPI which can only be used for Windows machines, and Certificate authentication which requires clients to send SSL certificates to verify their identity [25].

6.2 User Names and Groups

Users can be created by simply executing a program named “createuser.” This is identical to the “createrole” program except that when using “createuser” the login parameter is assumed to be true by default. The login parameter means that the user name is used to identify the user for that connection session and thus they can log in. Not being able to log in is still okay as permissions and access can still be managed by the said role. Only users that have the superuser privilege or the createrole privilege are allowed to create new users. The superuser privilege exceeds all other privileges, thus only a superuser can create another superuser, not merely another user with the create role privilege. It is important to note that superuser delegation must be thought of very carefully, as they are essentially given full access and could pretty much do anything they want. When users are created, they can be given privileges themselves such as the ability to create databases and roles themselves. Also, those creating a user can specify the length of time the user’s password may be active, forcing it to be changed later on which is great for security [28]. These created users can then be assigned to groups using the CREATE GROUP command (which is actually also another way of saying CREATE ROLE). In this case, if many users are to have the same roles (CREATEDB, LOGIN, SUPERUSER) or access privileges (in terms of reading and writing data from databases), they can all be given access at the same time because all roles in a group are given the same permissions the group has. It will be tested in the case study as to whether or not individual users can still be given privileges that the rest of the group

cannot have [29].

6.3 Access Control

As previously mentioned, Postgres allows users to choose the amount of access to their data other users will have. There are multiple methods of controlling this: being a database superuser, changing access privileges, and modifying or removing classes.

In the first case, if one is a database superuser, they basically have full access and almost all controlling methods used do not apply to them. The only thing that superusers are still required to do if applicable is login. They therefore can modify databases, create users, and hand down access privileges to any other user they would like. As previously mentioned, only database superusers can create other superusers, so this is one way of controlling it [30].

Access privileges are a bit more strict. When a user creates another user or role or any other kind of object, that object is assigned an owner, usually its creator. At creation time, only the owner or other superusers have the right to perform operations on the object. Privileges allow other users / roles to perform operations themselves. Multiple privileges based on SQL commands exist, such as SELECT, UPDATE, INSERT, DELETE, TRIGGER, and ALTER among others. The ALTER command is used on its own to change the object, such as by giving it a new owner or changing its name. For the other SQL commands, the GRANT or REVOKE statements need to be used. For example, if a user wanted to select values from a database table, the owner of the database could issue the following command: `GRANT SELECT ON database_name TO other_user`. This new user would then be able to SELECT values from this database. In the event the owner of the database decided that the user should no longer be able to select values, they could issue the following command: `REVOKE SELECT ON database_name FROM other_user`. Access privileges therefore allow owners of objects to share what they feel comfortable sharing with other users, and if they giveth, they may taketh away [31].

For class removal and modification, there are two main commands that can be used. There is the ALTER command which was already mentioned, and then there is the DROP command. The DROP command is used when something is no longer needed and is kind of used as a last resort. For instance, if there was a user who was no longer needed, once all necessary data had been saved, the user would be dropped using the DROP USER command. Note that if a role is referenced in any of the databases, the role cannot be dropped, rather the database must be modified or dropped first. Also, if the role had been given other privileges such as SELECT or UPDATE, these privileges must be removed as well. The DROP OWNED command may serve some use here [32].

6.4 Functions

Functions need to be mentioned because there is a huge security issue with them. Functions give users the ability to import some code into the PostgreSQL back-end server that other users will not know exists but can still accidentally run. It is basically allowing one to write a virus program and have others users get infected by it. This actually also

applied to triggers and RLS. Thus, choosing who to give the power of writing objects to needs to be very carefully considered. If the most careful control is not applicable, then another method of protection is to make sure that only owners that are fully trusted can have their objects written to. Any users that are not trusted should not be allowed to create objects [33].

It is the back-end server that houses where functions are executed. These functions have the capability to change data in the server given the programming language used, thus they can completely bypass any access control methods that are in place. Therefore, there are languages that are untrusted and these languages can only be used by superusers to create functions. Thus, superusers are assumed to be trusted [33].

6.5 Security Case Study

Now that all of the security has been laid out, some tests were to be conducted to see how PostgreSQL handles them. All tests would be conducted using the shell, and the tests to be run would include accessing the databases, creating users and groups, checking access control methods, and creating functions. The first thing that was chosen to be tested was accessing the database via Unix and TCP/IP sockets. To keep things simple, it was the superuser postgres that was logged into. The Unix login was simple enough to perform, as shown below in Figure 9.

Figure 9: Unix Socket Connection

```
C:\Program Files\PostgreSQL\16\bin>psql -U postgres -d postgres
Password for user postgres:
psql (16.0)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# |
```

```
postgres=# \conninfo
You are connected to database "postgres" as user "postgres" on host "localhost" (address "::1") at port "5432".
```

In this, the -U command indicates the username and the -d command indicates the database. The database flag is optional and if it is removed, PostgreSQL will attempt to connect to a database that has the same name as the username, thus it would try to connect to a database named postgres. For Unix sockets, the connection is made to the local-host and port 5432. A TCP/IP connection should allow us to change the IP address and port number used. To keep things simple, we will keep the port number the same because we know it will only be used for PostgreSQL, but we will attempt to change the host-name to UWO's IP address as demonstrated in Figure 10.

Figure 10: Failed TCP/IP Connection

```
C:\Program Files\PostgreSQL\16\bin>psql -U postgres -d postgres -h 172.30.104.164
psql: error: connection to server at "172.30.104.164", port 5432 failed: FATAL: no
pg_hba.conf entry for host "172.30.104.164", user "postgres", database "postgres", no encryption
```

The connection failed, which makes sense because by default, only Unix connections are allowed. Thus, we need to allow TCP connections by modifying two files. First, we need to take a look at the PostgreSQL configuration file and see what listen addresses are allowed. If the ip address is not allowed to be listened on, this needs to be changed.

Figure 11: postgres.conf File

```
#-----
# CONNECTIONS AND AUTHENTICATION
#-----

# - Connection Settings -

listen_addresses = '*'          # what IP address(es) to listen on;
                                # comma-separated list of addresses;
                                # defaults to 'localhost'; use '*' for all
                                # (change requires restart)
port = 5432                     # (change requires restart)
```

From the file in Figure 11, we can see that all IP addresses can be listened to while only port 5432 can be connected to. This is perfect, as this is exactly what we wanted. Now we need to check the pg_hba.conf file and see what changes need to be made to it.

Figure 12: pg_hba.conf File

#	TYPE	DATABASE	USER	ADDRESS	METHOD
# "local" is for Unix domain socket connections only					
local	all		all		scram-sha-256
# IPv4 local connections:					
host	all		all	127.0.0.1/32	scram-sha-256
# IPv6 local connections:					
host	all		all	::1/128	scram-sha-256

As can be seen in Figure 12, the server only allows connections from the local-host. This needs to be changed so that the requested ip address can connect to the server. We will also take note that both Unix connections and TCP/IP connections use password (scram-sha-256) authentication. We will have to modify the file as follows in Figure 13.

Figure 13: Fixing pg_hba.conf File

#	TYPE	DATABASE	USER	ADDRESS	METHOD
# "local" is for Unix domain socket connections only					
local	all		all		scram-sha-256
# IPv4 local connections:					
host	all		all	172.30.104.164/32	scram-sha-256
# IPv6 local connections:					
host	all		all	::1/128	scram-sha-256

Now we are only allowing connections from the host we specified. We are required to restart our PostgreSQL instance for these changes to be applied, and then we can try again.

Figure 14: Completed TCP/IP Connection

```
C:\Program Files\PostgreSQL\16\bin>psql -U postgres -d postgres -h 172.30.104.164
Password for user postgres:
psql (16.0)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# |
```

From Figure 14, it can be seen that we connected to the server using a TCP socket. If we wanted to try a different port we would have to change the allowed port in the postgresql.config file. This was a bit of a challenge to do, as it turns out one cannot simply connect to a random IP address as the host, it rather needs to be an IP address of a network one is connected to, thus why UWO's IP address was used. For all the future tests, we will be using a UNIX socket.

The next thing that was to be done was to create a new user and see what could be done. The new user would be created by the postgres user, as it is a superuser and therefore can bypass pretty much anything as previously mentioned. The user created was called "alex" and it was created with no optional arguments at all. This immediately caused an issue as seen below in Figure 15:

Figure 15: Failed No Password

```
postgres=# CREATE USER alex;
CREATE ROLE
postgres=# \q

C:\Program Files\PostgreSQL\16\bin>psql -U alex
Password for user alex:
psql: error: connection to server at "localhost" (:::1), port 5432 failed: fe
_sendauth: no password supplied
```

As we saw from attempting a TCP connection, the PostgreSQL server uses the password authentication method, thus because we did not specify a password, we cannot connect to the server using our new user at all. Therefore, we are required to go back and create a new password for the user. This actually led to another issue however, and that is involving attempting to connect to a database shown in Figure 16.

Figure 16: Failed With Password

```
postgres=# ALTER USER alex WITH PASSWORD 'cs4411a';
ALTER ROLE
postgres=# \q

C:\Program Files\PostgreSQL\16\bin>psql -U alex
Password for user alex:
psql: error: connection to server at "localhost" (:::1), port 5432 failed: FA
TAL: database "alex" does not exist
```

It appears as though PostgreSQL will automatically attempt to connect to a database of the same user's name by default, as it is looking to connect to the database "alex" which of course does not exist. Although it is possible to connect to a different database at the start, PostgreSQL requires one to connect to a database in order for the user to login, so in order for the user "alex" to be able to do anything, a database needs to be created for him or we need to specify a database to connect to. In this case we will create a new database just for the user.

Figure 17: New User Authenticated

```
postgres=# CREATE DATABASE project WITH OWNER alex;
CREATE DATABASE
postgres=# ALTER USER alex WITH CREATEDB;
ALTER ROLE
postgres=# \q

C:\Program Files\PostgreSQL\16\bin>psql -U alex -d project
Password for user alex:
psql (16.0)
WARNING: Console code page (437) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
Type "help" for help.

project=>
```

As can be seen in Figure 17, a new database named "project" was created by the superuser and given to alex, in addition to the ability to create other databases himself. Before this ability, alex would not have been able to create databases and would need to have all of them made for him. However, alex still has some restrictions when compared to the superuser, as the only thing alex is able to do is create and modify databases. Therefore, if this user attempts to create a new user themselves or attempts to remove a database they do not own, they will not be able to do so. There is already a database owned by postgres called "testdb", so if "alex" attempts to delete it, they will fail as in Figure 18.

Figure 18: Failed Delete

```
You are now connected to database "project" as user "alex".
project=> CREATE USER alex2;
ERROR:  permission denied to create role
DETAIL:  Only roles with the CREATEROLE attribute may create roles.
project=> DROP DATABASE testdb;
ERROR:  must be owner of database testdb
project=> \c testdb
You are now connected to database "testdb" as user "alex".
testdb=> |
```

There is nothing that stops "alex" from connecting to the "testdb" database though, so it looks as though "alex" can view the database and tables in it completely fine. However, it can be seen in Figure 19 that "alex" is not at all allowed to modify the database or write anything in it because again they are not the owner. However, this does not apply to postgres, as it has the superuser privilege access and is the owner of the database.

Figure 19: Postgres vs Alex

```
testdb=> ALTER DATABASE testdb RENAME TO testdb2;
ERROR:  must be owner of database testdb
testdb=> ALTER DATABASE testdb OWNER TO alex;
ERROR:  must be owner of database testdb
```

```
postgres=# ALTER DATABASE project RENAME TO project2;
ALTER DATABASE
postgres=# ALTER DATABASE project2 OWNER TO postgres;
ALTER DATABASE
```

It is clear that the roles one has is crucial to determine how much change one can do to the different databases. With an idea of access control, it was also decided to try and do the group functionality. Therefore, multiple new users were created by postgres in Figure 20 to do this.

Figure 20: Create New Users

```
postgres=# CREATE USER alex2 WITH PASSWORD 'cs4411a' CREATEDB;
CREATE ROLE
postgres=# CREATE USER alex3 WITH PASSWORD 'cs4411a' CREATEDB;
CREATE ROLE
postgres=# \du
```

Role name	Attributes
alex	Create DB
alex2	Create DB
alex3	Create DB
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS

Now each user only has the ability to create a database, thus they should be grouped for ease of access like in Figure 21.

Figure 21: Grouping

```
postgres=# CREATE GROUP alexnet;
CREATE ROLE
postgres=# ALTER GROUP alexnet ADD USER alex, alex2, alex3;
ALTER ROLE
postgres=# \dg
```

Role name	Attributes
alex	Create DB
alex2	Create DB
alex3	Create DB
alexnet	Cannot login
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS

Now to see if this will work, we will apply an access control method to one of the tables in the testdb database in Figure 22. There are the tables counties and maskuse, and

we will choose counties and will attempt to enable SELECT on this table to the group. Prior to this, no user would have been able to SELECT anything from the table as they were not the owner and did not have the privilege to.

Figure 22: Group Privilege

```
postgres=# \c testdb
You are now connected to database "testdb" as user "postgres".
testdb=# GRANT SELECT ON counties TO GROUP alexnet;
GRANT
```

After this, all three users in the group are able to select values from the counties table but not the maskuse table in Figure 23.

Figure 23: Granted Access

```
testdb=> \c testdb
You are now connected to database "testdb" as user "alex"
testdb=> SELECT COUNT(*) FROM counties;
 count
-----
 914027
(1 row)

testdb=> SELECT COUNT(*) FROM maskuse;
ERROR:  permission denied for table maskuse
```

```
testdb=> \c testdb
You are now connected to database "testdb" as user "alex2".
testdb=> SELECT COUNT(*) FROM counties;
 count
-----
 914027
(1 row)

testdb=> SELECT COUNT(*) FROM maskuse;
ERROR:  permission denied for table maskuse
```

```

testdb=> \c testdb
You are now connected to database "testdb" as user "alex3".
testdb=> SELECT COUNT(*) FROM counties;
 count
-----
 914027
(1 row)

testdb=> SELECT COUNT(*) FROM maskuse;
ERROR:  permission denied for table maskuse

```

The group functionality proves that roles can be added to multiple users at a time while ensuring only they receive the roles. In addition, it is clear that access control methods such as the GRANT and REVOKE commands work for all members of the group. One question may still arise though, and that is what will happen if only one user in a group had extra access to something and the others did not.

Figure 24: Individual Access

```

postgres=# \c testdb
You are now connected to database "testdb" as user "postgres".
testdb=# GRANT SELECT ON maskuse TO alex3;
GRANT

```

```

testdb=> \c testdb
You are now connected to database "testdb" as user "alex3".
testdb=> SELECT COUNT(*) FROM maskuse;
 count
-----
   3142
(1 row)

```

```

testdb=> \c testdb
You are now connected to database "testdb" as user "alex2".
testdb=> SELECT COUNT(*) FROM maskuse;
ERROR:  permission denied for table maskuse
testdb=> |

```

Therefore, if a privilege is given to a group, it will be given to all members of the group like in Figure 24. However, individual privileges that only some members of the group can have are still allowed provided the privilege is given to the user and not the group as a whole.

It was also decided to test writing a simple function with the postgres user. It was already established that functions can be a little bit sneaky and cause other users to become infected.

Figure 25: Function [34]

```
postgres=# CREATE FUNCTION add_numbers(a integer, b integer)
postgres-# RETURNS integer AS $$
postgres$$ BEGIN
postgres$$ RETURN a + b;
postgres$$ END; $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
```

The function in Figure 25 will obviously not cause someone to become infected, but is just to demonstrate how it should be made. This is a very simple function that just adds two numbers. In order to execute the function, it needs to be used in an SQL statement like the following: `SELECT * FROM add_numbers(3,8)`.

7 Query Evaluation

Query evaluation in PostgreSQL refers to the process through which the database management system (DBMS) processes and executes a user's query to retrieve or modify data stored in the database. Throughout this process, PostgreSQL aims to provide users with accurate and timely results while optimizing resource utilization. Efficient query evaluation is crucial for maintaining high-performance levels in database systems, especially when dealing with large data sets or complex queries. By continuously improving the query optimizer and execution engine, PostgreSQL strives to deliver reliable and responsive query processing for a wide range of applications.

In this section, we mainly focus on showing the significance of implementing query optimization strategies [35]. Since PostgreSQL uses a cost-based query optimizer to determine the most efficient way to execute a given SQL query, we use query cost as a performance metric. PostgreSQL allows us to use the **EXPLAIN** command to obtain information about the execution plan of a query [36].

In the following example, we will highlight two essential optimization strategies aimed at boosting the execution performance of queries in PostgreSQL. Our focus will be on query rewriting and adding indexes to achieve enhanced efficiency in query processing.

(Dataset Resources: Assignment 1 “counties”)

7.1 Query Rewriting

Query rewriting plays a crucial role in improving the performance of queries by transforming the original query into an equivalent, but more efficient, representation. First, query rewriting can simplify complex queries by applying various algebraic rules and transformations. This simplification can lead to a more straightforward and efficient execution plan. For a query containing a join operation, the order in which tables are joined in a query can have a substantial impact on performance. Query rewriting can explore different join order possibilities and select the one that minimizes the overall

cost. By rearranging the join order, the optimizer may find more efficient ways to access and combine the data, leading to faster query execution.

Figure 26: Original Query & Rewritten Query

<pre> cs4411=# Explain (FORMAT JSON, ANALYZE) cs4411=# SELECT DISTINCT cs4411=# state, cs4411=# AVG(cases) OVER (PARTITION BY state) AS avg_cases cs4411=# FROM counties; </pre>	<pre> cs4411=# EXPLAIN (FORMAT JSON, ANALYZE) cs4411=# SELECT state, AVG(cases) AS avg_cases cs4411=# FROM counties cs4411=# GROUP BY state; </pre>
<p>QUERY PLAN</p> <pre> [{ "Plan": { "Node Type": "Unique", "Parallel Aware": false, "Async Capable": false, "Startup Cost": 128462.36, "Total Cost": 223074.23, "Plan Rows": 91403, "Plan Width": 41, "Actual Startup Time": 323.647, "Actual Total Time": 670.258, "Actual Rows": 55, "Actual Loops": 1, "Plans": [{ </pre>	<p>QUERY PLAN</p> <pre> [{ "Plan": { "Node Type": "Aggregate", "Strategy": "Sorted", "Partial Mode": "Finalize", "Parallel Aware": false, "Async Capable": false, "Startup Cost": 15337.97, "Total Cost": 15352.32, "Plan Rows": 55, "Plan Width": 41, "Actual Startup Time": 98.384, "Actual Total Time": 99.101, "Actual Rows": 55, "Actual Loops": 1, "Group Key": ["state"], "Plans": [</pre>

In Figure 26, our goal is to find average cases of each state in the data set “counties.” The first query employed DISTINCT and PARTITION BY, leading to a total cost of 223074.23. However, after we rewrote the query to a simpler vision that only used GROUP BY, it significantly reduced the cost to 15352.32. Thus, even if two queries do the same thing and produce identical output, they could still be different in ways to retrieve or modify data. For example, we can see the plan rows reduced from 91403 to 55. While the original query processed every record in the data set, the rewritten query only processed 55 states. This is evidence that shows the importance of query rewriting in query optimization.

7.2 Using Index

Most of the time, we already have the query’s simplest version, but considering the complex query and huge data set could still be time-consuming and computationally expensive, we want to further optimize the query and enhance the performance by using the index.

Indexes improve query performance by providing a “shortcut” for a database engine to access and retrieve data. Instead of scanning entire tables, indexes create organized data structures that allow the engine to pinpoint specific rows that meet the criteria specified in a query’s WHERE clause. This reduces the number of I/O operations and accelerates data retrieval, particularly for SELECT, JOIN, ORDER BY, and GROUP BY operations. Indexes facilitate faster access to data by enabling the engine to bypass full table scans and directly navigate to the relevant subset of rows. That is, indexes can significantly boost read performance.

7.2.1 B Tree Index

Figure 27: Apply B Tree Index

```
cs4411=# CREATE INDEX idx_state ON counties(state);
CREATE INDEX
cs4411=# Explain (FORMAT JSON, ANALYZE)
SELECT DISTINCT
    state,
    AVG(cases) OVER (PARTITION BY state) AS avg_cases
FROM counties;

QUERY PLAN
-----
[
  {
    "Plan": {
      "Node Type": "Unique",
      "Parallel Aware": false,
      "Async Capable": false,
      "Startup Cost": 2347.57,
      "Total Cost": 145089.94,
      "Plan Rows": 91403,
      "Plan Width": 41,
      "Actual Startup Time": 83.148,
      "Actual Total Time": 583.836,
      "Actual Rows": 55,
      "Actual Loops": 1,
    }
  }
]
```

Comparing the result to the original query in Figure 27, the cost reduction from 223074 to 145098 indicates a more efficient execution of the query after the introduction of the B-tree index [37].

7.2.2 Hash Index

Figure 28: Apply Hash Index

```
cs4411=# CREATE INDEX idx_state_hash ON counties USING hash (state);
CREATE INDEX
cs4411=# Explain (FORMAT JSON, ANALYZE)
SELECT DISTINCT
    state,
    AVG(cases) OVER (PARTITION BY state) AS avg_cases
FROM counties;

QUERY PLAN

-----
[
  {
    "Plan": {
      "Node Type": "Unique",
      "Parallel Aware": false,
      "Async Capable": false,
      "Startup Cost": 128463.36,
      "Total Cost": 223075.23,
      "Plan Rows": 91403,
      "Plan Width": 41,
      "Actual Startup Time": 263.553,
      "Actual Total Time": 604.772,
      "Actual Rows": 55,
      "Actual Loops": 1,
    }
  }
]
```

We notice that the total cost has no change compared to the original query in Figure 28. This result suggests that the hash index may not have had a significant impact on query performance in terms of the execution plan and resource utilization.

7.2.3 Clustered B Tree Index

Clustering on an index is beneficial in database optimization because it involves physically organizing the table's data based on the order of the indexed columns. This optimization, often associated with clustered indexes, enhances data locality by grouping rows with similar index key values. As a result, consecutive rows are stored close to each other on the disk, reducing the number of disk I/O operations and leading to faster query performance. Range queries benefit significantly from clustering, as the database engine can efficiently scan the relevant portion of the clustered index. Furthermore, fewer page reads are required due to the physical ordering of data, and JOIN operations become more efficient when tables are clustered on the join key. The improved ORDER BY performance is notable, as the data is already sorted based on the clustered index key.

Figure 29: Apply Clustered B Tree Index

```
cs4411=#
cs4411=# CLUSTER counties USING idx_state;
CLUSTER
cs4411=# Explain (FORMAT JSON, ANALYZE)
SELECT DISTINCT
    state,
    AVG(cases) OVER (PARTITION BY state) AS avg_cases
FROM counties;

QUERY PLAN

[
  {
    "Plan": {
      "Node Type": "Unique",
      "Parallel Aware": false,
      "Async Capable": false,
      "Startup Cost": 2347.24,
      "Total Cost": 145072.01,
      "Plan Rows": 91403,
      "Plan Width": 41,
      "Actual Startup Time": 32.719,
      "Actual Total Time": 457.495,
      "Actual Rows": 55,
      "Actual Loops": 1,
    }
  }
]
```

In comparison to the unclustered B Tree index, implementing clustering slightly improved the performance. Both plan total cost and actual total time are reduced by cluster.

In Figure 29, query rewriting has the best performance. Generally, for complex queries involving multiple joins, aggregations, and conditions, query rewriting may be a suitable strategy. If the queries are relatively simple, indexes might be more effective. Different types of indexes serve distinct purposes, and their appropriateness depends on the distribution of the data and the types of queries.

8 Performance Tuning

Performance tuning is a critical aspect of database management, aimed at optimizing the speed, efficiency, and overall performance of the database system. Given the aforementioned features and performance considerations of PostgreSQL, there exists many tools to help database administrators streamline their analysis and decision making progress. The following parameters are referenced when fine tuning the PostgreSQL system for performance.

8.1 PostgreSQL Parameters for Tuning

8.1.1 Query Evaluation Engine Memory

The executor uses local memory for sorting tuples by ORDER BY and DISTINCT operations, and for joining tables by merge-join and hash-join operations. It additionally uses local memory for storing tables. The `work_mem` parameter specifies the memory size for the former operations, while `temp_buffers` specifies the latter [5].

The recommended starting point for `work_mem` is $((\text{Total RAM} - \text{shared_buffers}) / (16 \times \text{CPU cores}))$. The formula provides for a relatively large limit for the general case where performance is often CPU bound when there exists so many queries running which causes the system to run out of memory. Setting `work_mem` to a higher value should be avoided as the amount of memory specified may be used by each node within a single query plan, thus a single query could use multiples of `work_mem` in total [38].

8.1.2 Buffer Size

The `shared_buffers` parameter configures the size of the shared buffer pool. The parameter can yield varying degrees of performance. Some workloads work best with very small values even with very large database volumes while other workloads require large values. Conventional wisdom suggests that $\text{MIN}(\text{RAM}/2, 10\text{GB})$ is a good default value. Though, the intricate interactions between the kernel cache and `shared_buffers` make it difficult to describe why it is the case [38].

8.1.3 Parallel Query Parameters

The number of merging worker processes per query is limited by the `max_parallel_workers_per_gather` parameter which is set to two by default. Therefore, each Gather or GatherMerge operation can use at most two worker processes. A single query may benefit from increased parallelisation, and therefore performance, by increasing the amount of workers. An additional parallelisation parameter is `max_parallel_workers` which limits the number of active parallel worker processes in the whole database cluster rather than for one query [10].

8.1.4 WAL Configuration

Checkpoints can cause significant I/O load and reach performance bottlenecks when it frequently flushes all dirty pages. Therefore, checkpoint activity is throttled so that I/O begins at checkpoint start and completes before the next checkpoint is due to start which minimizes performance degradation during checkpoints [19].

A checkpoint begins whenever the first of `checkpoint_timeout` seconds or `max_wal_size` is about to be exceeded. Reducing these parameters causes checkpoints to occur more often. This allows faster after-crash recovery but increases the cost of flushing dirty data pages more often. If `full_page_writes` is set, there is another consideration: to ensure data page consistency, the first modification of a page after each checkpoint results in logging the entire page. Thus, a smaller checkpoint interval increases the volume of

output to the WAL, partially negating the goal of using a smaller interval and causing more disk I/O. Additionally, writing dirty buffers during a checkpoint is spread over a period of time controlled by `checkpoint_completion_target` to avoid flooding the I/O system with a burst of page writes [19].

8.2 pg_stat

8.2.1 pg_stat_statements

`pg_stat_statements` is an extension that is used to help optimize database queries. Basically, users can use it if they want to determine which queries have taken the largest total amount of time to execute, how frequently queries are getting executed, and the average time to execute a query. In order to use it, the extension has to be enabled in the database that is to be analyzed. This is done by executing the following command: `CREATE EXTENSION pg_stat_statements` [39]. In addition, some changes need to be made to the `postgresql.conf` file like in Figure 30:

Figure 30: Changes to Activate `pg_stat_statements` [40]

```
# - Shared Library Preloading -  
  
#local_preload_libraries = ''  
#session_preload_libraries = ''  
shared_preload_libraries = 'pg_stat_statements' # (change requires restart)  
#jit_provider = 'llvmjit' # JIT library to use
```

```
# - Monitoring -  
  
compute_query_id = on  
pg_stat_statements.max = 10000  
pg_stat_statements.track = all  
#log_statement_stats = off  
#log_parser_stats = off  
#log_planner_stats = off
```

With this done, the analysis can begin. It is important to note that `pg_stat_statements` is essentially just a view that contains data on all other queries. There is `userid`, `queryid`, `plan time`, `execution time`, and `calls` among other things. For instance, Figure 31 shows that one can determine which two queries have taken the greatest total amount of time

to execute and print their execution times, or perhaps see which queries have been run the greatest number of times:

Figure 31: Application of pg_stat_statements

```
cs4411=# SELECT query, total_exec_time
cs4411=# FROM pg_stat_statements
cs4411=# ORDER BY total_exec_time DESC
cs4411=# LIMIT 2
cs4411=# ;
```

query	total_exec_time
Explain (FORMAT JSON, ANALYZE) SELECT DISTINCT state, AVG(cases) OVER (PARTITION BY state) AS avg_cases FROM counties	4174.9725
Explain (FORMAT JSON, ANALYZE) SELECT DISTINCT state, AVG(cases) OVER (PARTITION BY state) AS avg_cases FROM counties;	4165.4844

(2 rows)

```
cs4411=# SELECT query, calls
cs4411=# FROM pg_stat_statements
cs4411=# ORDER BY calls DESC
cs4411=# LIMIT 3;
```

query	calls
SELECT COUNT(*) FROM counties	3
SELECT query, total_exec_time FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT \$1	2
SELECT COUNT(*) FROM maskuse	2

(3 rows)

As we can see from the last output of Figure 31, pg_stat_statements also includes queries made that involve itself. One other important note is that databases that the extension has not been authorized for can still have their data taken by pg_stat_statements. This tool is great for identifying queries that are taking longer than expected and can help one to decide on a proper query to use if such queries are equivalent. In addition, statistics can be reset by running a command if there is too much information. One downside is that queries executed on the side can still occur, and thus they may appear in the output if not properly dealt with.

8.2.2 pg_stat_activity

Another area of pg_stat is pg_stat_activity, which is used to show information relating to current activities. Thus, it can show information relating to queries that are in the process of executing, along with the users who attempted to execute these queries [41]. Since there is only one user at a time in our case, this is not really applicable but is good to know in the event multiple users were working on databases at the same time.

8.2.3 pg_stat_user_functions

This statistical record keeps information on executed functions provided tracking has been enabled. If it has, then the total number of times the function has been called will be kept track of as well as the total amount of time that has been spent in the function itself and other functions that are called by the function [41]. After modifying postgresql.config to keep track of functions, it was attempted to check this statistic by using it on the function created previously for adding two numbers.

Figure 32: Application of pg_stats_user_functions

```
postgres=# SELECT * FROM pg_stat_user_functions;
```

funcid	schemaname	funcname	calls	total_time	self_time
16537	public	add_numbers	3	2.078	2.078

(1 row)

8.3 pgMustard

pgMustard was started by Michael Christofides and David Conlin in 2018. The application aims to help users understand and analyze PostgreSQL query plans. Specifically, pgMustard simplifies and assesses the output of the EXPLAIN and ANALYZE commands, highlighting the most time-consuming operations and providing tips to making queries faster [42]. By offering a data-driven approach to query optimization, pgMustard contributes to improving database performance which is crucial for applications relying on PostgreSQL. Additionally, query metrics are presented in an intuitive and easy-to-understand manner rather than in the verbose output of PostgreSQL thereby streamlining the performance tuning process.

Upon navigating to the application, the user is prompted with inputting the query plan for review. Additionally, the statement which generates the plan must be prefixed using the EXPLAIN command with the stated parameters for the best results [42].

8.3.1 Annotated Plans

pgMustard visualizes plans with a “timing bar” and a tree structure as shown in Figure 33. The timing bar displays different-sized bars that represent the per-operation timings of each node in the query plan to help users immediately see at a glance which operations are the costliest [42].

The reading pane on the left hand side displays the total memory used and execution cost of the query in milliseconds. The pane also detailed information of each operation shown in Figure 33 and statistics about buffer usage as shown in Figure 34. The pane also includes various optimization tips if applicable. The tips are based on different aspects of the operation’s performance, such as the potential to discard additional rows through filtering operations like GROUP BY, and are scored on a scale of zero to five. The scores represent pgMustard’s estimations of the potential impact each modification could have on improving the query’s performance, with five being the highest impact and any aspect with that score should be heavily considered [42].

Figure 33: pgMustard overview



Figure 34: pgMustard buffer usage of an operation

Buffers: 68 MB

Shared hit blocks	577 (5 MB)
Shared read blocks	8,079 (63 MB)
Shared dirtied blocks	0
Shared written blocks	0
Local hit blocks	0
Local read blocks	0
Local dirtied blocks	0
Local written blocks	0
Temp read blocks	0
Temp written blocks	0

Figure 35: pgMustard operational details

▼ 🔍 Operation detail

Relation name	counties
Alias	counties
Filter	(counties.cases > 5000)
Rows removed by filter	281,083
Width	16 bytes
Parent relationship	Outer
Parallel aware	true
Schema	public
Actual startup time	0.03ms
Actual total time	27.28ms
Actual loops	3
Startup cost	0
Total cost	13,416.56
WAL records	0
WAL FPI	0
WAL bytes	0
Output	["counties.reportdate" "counties.county" "counties.state" "counties.fids" "counties.cases" "counties.deaths"]
Async capable	false

8.3.2 Query Examples Analysis

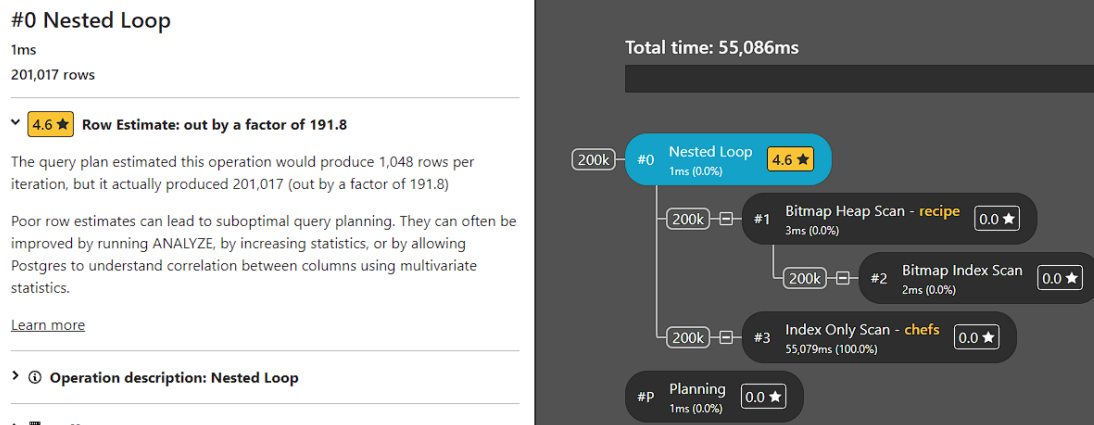
8.3.2.1 Inaccurate Row Estimate

PostgreSQL collects statistics about the table's contents when the user runs the ANALYZE command or the VACUUM command with the ANALYZE option. This process is automatically triggered by the auto-vacuum daemon by default [43]. Statistics include the most common values in a column and the distribution of values, which are calculated from a sample of the table's rows and stored in the database. The statistics are used to estimate the evaluation cost of a query by determining how many rows from your data will pass a given operation or filter [44]. Since the query planner uses row count estimates to choose between different query execution plans, if the estimates are inaccurate then it follows that the planner will ultimately choose an sub-optimal plan that runs slowly.

pgMustard calculates the difference between the planner's row estimates and the actual

returned rows of the query plan for all operation nodes. It will flag particularly bad row count estimates on slow parts of the query [42]. In Figure 35, pgMustard suggests to run ANALYZE, by increasing statistics, or by allowing Postgres to understand correlation between columns using multivariate statistics. Using ANALYZE will cause a cleanup of any deleted data and trigger a recalculation of statistics values to be more up-to-date and accurate. Multivariate statistics are gathered on the columns as a group, so that PostgreSQL can factor correlations between columns for its estimations.

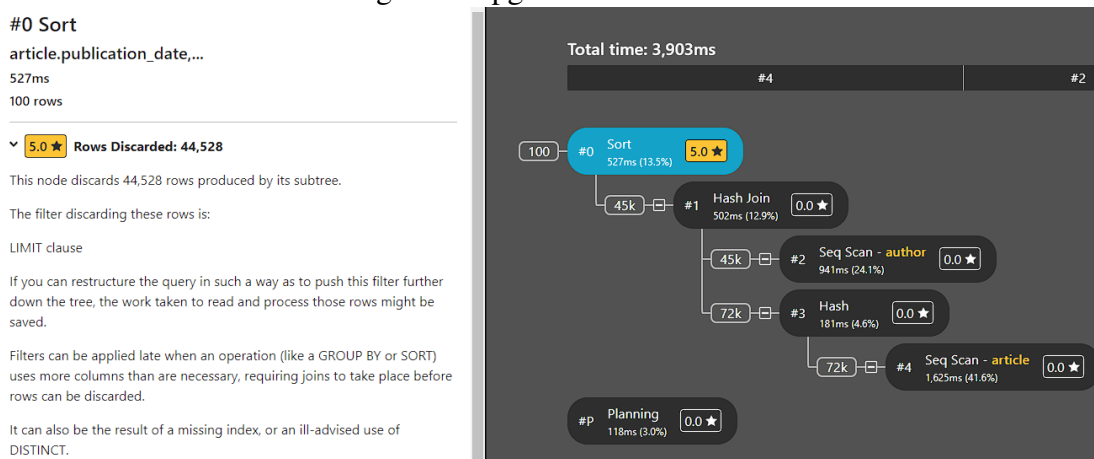
Figure 36: pgMustard poor row estimate



8.3.2.2 High Evolution Cost

Query execution plans with lower evolution costs yields better performance [44]. In Figure 36, pgMustard is able to calculate the proportion of rows discarded by a specific node and assess its significance. It will restructure the query so that the filters, operations, or nodes are executed first. By doing so, it reduces the workload of upper-level nodes by having to work on fewer tuples. Thus, pgMustard helps the user identify areas of query rewriting to improve performance.

Figure 37: pgMustard late filter



8.3.2.3 Poor Disk and Index Usage

An index can be used to retrieve the records in search key order by traversing the sequence set thereby improving performance by avoiding sorting altogether [44]. In Fig-

ure 37, it can be seen that pgMustard complies with this optimization by recommending adding an index on potentially inefficient operations. In that, it can help users identify when there is a sort or a sequential scan that discards a high proportion of the rows it reads.

pgMustard is additionally able to identify potential memory inefficiencies. One of the costliest operations is having to perform a sort or hash operation on disks as it is significantly slower than memory [44]. pgMustard suggests three strategies to improve performance in this scenario. In Figure 38, pgMustard suggests to reduce the amount of memory consumed by the operation, to reduce the number of rows with a clause, reducing the size of each row by operating on fewer columns, using data types with a smaller footprint, or increasing the amount of memory available for this operation by adjusting parameters like `work_mem`.

Figure 38: pgMustard sorting on disk

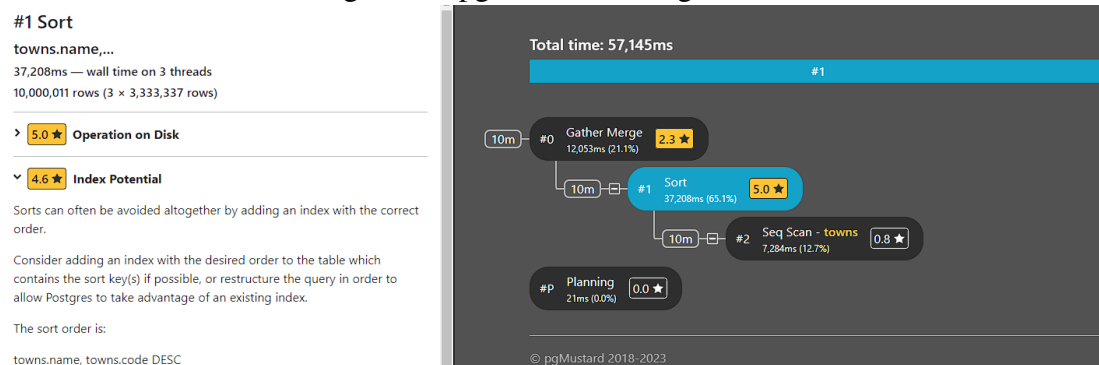
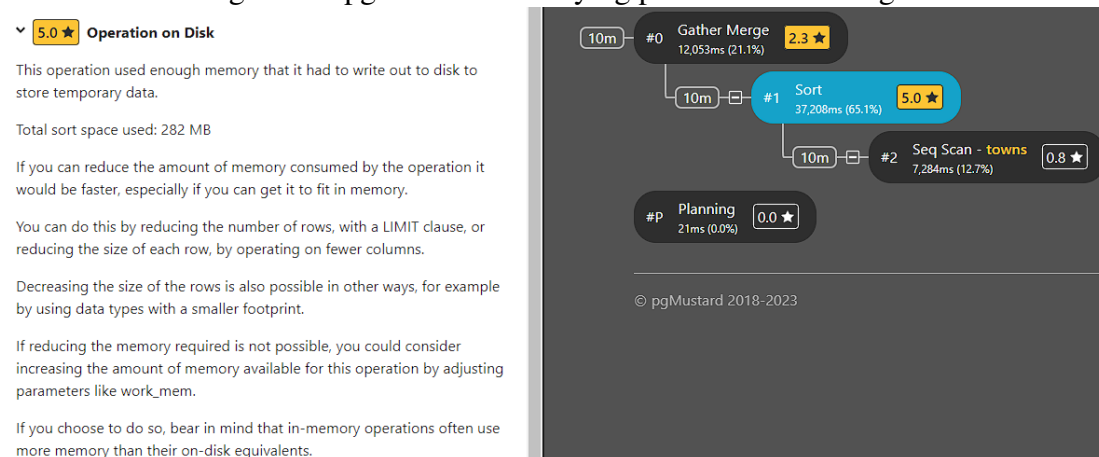


Figure 39: pgMustard identifying potential index usage



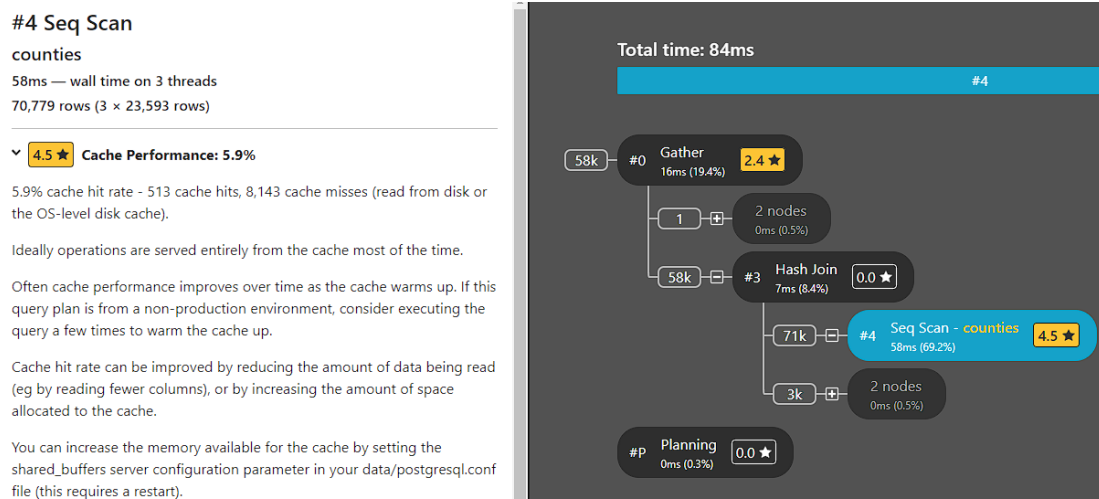
8.3.2.4 Increasing Parallelism

The query plan for the following two cases can be found as plain-text in the `pgMustard_input.txt` file with data being retrieved copied from the `mask-use-by-county.csv` and `us-counties.csv`. The query plan is from inputting the command:

```
SELECT a.fids, a.always, b.cases
FROM (SELECT fids, always FROM maskuse
WHERE always > (SELECT AVG(always) FROM maskuse)) a
JOIN (SELECT cases,fids FROM counties WHERE cases > 5000) b
ON (a.fids = b.fids);
```

The command selects from a joined table consisting of two columns. The always column from maskuse, where the percentage of people that always wear a mask is higher than the average, is joined with the cases column from counties where the number of cases exceeds 5000. In Figure 39, pgMustard is able to identify if an operation uses the maximum number of background worker processes, thus helping the user identify any bottlenecks in parallel querying from resource limitations. It provides optimization tips for both transactional and analytical queries which have different use cases. The strategies are to either reduce the amount of work being performed for transactional queries and increase the resources available for analytical queries (or by changing the parallel parameters). Additionally, it states which parameters can be changed to realize these optimizations.

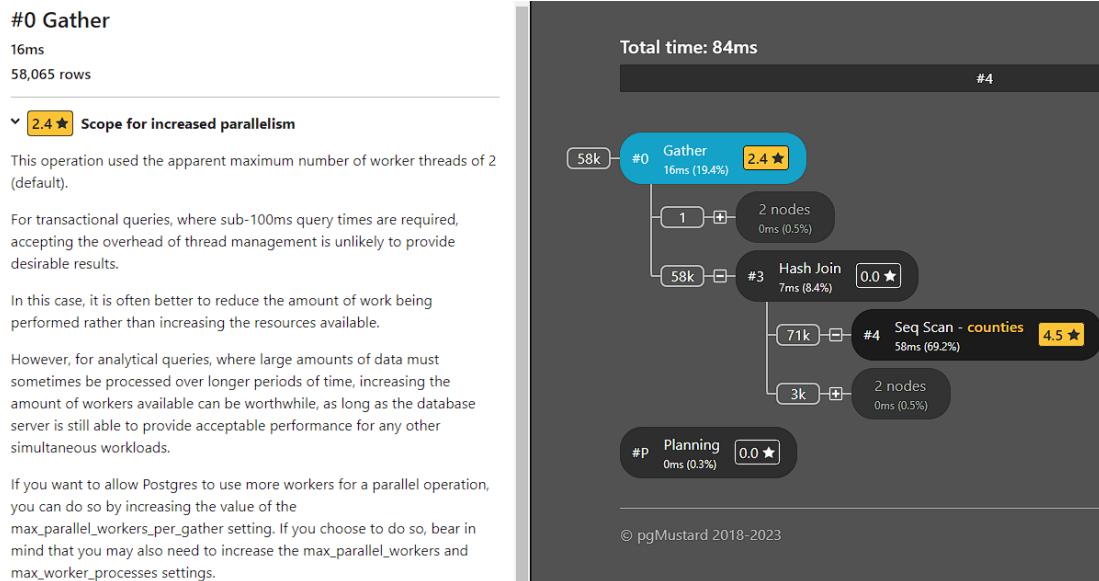
Figure 40: pgMustard identifying worker limit reached



8.3.2.5 Identifying Cache Misses

Cache operations are faster than even memory operations, thus operations should aim to read from cache for the most efficiency [44]. In Figure 40, pgMustard suggests several optimizations for improving cache performance in this query. Firstly, cache performance often improves as the cache “warms up” over time. Moreover, if the query plan is from a non-production environment, it’s suggested to execute the query multiple times to warm up the cache. Cache hit rate can be increased by reducing the amount of data being read (for example, by reading fewer columns) or by increasing the space allocated to the cache. One way to increase the memory available for the cache is by increasing the `shared_buffers` server configuration parameter.

Figure 41: pgMustard assessing cache performance



8.3.3 pgMustard Case Study

In this section, some of the query plans from the query analysis section were recreated and fed into pgMustard to see just how pgMustard handles them. Both of the plans from the explain commands from the query evaluation section (Figure 26) were recreated along with the index enhanced version (Figure 27) and fed into pgMustard.

Figure 42: pgMustard Assessing First Query Plan

Summary

Total time: 4,288ms

Top tips

5.0 ★

Operation #3: Operation on Disk

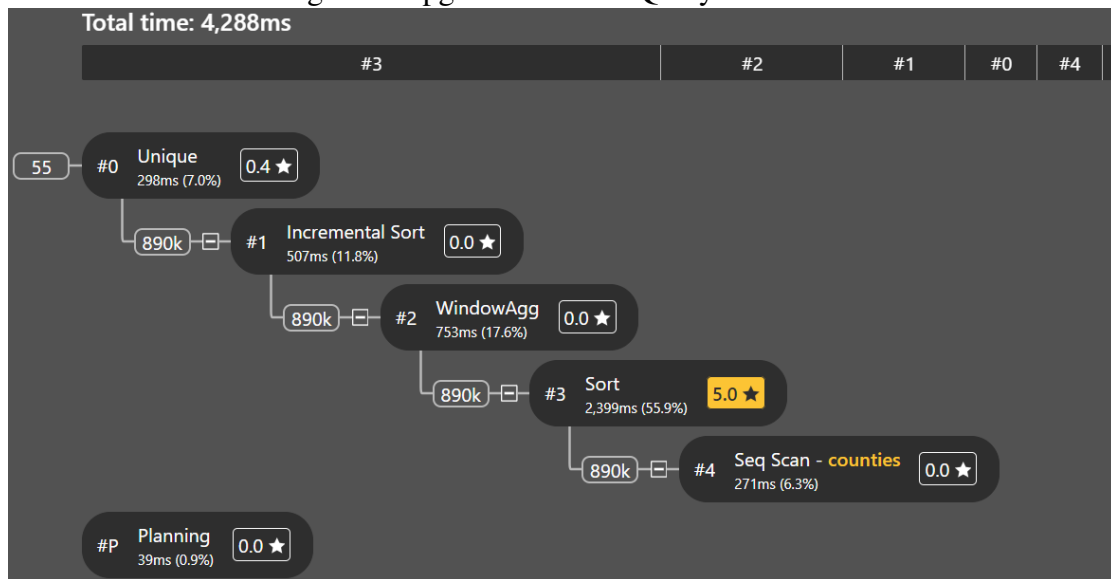
3.9 ★

Operation #3: Index Potential

0.4 ★

Operation #0: Row Estimate: out by a factor of 1,661.9

Figure 43: pgMustard First Query Plan Tree



From Figure 42 and 43, it can be seen that the sort operation performed extremely poorly and that so much memory was used that external memory was required to store all of the data. The recommendation states that the amount of memory should attempt to be reduced somehow, perhaps by reducing the number of rows or shortening their size. In addition, the recommendations also indicate that there is a potential for index use here and to try and use it on the sort key “state,” which we end up doing.

Figure 44: pgMustard Assessing Second Query Plan

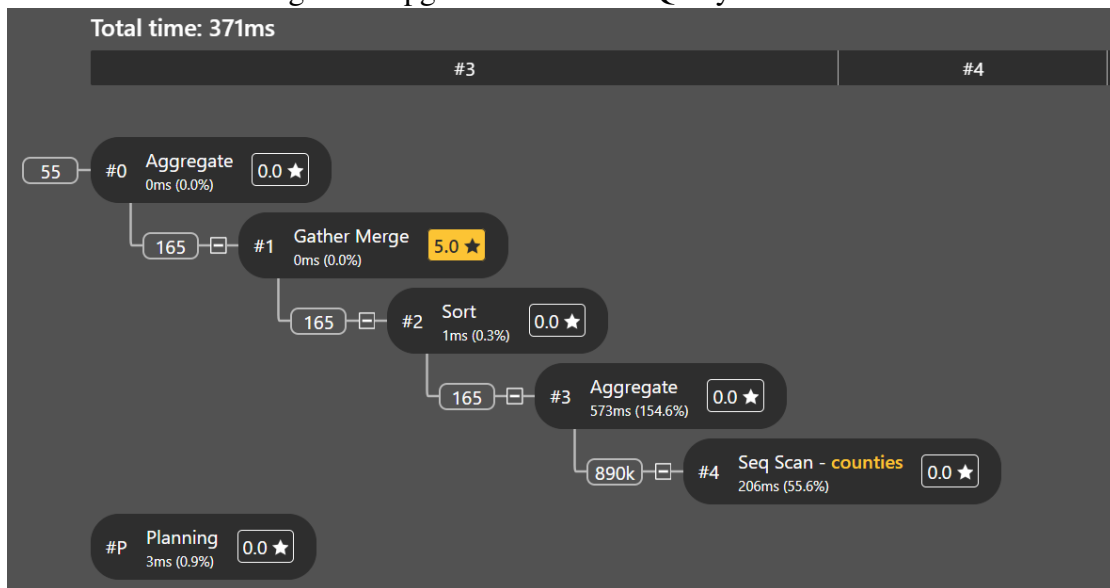
Summary

Total time: 371ms

Top tips

5.0 ★ Operation #1: Scope for increased parallelism

Figure 45: pgMustard Second Query Plan Tree



As we witnessed in the query execution section from Figure 44 and 45, this area performed better but still had one major area of improvement according to pgMustard. In the Gather Merge operation, the greatest number of worker threads were used, and it was recommended to try and reduce this number to lessen the amount of work being done. However, there was also the notice that having a great number of workers can still be alright if a large amount of data is input to the server.

Figure 46: pgMustard Assessing Second Query Plan with Index

#0 Unique

297ms

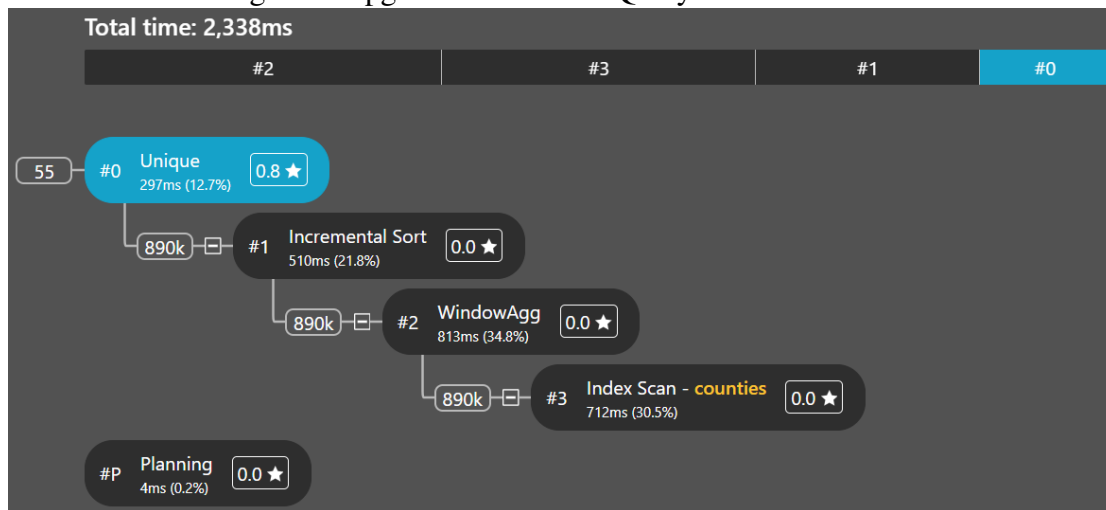
55 rows

▼ **0.8 ★** Row Estimate: out by a factor of 1,624.3

The query plan estimated this operation would produce 89,337 rows per iteration, but it actually produced 55 (out by a factor of 1,624.3)

Poor row estimates can lead to suboptimal query planning. They can often be improved by running ANALYZE, by increasing statistics, or by allowing Postgres to understand correlation between columns using multivariate statistics.

Figure 47: pgMustard Second Query Tree with Index



For the first query with the index, the row estimate was very off. This was pretty much the only issue though and the star is less than 1.0, so it still performed well and pgMustard agrees in Figure 46 and 47. This also proved that pgMustard was right in the first query plan by recognizing the opportunity for an index. pgMustard can take a poor query plan and give accurate recommendations on how to improve it, and users can take this knowledge and improve the query plans until they cannot get any better.

8.4 dbForge Studio for PostgreSQL

dbForge Studio is a universal front-end client for database management, administration, and development developed by Devart. The software provides utilities to compare, synchronize, and back up databases. It also allows users to visually design database structures, execute SQL queries and scripts, and manage users and privileges. dbForge Studio has received praise for its wide range of use cases and valuable features and is used by developers and database administrators for various tasks including query management, query editing, and database design [45].

Although dbForge Studio has many features, the report will only showcase the query profiler as it relates to performance tuning. Nonetheless, there are many reasons why dbForge Studio is a great tool for performance tuning. It helps in recognizing the usage of memory, disk, and CPU which are major factors in database performance. Moreover, dbForge Studio aids in configuring server or system parameter options which increase the performance of background processes, queries, and indexes [45]. Thus, dbForge Studio helps in optimizing the database at several levels, ensuring the stability, reliability, and speed of database-driven applications.

8.4.1 Query Profiler

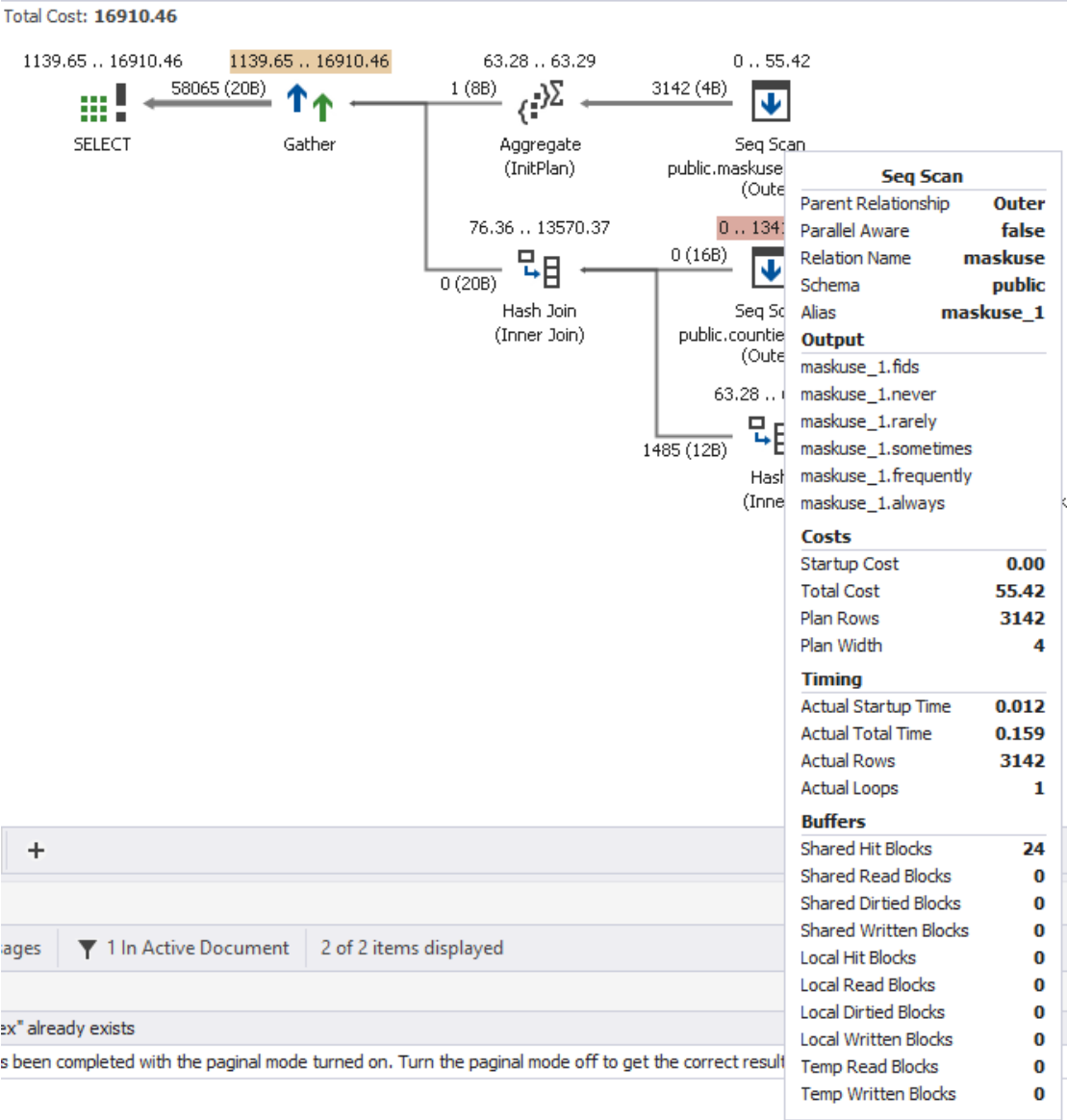
dbForge Studio includes a query profiling tool that allows users to quickly get the PostgreSQL execution plan and query execution statistics in a visual, comprehensive format [45]. With the profiler, the user can systematically identify operations or queries

that create performance bottlenecks similar. The functionality includes the following: [45]:

8.4.1.1 Execution Plan Diagram

The execution plan diagram is a graphical tool that visualizes the efficiency of queries. An example of the diagram is shown in Figure 48. Each query in the batch being analyzed is displayed with the cost of each query as a fraction of the total batch cost. Hovering over a specific operation or node reveals more information such as the estimated and actual costs, tuples or columns returned as output, node relationships, and buffer usage. The diagram also highlights particularly slow operations with red being the slowest or being the highest batch cost, and green being lower.

Figure 48: pgMustard dbForge Execution Plan Diagram example



8.4.1.2 Plan Tree

Plan tree displays EXPLAIN results returned by the executed query. As seen in Fig-

ure 49, the statements and operations are hierarchically arranged in a tree view. The corresponding database objects, estimated cost, and rows of a given statement are also displayed. Users can discriminate between the operations that use the most resources and time to better understand which parts operations need optimizations. Additionally, the hierarchical view can aid users in identifying where indexes could be inserted to improve performance.

Figure 49: dbForge Plan Tree example

Node Type	Relation Name	Startup Cost	Total Cost	Actual Rows	Plan Rows	Plan Width	Actual Startup Time	Actual Total Time	Actual Loops	Parallel Aware
▼ Gather		1139.65	16910.46	58065	22768	20	1.106	71.972	1	false
▼ Aggregate		63.28	63.29	1	1	8	0.449	0.450	1	false
Seq Scan	public.maskuse maskuse_1	0.00	55.42	3142	3142	4	0.012	0.159	1	false
Hash Join		76.36	13570.37	0	9487	20	0.002	0.002	1	false
Seq Scan	public.counties counties	0.00	13416.56	0	29468	16	0.001	0.001	1	true
Hash		63.28	63.28	1485	1047	12	0.415	0.415	1	false
Seq Scan	public.maskuse maskuse	0.00	63.28	1485	1047	12	0.065	0.309	1	false

8.4.1.3 Comparison for the Query Profiling Results

Users are able to compare profiles to see the differences in performance. Differences in performance are highlighted where improved results are marked in green, while declines in performance are marked in red. This method helps in identifying the impact of changes made to the query.

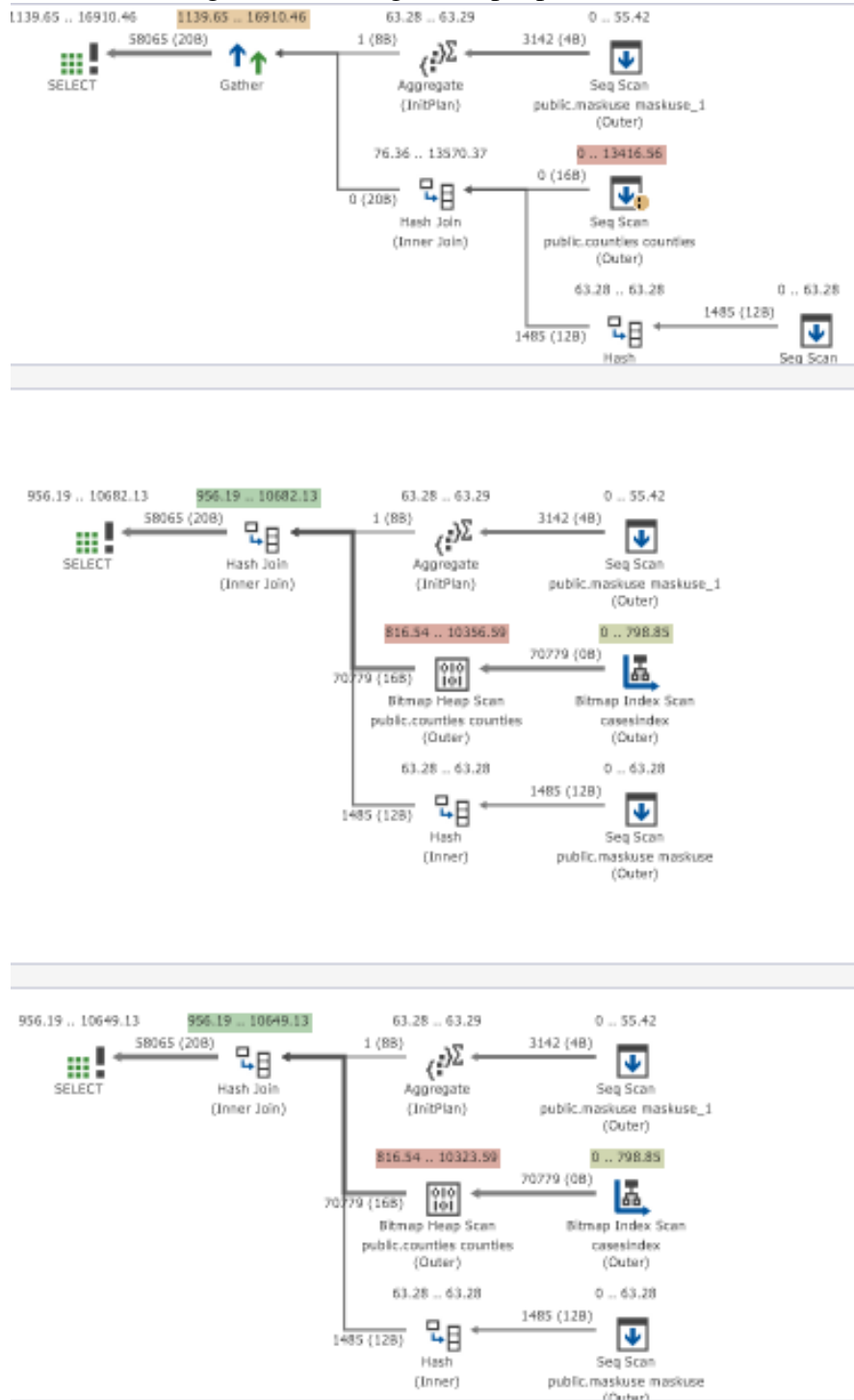
In Figure 50, the query from the pgMustard example is executed three times. However, the middle diagram is from the execution of the query after adding a B+tree index using the command:

```
CREATE INDEX casesindex ON counties USING BTREE (cases asc nulls last);
```

The bottom diagram shows the plan execution of a clustered B+tree index using the command:

```
CLUSTER counties USING casesindex;
```

Figure 50: dbForge multiple profile views



Finally, the top plan is executing the query with no index. Clearly, the root operation in the top diagram executes slower, which agrees with the colour distinction going from orange to green. Moreover, the bitmap index scan is faster than a the sequential scan of the disk which is also reflected in the diagram.

There is a significant improvement from clustering the index by the search key as well. In Figure 51, we can see the Shared Read Blocks statistics which represents the number of blocks that were read from disk to perform the query. The number of disk reads were

significantly lower in the clustering profile.

Figure 51: dbForge effects of clustering

Bitmap Heap Scan		Bitmap Heap Scan	
Parent Relationship	Outer	Parent Relationship	Outer
Parallel Aware	false	Parallel Aware	false
Exact Heap Blocks	7291	Exact Heap Blocks	676
Lossy Heap Blocks	0	Lossy Heap Blocks	0
Relation Name	counties	Relation Name	counties
Schema	public	Schema	public
Alias	counties	Alias	counties
Rows Removed by Index Recheck	0	Rows Removed by Index Recheck	0
Recheck Cond		Recheck Cond	
(counties.cases > 5000)		(counties.cases > 5000)	
Output		Output	
counties.reportdate		counties.reportdate	
counties.county		counties.county	
counties.state		counties.state	
counties.fids		counties.fids	
counties.cases		counties.cases	
counties.deaths		counties.deaths	
Costs		Costs	
Startup Cost	816.54	Startup Cost	816.54
Total Cost	10356.59	Total Cost	10323.59
Plan Rows	70724	Plan Rows	70724
Plan Width	16	Plan Width	16
Timing		Timing	
Actual Startup Time	4.064	Actual Startup Time	2.095
Actual Total Time	49.770	Actual Total Time	9.451
Actual Rows	70779	Actual Rows	70779
Actual Loops	1	Actual Loops	1
Buffers		Buffers	
Shared Hit Blocks	1597	Shared Hit Blocks	64
Shared Read Blocks	5825	Shared Read Blocks	740
Shared Dirtied Blocks	0	Shared Dirtied Blocks	0
Shared Written Blocks	0	Shared Written Blocks	0
Local Hit Blocks	0	Local Hit Blocks	0
Local Read Blocks	0	Local Read Blocks	0
Local Dirtied Blocks	0	Local Dirtied Blocks	0
Local Written Blocks	0	Local Written Blocks	0
Temp Read Blocks	0	Temp Read Blocks	0
Temp Written Blocks	0	Temp Written Blocks	0

9 Conclusion

PostgreSQL's design puts an emphasis on extensibility and adherence to SQL standards which allows for complex data types. Consequently, PostgreSQL offers a wide array of advanced functionalities. Features such as parallel querying, advanced indexing, and partitioning contribute to PostgreSQL's robustness and scalability.

Along with its functionalities, there are several aspects of the architecture of PostgreSQL which need to be understood so that administrators can create solutions that configure and optimize the database for performance. Notably, data and memory management for processes can be a significant performance bottleneck. For example, one optimization is in determining the correct balance between check-pointing frequency and post-crash recovery speed.

To help in the performance tuning process, the report provides insight into three major tuning tools. The tools aim to provide insight into database performance so that users can enhance the efficiency and speed of their applications.

References

- [1] PostgreSQL. About. <https://www.postgresql.org/about/>, 2023. Accessed: 2023-12-20.
- [2] Stonebraker, M. and Rowe, L.A. THE DESIGN OF POSTGRES. Technical report, University of California, June 1986. <https://dsf.berkeley.edu/papers/ERL-M85-95.pdf>. Accessed: 2023-12-20.
- [3] Simkovics, S. *Enhancement of the ANSI SQL Implementation of PostgreSQL*, November 1998.
- [4] PostgreSQL 16.1 Documentation. How Connections Are Established. <https://www.postgresql.org/docs/current/connect-estab.html>, 2023. Accessed: 2023-12-20.
- [5] Suzuki, H. *The Internals of PostgreSQL for database administrators and system developers*. Interdb, 2015.
- [6] PostgreSQL 16.1 Documentation. The Parser Stage. <https://www.postgresql.org/docs/current/parser-stage.html>, 2023. Accessed: 2023-12-20.
- [7] PostgreSQL 16.1 Documentation. The PostgreSQL Rule System. <https://www.postgresql.org/docs/current/rule-system.html>, 2023. Accessed: 2023-12-20.
- [8] PostgreSQL 16.1 Documentation. Planner/Optimizer. <https://www.postgresql.org/docs/current/planner-optimizer.html>, 2023. Accessed: 2023-12-20.
- [9] PostgreSQL 16.1 Documentation. Executor. <https://www.postgresql.org/docs/current/executor.html>, 2023. Accessed: 2023-12-20.
- [10] PostgreSQL 16.1 Documentation. Parallel Query. <https://www.postgresql.org/docs/current/parallel-query.html>, 2023. Accessed: 2023-12-20.
- [11] Amazon Web Services. What is database sharding? <https://aws.amazon.com/what-is/database-sharding/>, 2023. Accessed: 2023-12-20.
- [12] PostgreSQL 16.1 Documentation. Table Partitioning. <https://www.postgresql.org/docs/current/ddl-partitioning.html>, 2023. Accessed: 2023-12-20.
- [13] Sharding Your Data With PostgreSQL 11: Using partitioning and foreign data wrappers. <https://pgdash.io/blog/postgres-11-sharding.html>, 2023. Accessed: 2023-12-20.
- [14] Ives, A. PostgreSQL 11 sharding with foreign data wrappers and partitioning. 2023. https://about.gitlab.com/handbook/engineering/infrastructure/core-platform/data_stores/database/doc/fdw-sharding.html. Accessed: 2023-12-20.

- [15] PostgreSQL 7.1 Documentation. Multi-Version Concurrency Control. <https://www.postgresql.org/docs/7.1/mvcc.html>, 2023. Accessed: 2023-12-20.
- [16] Gymafi, L. Getting Started with Multiversion Concurrency Control (MVCC) in PostgreSQL. 2023. <https://www.dbvis.com/thetable/getting-started-with-multiversion-concurrency-control-mvcc-in-postgresql/#:~:text=In%20simple%20terms%2C%20MVCC%20creates,without%20conflicting%20with%20one%20another>. Accessed: 2023-12-20.
- [17] PostgreSQL 16.1 Documentation. Routine Vacuuming. <https://www.postgresql.org/docs/current/routine-vacuuming.html>, 2023. Accessed: 2023-12-20.
- [18] PostgreSQL 16.1 Documentation. Resource Consumption. <https://www.postgresql.org/docs/current/runtime-config-resource.html>, 2023. Accessed: 2023-12-20.
- [19] PostgreSQL 16.1 Documentation. Write-Ahead Logging (WAL). <https://www.postgresql.org/docs/current/wal-intro.html>, 2023. Accessed: 2023-12-20.
- [20] Tabassam, R. What are the types of index access methods in PostgreSQL? 2023. https://www.educative.io/answers/what-are-the-types-of-index-access-methods-in-postgresql?utm_campaign=interview_prep&utm_source=google&utm_medium=ppc&utm_content=pmax&utm_term=&eid=5082902844932096&utm_term=&utm_campaign=%5BNew-Oct+23%5D+Performance+Max+-+Coding+Interview+Patterns&utm_source=adwords&utm_medium=ppc&hsa_acc=5451446008&hsa_cam=20684486602&hsa_grp=&hsa_ad=&hsa_src=x&hsa_tgt=&hsa_kw=&hsa_mt=&hsa_net=adwords&hsa_ver=3&gad_source=1&gclid=EAIaIQobChMI4pftoP-jgwMVS0lHAR0-UwAwEAAYASAAEgIf9fD_BwE. Accessed: 2023-12-22.
- [21] Percona. What is HASH Index? (Index Types) - Deep Dive Into PostgreSQL Indexes Course. 2023. <https://www.youtube.com/watch?v=xVtkS4D2ZLI&list=PLWhC0zeznqkla9TwsihDz--n5c0Ik6YgS&index=8>. Accessed: 2023-12-22.
- [22] PostgreSQL 7.0 Documentation. Security. <https://www.postgresql.org/docs/7.0/security.htm>, 2000. Accessed: 2023-12-20.
- [23] PostgreSQL 16.1 Documentation. psql. <https://www.postgresql.org/docs/current/app-psql.html>, 2023. Accessed: 2023-12-20.
- [24] PostgreSQL 16.1 Documentation. The pg_hba.conf File. <https://www.postgresql.org/docs/current/auth-pg-hba-conf.html>, 2023. Accessed: 2023-12-20.
- [25] PostgreSQL 16.1 Documentation. Authentication Methods. <https://www.postgresql.org/docs/current/auth-methods.html>, 2023. Accessed: 2023-12-20.

- [26] PostgreSQL 16.1 Documentation. Trust Authentication. <https://www.postgresql.org/docs/current/auth-trust.html>, 2023. Accessed: 2023-12-20.
- [27] PostgreSQL 16.1 Documentation. Password Authentication. <https://www.postgresql.org/docs/current/auth-password.html>, 2023. Accessed: 2023-12-20.
- [28] PostgreSQL 16.1 Documentation. createuser. <https://www.postgresql.org/docs/current/auth-password.html>, 2023. Accessed: 2023-12-20.
- [29] PostgreSQL 16.1 Documentation. CREATE GROUP. <https://www.postgresql.org/docs/current/sql-creategroup.html>, 2023. Accessed: 2023-12-20.
- [30] PostgreSQL 16.1 Documentation. Role Attributes. <https://www.postgresql.org/docs/current/role-attributes.html>, 2023. Accessed: 2023-12-20.
- [31] PostgreSQL 16.1 Documentation. Privileges. <https://www.postgresql.org/docs/current/ddl-priv.html>, 2023. Accessed: 2023-12-20.
- [32] PostgreSQL 16.1 Documentation. Drop Role. <https://www.postgresql.org/docs/current/sql-droprole.html>, 2023. Accessed: 2023-12-20.
- [33] PostgreSQL 16.1 Documentation. Function Security. <https://www.postgresql.org/docs/current/perm-functions.html>, 2023. Accessed: 2023-12-20.
- [34] Rathbone, M. How to Create a Custom Function in PostgreSQL. <https://www.beekeeperstudio.io/blog/postgresql-create-function>, March 15, 2023. Accessed: 2023-12-21.
- [35] Node Team. How to Optimize PostgreSQL Queries. 2023. <https://nodeteam.medium.com/how-to-optimize-postgresql-queries-226e6ff15f72>. Accessed: 2023-12-22.
- [36] EDB Team. How to Use EXPLAIN ANALYZE for Planning and Optimizing Query Performance in PostgreSQL. March 30, 2023. <https://www.enterprisedb.com/blog/postgresql-query-optimization-performance-tuning-with-explain-analyze>. Accessed: 2023-12-22.
- [37] PostgreSQL 16.0 Documentation. CREATE INDEX. <https://www.postgresql.org/docs/current/sql-createindex.html>, 2023. Accessed: 2023-12-22.
- [38] Fearing, V. An Introduction to PostgreSQL Performance Tuning and Optimization. <https://www.enterprisedb.com/postgres-tutorials/introduction-postgresql-performance-tuning-and-optimization#sharedbuffers>, 2023. Accessed: 2023-12-20.

- [39] Batuigas, K. Query Optimization in Postgres with pg_stat_statements. <https://www.crunchydata.com/blog/tentative-smarter-query-optimization-in-postgres-starts-with-pg-stat-statements>, February 19, 2021. Accessed: 2023-12-21.
- [40] PostgreSQL 16.0 Documentation. pg_stat_statements — track statistics of SQL planning and execution. <https://www.postgresql.org/docs/16/pgstatstatements.html>, 2023. Accessed: 2023-12-21.
- [41] PostgreSQL 16.0 Documentation. The Cumulative Statistics System. <https://www.postgresql.org/docs/current/monitoring-stats.html#MONITORING-PG-STAT-USER-FUNCTIONS-VIEW>, 2023. Accessed: 2023-12-22.
- [42] pgMustard. <https://www.pgmustard.com/>. Accessed: 2023-12-20.
- [43] PostgreSQL 16.1 Documentation. ANALYZE. <https://www.postgresql.org/docs/current/sql-analyze.html>, 2023. Accessed: 2023-12-20.
- [44] Ramakrishnan, R and Gehrke, J. *Database Management Systems 3rd Edition*. McGraw-Hill, 2002.
- [45] Devart. dbForge Studio for PostgreSQL. <https://www.devart.com/dbforge/postgresql/>. Accessed: 2023-12-20.