

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

Costly Calculations

A Deep Dive into Cost Estimation Algorithms for
Optimised Query Execution in PostgreSQL

PROJECT 2

**AY 2023/24 SEMESTER 2
CZ4031 - Database System Principles**

Delivered by—

Pathak Siddhant (U2023715K)

Mahtolia Ronan (U2023144J)

Adrian Ang Jun Hao (U2121798A)

Douglas Toh Zheng Xun (U2121653B)

Submitted on —

April 21, 2024

Table of Contents

Costly Calculations	1
Table of Contents	2
1. Introduction	3
2. Background	3
2.1 Query Execution Plan (QEP)	3
2.2 Need for QEP optimization	4
2.3 Introduction to PostgreSQL	4
3. Experimental Setup	4
3.1 Dataset for evaluation	4
3.2 Software Setup	5
3.2.2 Technologies Used	6
3.3 Understanding cost estimation by PostgreSQL EXPLAIN	6
3.3.1 Commonly used notations, abbreviations	6
3.3.2 Query Operators	8
3.3.2.1 Sequential Scan	8
3.3.2.2 Index Scan	8
3.3.2.3 Index-Only Scan	9
3.3.2.4 Bitmap Heap Scan (Multi-Index Scan)	9
3.3.2.5 Bitmap Index Scan (Multi-Index Scan)	10
3.3.2.6 Aggregate	10
3.3.2.7 Sort	11
3.3.2.8 Hash	11
3.3.2.9 Gather	11
3.3.2.10 Gather Merge	12
3.3.2.11 Limit	12
3.3.2.12 Hash Join	13
3.3.2.13 Nested Loop Join	13
3.3.2.14 Merge Join	14
4. Discussion	15
4.1 Results	15
4.2 Limitations	17
5. Conclusion	18
7. Appendix	18
A: Worked Examples	18
B: Lo-fi mockups for user interface	19
C: Usage of LLMs	19

1. Introduction

In this project we focused on investigating and enhancing database query performance and the optimization of Query Execution Plans (QEP) in PostgreSQL. The following report encompasses our algorithm design and implementation used, alongside with their cost estimation and performance for several SQL queries. To effectively demonstrate and evaluate our results, we made use of a front-end interface, Gradio. With Gradio it allowed us to test the SQL queries in tandem with the PostgreSQL database interactively and providing us with valuable insights.

2. Background

2.1 Query Execution Plan (QEP)

A Query Execution Plan is a detailed roadmap that a database management system (DBMS) that is created efficiently for the execution of a query. It outlines the steps necessary for the retrieval of requested data. This plan includes information about how tables will be scanned, which joins will be used, indexes utilized, and in what order. (Refer to Figures below)

Query Execution

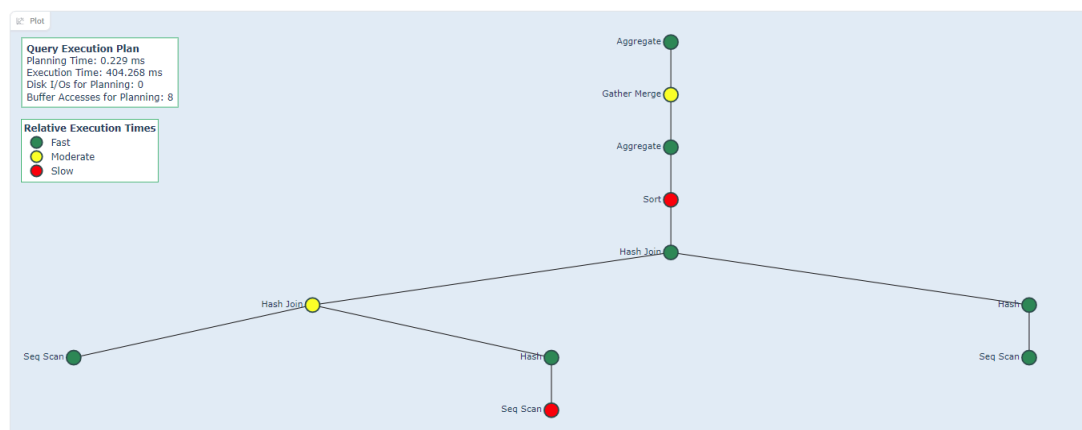
Query

```
select
  n.n_name as customer_nation,
  o.o_orderstatus as order_status,
  count(*) as total_orders,
  sum(o.o_totalprice) as total_revenue
from
  orders o
join
  customer c on o.o_custkey = c.c_custkey
join
  nation n on c.c_nationkey = n.n_nationkey
where
  extract(year from o.o_orderdate) = 1995
group by
  customer_nation,
  order_status
order by
  customer_nation,
  order_status;
```

Execute query

Example Query Executed

QEP Tree



Example of QEP Tree generated from above executed Query

2.2 Need for QEP optimization

QEP optimization is essential as it helps enhance the performance of database queries. Additionally, It helps to reduce the time it takes to execute a query by ensuring that the database can retrieve and process data in the least amount of time while utilizing the least amount of resources. Furthermore, having an optimized query plan makes better use of computational resources, such as CPU and memory. They also optimize data patterns which reduce disk I/O operation.

2.3 Introduction to PostgreSQL

PostgreSQL is an open-source objection relation database management system emphasizing extensibility and standards compliance. Its primary function is to store data securely and the retrieval of data. Additionally, it has a strong reputation for its proven architecture, reliability, data integrity and robust feature set. It is also widely used by many large companies and organizations as it has the ability to handle complex queries and extensive data types, including JSON and hstore (key-value store within a database). Furthermore, it supports advanced data types and performance optimization features.

PostgreSQL also includes tools such as **EXPLAIN** command, which is used to analyze and understand the query execution plan for a SQL query. When a prefix is used with a query, PostgreSQL will return information about how the database would execute the query, including details on the path the query planner considers and the estimated cost associated with each path.

3. Experimental Setup

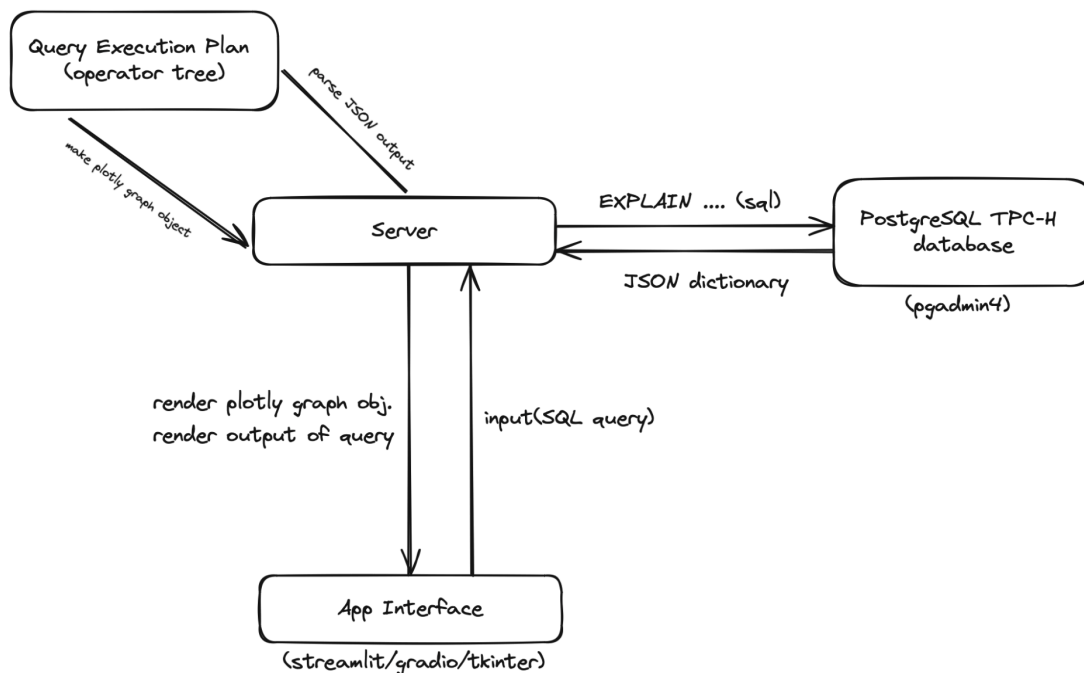
To design and implement an algorithm that explains the computation of the estimated cost associated with a QEP for a given input SQL query with the knowledge provided regarding various cost estimations.

3.1 Dataset for evaluation

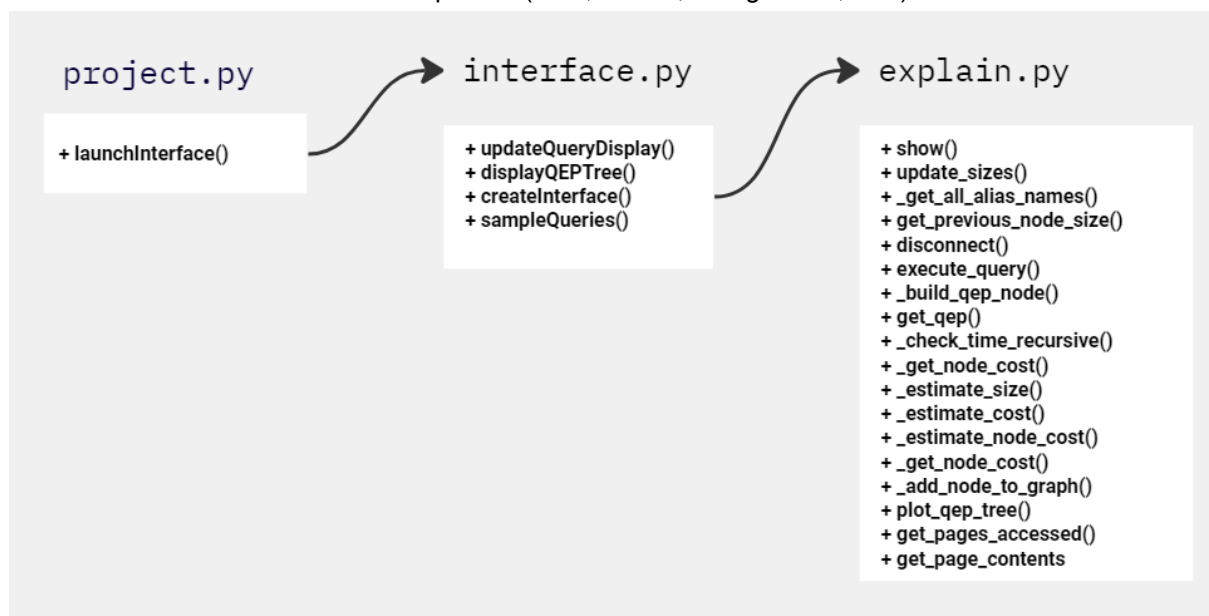
The dataset being used in these projects consist of a table that includes: customers, nation, orders, parts, partsupp, region and supplier.

3.2 Software Setup

3.2.1 System Architecture and Design



This general flow showcases how a single query is being processed in our system architecture and handshaked between individual component (QEP, Server, PostgreSQL, GUI).



The software consists of three main components: `explain.py`, `interface.py` and `project.py`. `explain.py` consists of processing, calculation and providing an explanation of data. `interface.py` is responsible for the GUI of the software, which helps manage user inputs and display outputs. Lastly, `project.py` is the central module that ties the functions of the other components together.

3.2.2 Technologies Used

Streamlining with the project's guidelines and objectives, we have decided to use a full Python-based framework. Below is a summarised list of carefully selected Python libraries used in developing our project.

Client: Gradio, Plotly

Gradio on Python supports the creation of web-based interfaces for our requirements. Plotly on Python is a versatile graphing library that enables the construction of interactive visualizations in an intuitive format

Server: Psycopg, NumPy, Pydantic

Psycopg works as a PostgreSQL adapter for python, allowing an ease of use interface for database operations and is known for its compliance with several python database APIs. Numpy was used in our project as well as it provides several mathematical functions to operate on our arrays. Finally, Pydantic was used mainly for its data validation which greatly reduces bugs and our code clarity.

3.3 Understanding cost estimation by PostgreSQL EXPLAIN

The PostgreSQL EXPLAIN command is a crucial tool for analyzing and optimizing query performance within relational database management systems. It provides a detailed breakdown of the Query Execution Plan (QEP), offering insights into the operational steps, including join types, scan methods, and estimated costs associated with executing each component of the SQL query. This information enables developers and database administrators to understand and refine the efficiency of query processing.

An example of the prowess of this command is shown below:

```
EXPLAIN SELECT * FROM foo;
```

QUERY PLAN

```
-----  
Seq Scan on foo (cost=0.00..155.00 rows=10000 width=4)  
(1 row)
```

The EXPLAIN command comes with the option of passing additional parameters to receive more and more information (especially related to the cost calculation, buffer usage, memory manipulation, and intermediate relation sizes). These are particularly helpful as this study aims to demystify the mathematical formulation behind the cost calculation of the optimizer behind PostgreSQL.

3.3.1 Commonly used notations, abbreviations

For the scope of discussion of estimated costs for different components of a Query Execution Platform, we provide the list of PostgreSQL settings used as default for our calculations and observations.

These constants are explained in the official mirror Github repository of PostgreSQL. They are used widely for the estimation and projection of expected output sizes for intermediate relations as well as the cost-based analysis.

Constant	Value	Description	Abbreviation
DEFAULT_CPU_TUPLE_COST	0.01	Cost of processing one tuple	CPU_{tuple}
DEFAULT_CPU_INDEX_TUPLE_COST	0.005	Cost of processing one index tuple	$CPU_{index\ tuple}$
DEFAULT_CPU_OPERATOR_COST	0.0025	Cost of executing one operator or function	CPU_{op}
DEFAULT_PARALLEL_TUPLE_COST	0.1	Cost of passing one tuple from worker to leader	$CPU_{parallel\ tuple}$
DEFAULT_PARALLEL_SETUP_COST	1000	Cost of setting up shared memory for parallelism	$CPU_{parallel\ setup}$
DEFAULT_SEQ_PAGE_COST	1	Cost of fetching a sequential page from	SPC
DEFAULT_RANDOM_PAGE_COST	4	Cost of fetching a non-sequential/random page from disk	RPC

For each operator, as you will see later in the upcoming subsections, the approximate mathematical formulation can be described generally as follows:

$$Cost = (\sum_i Cost_i) . MF$$

We define it as a multiplicative factor, which is derived after various trials and errors to generalise. This does not represent nor guarantee any exact solution but rather gives accountability to various cost overheads not considered, such as parallelism, hash costing, caching etc. This factor qualitatively (and quantitatively too, to some extent) represents these minute overheads.

3.3.2 Query Operators

For this project's scope, we mainly cover the query operators covered in the academic curriculum and standard SQL clauses used on a day-to-day basis.

3.3.2.1 Sequential Scan

A sequential scan represents the most rudimentary method for data retrieval from a table, systematically progressing through each data page in sequence. Similar to other scanning techniques, it permits the application of filters during data acquisition; however, it requires initial data reading followed by potential discarding. This approach lacks the capability to selectively target specific data segments, necessitating the reading of the entire table content. Typically, this method is less efficient unless a significant portion of the table data is required to fulfil the query. Nonetheless, it remains a consistently available option and may occasionally represent the sole viable method.

$$Cost_{Sequential Scan} = (Cost_{disk} + Cost_{CPU} + Cost_{Filter}) \cdot MF$$

where

- $Cost_{disk} = SPC \cdot N_{pages}$ is the cost inferred by disk utility and I/O movements,
- $Cost_{CPU} = CPU_{tuple} \cdot N_{tuples}$ is the cost by cpu running costs
- $Cost_{Filter} = CPU_{op} \cdot N_{tuples} \cdot N_{filters}$ is the cost incurred if there are filtering conditions associated with the scan.
- $MF = 0.61$ is the multiplicative factor.

3.3.2.2 Index Scan

An index scan utilizes an index to locate specific rows or all rows that satisfy a given predicate. This method can either target individual rows sequentially or as part of the inner table in a nested loop, correlating to the current row from the outer table. Alternatively, it can navigate through a contiguous segment of the table in a structured order. Initially, an index scan retrieves each row from the index, subsequently verifying the actual table data for the corresponding index entry. This verification is necessary to confirm the visibility of the row to the ongoing transaction and to acquire any query-relevant columns not included in the index. Despite its higher per-row operational overhead compared to a sequential scan, the principal advantage of an index scan lies in its ability to limit the reading to specific table rows. However, if the query predicate filters out only a few rows, a sequential scan may prove more efficient than an index scan.

$$Cost_{Sequential Scan} = (Cost_{disk} + Cost_{CPU} + Cost_{Filter}) \cdot MF$$

where

- $Cost_{disk} = SPC \cdot N_{pages}$ is the cost inferred by disk utility and I/O movements,
- $Cost_{CPU} = CPU_{tuple} \cdot N_{tuples}$ is the cost by
- $Cost_{Filter} = CPU_{op} \cdot N_{tuples} \cdot N_{filters}$ is the cost by
- $DF = CPU_{op} \cdot N_{tuples} \cdot N_{filters}$ is the cost by

3.3.2.3 Index-Only Scan

An index-only scan is closely related to a standard index scan but is distinguished by its ability to fetch data exclusively from the index, bypassing the table data entirely. This process is facilitated by a specialized visibility check, enhancing its speed relative to traditional index scans. However, the availability of an index-only scan as a substitute for a regular index scan is conditional. It requires the index type to support this functionality, with common B+ tree indexes typically fulfilling this criterion. Additionally, the query must be structured to project only those columns that are contained within the index. For instance, even if a query initially uses `SELECT *` but does not require all columns, modifying the query to select only indexed columns could enable the use of an index-only scan, thereby optimizing query performance.

$$Cost_{Sequential\ Scan} = (Cost_{disk} + Cost_{CPU} + Cost_{Filter}) \cdot MF$$

where

- $Cost_{disk} = SPC \cdot N_{pages}$ is the cost inferred by disk utility and I/O movements,
- $Cost_{CPU} = CPU_{tuple} \cdot N_{tuples}$ is the cost by
- $Cost_{Filter} = CPU_{op} \cdot N_{tuples} \cdot N_{filters}$ is the cost by
- $DF = CPU_{op} \cdot N_{tuples} \cdot N_{filters}$ is the cost by

3.3.2.4 Bitmap Heap Scan (Multi-Index Scan)

A bitmap heap scan employs a bitmap, generated via a Bitmap Index Scan or a combination of bitmap set operations (BitmapAnd and BitmapOr nodes), to determine relevant data locations. This bitmap categorizes data chunks as either exact—directly pinpointing specific rows—or lossy—indicating a page that contains at least one row meeting the query criteria.

$$Cost_{Bitmap\ Heap\ Scan} = (RunCost_{Page} + RunCost_{Output} + StartupCost) \cdot MF$$

where

- $RunCost_{Page} = SPC \cdot N_{pages}$ SPC is Single Page Cost; we used random pages for this
- $RunCost_{Output} = CPU_{tuple} \cdot N_{tuples}$ is the cost of the CPU outputting the output tuples
- $StartupCost = CPU_{op} \cdot N_{tuples} + CostStartup$ is the cost required to get the query and start running it.
- MF is a multiplying factor that helps keep the values proportional

One important thing to note is that while the formula was taken from the implementation of PostgreSQL's open-source repository, some variables took into account caching buffers across different nodes, which were infeasible to track. To make up for the exclusion of these variables, we added a multiplying factor to ensure the values were proportional. PostgreSQL prioritizes the use of exact blocks for data retrieval. The designation of blocks as lossy or exact is determined by the operations preceding the bitmap heap scan. This distinction becomes critical during the processing phase, particularly if a block is lossy, requiring the node to fetch the entire page and re-evaluate the index condition to accurately identify the necessary rows. This additional step is needed because the exact rows satisfying the condition on the page are not known in advance.

3.3.2.5 Bitmap Index Scan (Multi-Index Scan)

A bitmap index scan represents an intermediary approach between a sequential scan and an index scan. It combines the precision of an index scan—scanning an index to determine specific data requirements—with the efficiency of a sequential scan, which capitalizes on the ease of reading data in bulk. The bitmap index scan functions collaboratively with a Bitmap Heap Scan; it does not retrieve data by itself. Instead, it generates a bitmap representing potential row locations, which is then utilized by a Bitmap Heap Scan. The latter scan interprets this bitmap to access and retrieve data from the pages indicated sequentially.

$$Cost_{Bitmap\ Index\ Scan} = (Path_{Index} + Bitmap_{Manipulation} + Path_{TotalCost}) \cdot MF$$

where

- $Path_{Index}$ is the cost required to traverse the path to the index
- $Bitmap_{Manipulation} = 0.1 * CPU_{Operator} \cdot N_{tuples}$ is the cost to manipulate and maintain the bitmap. 0.1 is chosen as a way to demonstrate a small fee
- $Path_{Total} = CPU_{Tuples} \cdot N_{tuples}$ is the cost for traversing the all the paths to the bitmaps

Commonly, a Bitmap Heap Scan serves as the primary consumer of the bitmap produced by a Bitmap Index Scan. However, complex query plans may integrate multiple Bitmap Index Scans through BitmapAnd or BitmapOr nodes before the actual data retrieval occurs. This strategy allows PostgreSQL to leverage multiple indexes simultaneously to optimise query execution, showcasing a flexible, layered approach to data access.

3.3.2.6 Aggregate

An Aggregate node in PostgreSQL facilitates both plain and grouped aggregation. For grouped aggregation, PostgreSQL can efficiently process either presorted or unsorted input. When dealing with unsorted input, the system employs an internal hashtable to manage and aggregate data based on the grouping criteria specified in the query. This allows for dynamic aggregation during query execution, adapting to the nature of the input data to optimize performance.

$$Cost_{Aggregate} = (N_{Child\ Rows} * CPU_{Operator} + N_{Current\ Rows} * CPU_{Tuple}) \cdot MF$$

where

- $CPU_{Operator}$ is the cost inferred by CPU for performing any basic operation on the data
- CPU_{Tuple} is the cost inferred by going through all the tuples
- $MF = 2$ is the multiplicative factor

3.3.2.7 Sort

A Sort node in PostgreSQL processes data received from a child node to generate a sorted output. It accomplishes this by utilizing available memory, as configured under the `work_mem` setting, or by spilling excess data to disk when memory limits are exceeded. This sorting operation is essential when the output explicitly requires sorting. However, in some scenarios, the input might already be in a sorted format, such as when scanning a Btree index that aligns with the desired sort order. In such cases, the Sort node can leverage the pre-sorted nature of the input to optimize the sorting process, reducing resource utilization and improving query performance.

$$Cost_{Sort} = (Cost_{run} + Cost_{startup}) \cdot MF$$

where

- $Cost_{run} = CPU_{op} \cdot N_{tuples}$ is the cost inferred by disk utility and I/O movements,
- $Cost_{startup} = N_{pages\ accessed} \cdot (SPC \cdot 0.75 + RPC \cdot 0.25)$ this part of $Cost_{startup}$ indicates the cost of retrieving the blocks of data
- $Cost_{startup} = 2 \cdot CPU_{op} \cdot N_{tuples} \cdot \log_2(N_{tuples}) + \delta$ this indicates the main sorting algorithm that runs on $O(N \log N)$ complexity, and for each operation, there is a cpu compare cost
- $MF = 1$ is the multiplicative factor

3.3.2.8 Hash

A Hash node in PostgreSQL functions by reading data into a hash table, where each entry can be swiftly accessed using a hash key. This mechanism is primarily employed in operations like hash joins and hash aggregates. The hash table organizes the data in such a way that it allows for efficient lookups, comparisons, and retrievals by hashing the values of the join or aggregation keys, thus optimizing the overall execution of queries that involve matching or grouping large sets of data based on these keys.

$$Cost_{Hash} = 0$$

3.3.2.9 Gather

A Gather node in PostgreSQL is used to aggregate data from multiple parallel workers. This node functions similarly to the Append node in that it consolidates results, but it specifically collects outputs from various worker processes executing parts of a query in parallel. The Gather node enables efficient parallel processing by distributing the workload across multiple workers and then collating their results into a single output stream for further processing or final output. This mechanism is pivotal in leveraging the advantages of parallel execution within a PostgreSQL environment, enhancing query performance by reducing execution time.

$$Cost_{Total} = (Cost_{Parallel\ Setup} + Cost_{Parallel\ Tuple}) \cdot N_{Tuples}$$

where

- $Cost_{Parallel\ Setup}$ is the cost inferred by setting up parallel operations and workers,
- $Cost_{Parallel\ Tuple}$ is the cost inferred by running the query in parallel
- N_{Tuples} is the number of tuples that are sent to the node from the (intermediate) relation

3.3.2.10 Gather Merge

A Gather Merge node in PostgreSQL is designed to merge the results of pre-sorted outputs from multiple workers, functioning in a manner similar to the Merge Append node. This specialised node efficiently combines sorted data streams from parallel query executions, ensuring that the merged result is also sorted according to the same criteria. By leveraging pre-sorted inputs from each worker, the Gather Merge node minimises the computational overhead typically associated with sorting during the merge process, thereby enhancing the overall efficiency of executing complex queries in a distributed environment.

$$Cost_{Total} = (Cost_{Run} + Cost_{Startup}) \cdot MF$$

where

- $Cost_{Run} = CPU_{parallel\ tuple} * N_{Tuples} * 1.05 + N_{Tuples} * \log_2(N_{Worker} + 1) * CPU_{Comparison}$ is the cost inferred by running the query. The 1.05 is a postgres given value that aims to account for when workers may be locked out of memory.
- $Cost_{Startup} = Cost_{ParallelSetup} + CPU_{Comparison} * N_{Workers+1} * \log_2(N_{workers+1})$ is the cost inferred by preparing for the query ready to run
- $MF = 1$ is the multiplicative factor

3.3.2.11 Limit

A Limit node in PostgreSQL handles data received from a child node and controls the output based on specified limit conditions. It manages the volume of data processed using available memory, governed by the 'work_mem' setting, or by spilling data to disk when the memory capacity is exceeded. This node is crucial when only a specified subset of the result set is required, such as the top N entries. Although primarily focused on restricting the size of the output, in scenarios where the input is already sorted—such as scanning a B+ tree index that aligns with the desired sort order—the Limit node can efficiently truncate the sorted data to meet query conditions without the need for additional sorting

$$Cost_{Limit} = (N_{Limit} / N_{TotalRows}) \cdot Cost_{Child}$$

where

- N_{Limit} is the numerical limit provided in the query
- $N_{TotalRows}$ is the total number of rows before applying the limit
- $Cost_{Child}$ is the cost of the child Node

3.3.2.12 Hash Join

In a hash join operation within PostgreSQL, the system first constructs a hash table from the inner table, using the join key as the index. This hash table serves as a rapid lookup structure to match rows when scanning the outer table. As the outer table is processed, the system checks for corresponding entries in the hash table based on the join key.

If the size of the hash table exceeds the allocated `work_mem`, PostgreSQL cannot maintain the entire hash table in memory. Consequently, the operation must be partitioned into multiple batches. Each batch involves creating temporary files on disk to store parts of the hash table. This multi-batch process significantly increases the complexity and reduces the speed of the join operation, as disk access is substantially slower than memory access. Managing these conditions effectively is crucial for maintaining performance in environments with limited memory resources.

$$Cost_{Hash\ Join} = (Cost_{Startup} + Cost_{run}) \cdot MF$$

where

- $Cost_{Startup} = StartCost_{OuterPath} + StartCost_{InnerPath} + (CPU_{Operator} * N_{hashClauses} + CPU_{Tuple}) * N_{InnerTuples}$
is the cost inferred getting the query ready and loading it
- $Cost_{run} = TotalCost_{OuterPath} - StartCost_{OuterPath} + CPU_{Operator} * N_{hashClauses} * N_{OuterTuples}$
is the cost inferred by running the query through disk I/O
- $MF = 2$ is the multiplicative factor

3.3.2.13 Nested Loop Join

In a Nested Loop Join within PostgreSQL, the process involves iterating over each row in the outer table and comparing it against every row in the inner table to check for matches based on the join condition. This method is generally considered inefficient due to its exhaustive nature, as it requires a complete scan of the inner table for each row of the outer table, leading to a potentially high number of comparisons.

$$Cost_{Aggregate} = (Rows_{Inner} * Rows_{outer} + Min(Rows_{Inner}, Rows_{Outer})) * CPU_{Tuple} \cdot MF$$

where

- $Cost_{Tuple}$ is the cost of going through all the tuples
- $Rows_{Inner}$ is the number of rows in the inner relation
- $Rows_{Outer}$ is the number of rows in the outer relation
- $MF = 2$ is the multiplicative factor

However, the performance of a Nested Loop Join can be enhanced if the inner relation can be scanned using an index that aligns with the join condition. This optimization reduces the number of rows scanned in the inner table, thereby improving the efficiency of the join.

Despite its inherent inefficiencies, the Nested Loop Join is always available as a joining method in PostgreSQL and is sometimes the only feasible option, especially in scenarios where other more efficient join types, like hash joins or merge joins, cannot be utilized due to constraints on the data or the query conditions.

3.3.2.14 Merge Join

A Merge Join node in PostgreSQL efficiently combines data from two child nodes that are already sorted by a common join key. This method entails a single scan through each input relation, leveraging the pre-sorted nature of the data to facilitate the join process. However, the prerequisite for this operation is that both inputs must already be sorted according to the join key prior to the merge join. Alternatively, if the inputs are sourced through methods that inherently produce sorted data, such as an index scan aligned with the required sort order, this condition is naturally satisfied. This setup allows the Merge Join node to execute the join operation with enhanced efficiency, minimizing the need for additional sorting and thus optimizing performance.

$$Cost_{Aggregate} = (Cost_{process} + Cost_{return}) \cdot MF$$

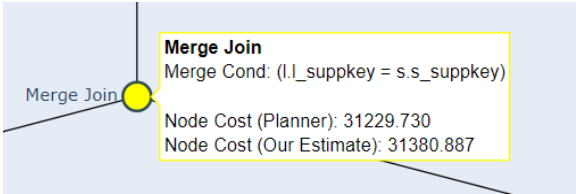
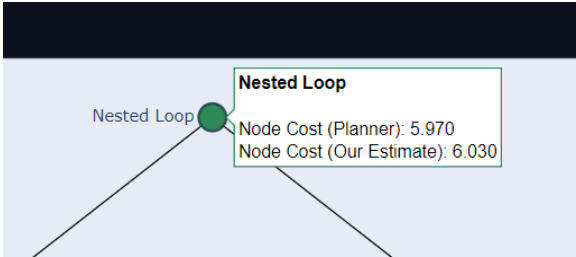
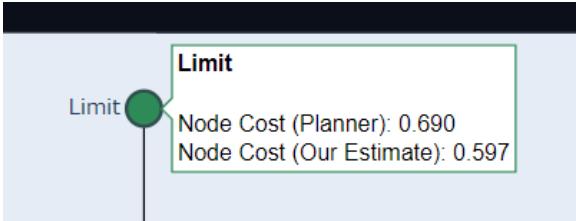
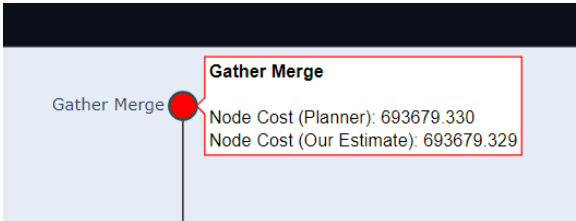
where

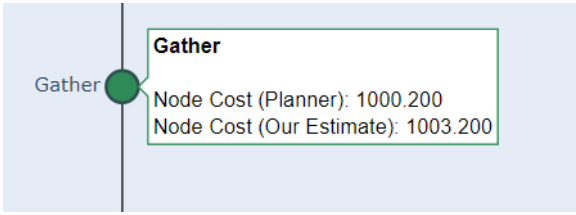
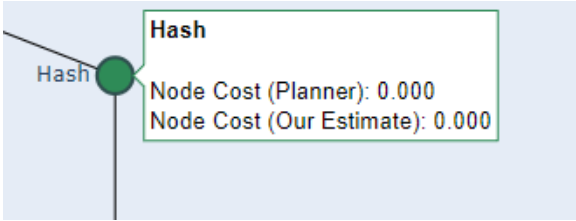
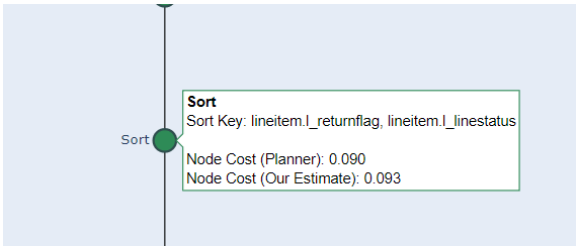
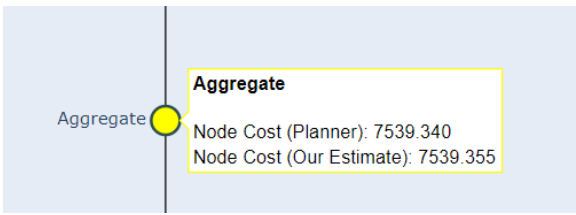
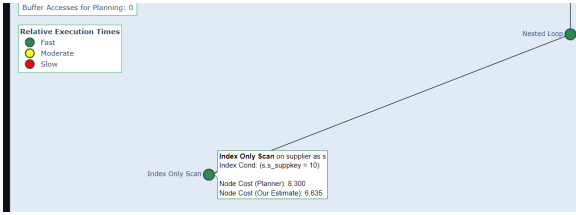
- $Cost_{process} = CPU_{op} \cdot N_{rows\ processed}$ is the cost to use the compare operator on all the tuples in both relations,
- $Cost_{return} = CPU_{tuple} \cdot N_{rows\ returned}$ is the cost of running through all the tuples in both relations
- $MF = 2$ is the multiplicative factor

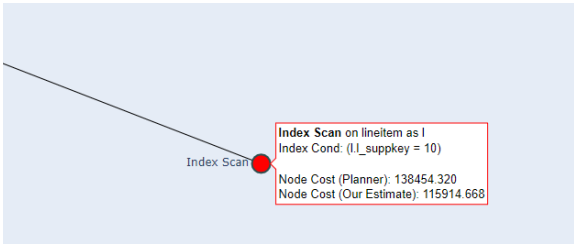
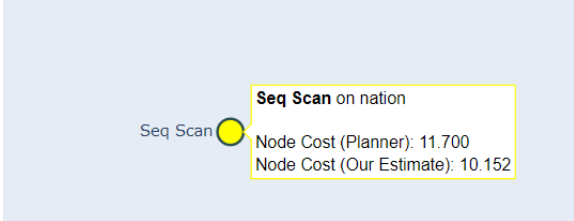
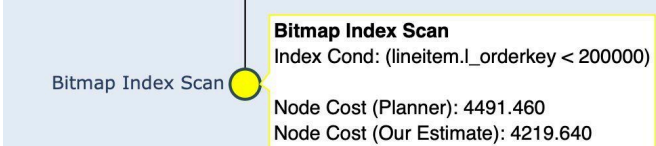
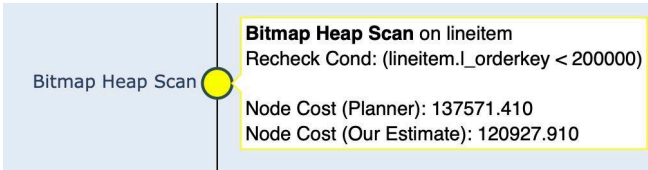
4. Discussion

4.1 Results

We implemented various cost estimation techniques for different SQL queries by referring to PostgreSQL's github. While some of our calculations were off due to the fact that we did not have access to all the buffers and cache that they did, our calculations were quite close to the predictions by PostgreSQL. We ran a variety of queries to elicit different QEPs from Postgre's optimizer. Below, we have attached examples of the queries we ran and their respective outputs with costs.

<pre>select * from lineitem l inner join supplier s on l.l_suppkey = s.s_suppkey order by l.l_suppkey ;</pre>	 <p>The diagram shows a query execution plan for a Merge Join. A yellow circle labeled 'Merge Join' is connected to a text box containing the following information:</p> <ul style="list-style-type: none">Merge JoinMerge Cond: (l.l_suppkey = s.s_suppkey)Node Cost (Planner): 31229.730Node Cost (Our Estimate): 31380.887
<pre>select * from lineitem l inner join supplier s on l.l_suppkey = s.s_suppkey where l.l_suppkey = 10;</pre>	 <p>The diagram shows a query execution plan for a Nested Loop join. A green circle labeled 'Nested Loop' is connected to a text box containing the following information:</p> <ul style="list-style-type: none">Nested LoopNode Cost (Planner): 5.970Node Cost (Our Estimate): 6.030
<pre>select * from nation LIMIT 10;</pre>	 <p>The diagram shows a query execution plan for a Limit operation. A green circle labeled 'Limit' is connected to a text box containing the following information:</p> <ul style="list-style-type: none">LimitNode Cost (Planner): 0.690Node Cost (Our Estimate): 0.597
<pre>select * from lineitem l inner join supplier s on l.l_suppkey = s.s_suppkey order by l.l_suppkey ;</pre>	 <p>The diagram shows a query execution plan for a Gather Merge operation. A red circle labeled 'Gather Merge' is connected to a text box containing the following information:</p> <ul style="list-style-type: none">Gather MergeNode Cost (Planner): 693679.330Node Cost (Our Estimate): 693679.329

<pre>select sum(l_tax * l_quantity) as totaltax from lineitem where l_quantity > 20;</pre>	 <p>Gather Node Cost (Planner): 1000.200 Node Cost (Our Estimate): 1003.200</p>
<pre>select * from lineitem l inner join supplier s on l.l_suppkey = s.s_suppkey where l.l_suppkey > 10;</pre>	 <p>Hash Node Cost (Planner): 0.000 Node Cost (Our Estimate): 0.000</p>
<pre>select * from lineitem l inner join supplier s on l.l_suppkey = s.s_suppkey order by l.l_suppkey ;</pre>	 <p>Sort Sort Key: lineitem.l_returnflag, lineitem.l_linestatus Node Cost (Planner): 0.090 Node Cost (Our Estimate): 0.093</p>
<pre>select sum(l_tax * l_quantity) as totaltax from lineitem where l_quantity > 20;</pre>	 <p>Aggregate Node Cost (Planner): 7539.340 Node Cost (Our Estimate): 7539.355</p>
<pre>select avg(l.l_quantity) from lineitem l inner join supplier s on l.l_suppkey = s.s_suppkey where l.l_suppkey=10;</pre>	 <p>Buffer Accesses for Planning: 0</p> <p>Relative Execution Times ● Fast ● Moderate ● Slow</p> <p>Index Only Scan Index Only Scan on supplier as s Index Cond: (s.s_suppkey = 10) Node Cost (Planner): 8.300 Node Cost (Our Estimate): 8.635</p> <p>Nested Loop</p>

<pre>select avg(l.l_quantity) from lineitem l inner join supplier s on l.l_suppkey = s.s_suppkey where l.l_suppkey=10;</pre>	
<pre>select * from nation;</pre>	
<pre>select * from lineitem where l_orderkey < 200000</pre>	
<pre>select * from lineitem where l_orderkey < 200000</pre>	

As it can be noted from the results above, our calculations were quite precise since we were following the original optimizers calculations as close as possible. Sometimes, the difference in cost would be significant, the reason for which, will be discussed below.

4.2 Limitations

Not all clauses for PostgreSQL are covered in our software; however, we have included all the clauses in the academic curriculum.

When there is a large query that requires a big QEP to calculate for cost estimation, it will tend to get slightly inaccurate results due to the cache overheads and buffer overheads.

We opt to build our GUI with Gradio as it provides a simple implementation to our environment, but it also lacks several useful functions and customizability found in conventional HTML.

From what we have experimented with so far, our project tool can only handle QEPs which can be visualized as a binary tree which was a common limitation found in other open sourced tools such as LANTERN.

5. Conclusion

The project has shown us the critical aspect of how a Query Execution Plan Optimization is critical in database management as it ensures queries are executed most efficiently. By optimizing it, it provides significant improvement in performance, which then leads to faster query response time. Furthermore, it can be seen that our cost estimation is within range of what the **EXPLAIN** and **ANALYZE** function has provided from PostgreSQL.

7. Appendix

A: Worked Examples

```
select supp_nation, cust_nation, l_year, sum(volume) as revenue from (
select  nl.n_name as supp_nation,n2.n_name as cust_nation,
DATE_PART('YEAR',l_shipdate) as l_year, l_extendedprice * (1 - l_discount)
as volume from supplier, lineitem, orders, customer, nation n1, nation n2
where s_suppkey = l_suppkey  and o_orderkey = l_orderkey and c_custkey =
o_custkey
and s_nationkey = n1.n_nationkey and
c_nationkey = n2.n_nationkey and ((n1.n_name = 'FRANCE' and n2.n_name =
'GERMANY') or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE') ) and
l_shipdate between '1995-01-01' and '1996-12-31' and o_totalprice > 100
and c_acctbal > 10 ) as shipping group by supp_nation, cust_nation, l_year
order by supp_nation, cust_nation,l_year;
```

```
select * from lineitem where l_orderkey < 200000
```

```
select avg(l_l_quantity) from lineitem l inner join supplier s on
l.l_suppkey = s.s_suppkey where l.l_suppkey=10;
```

```
select * from nation;
```

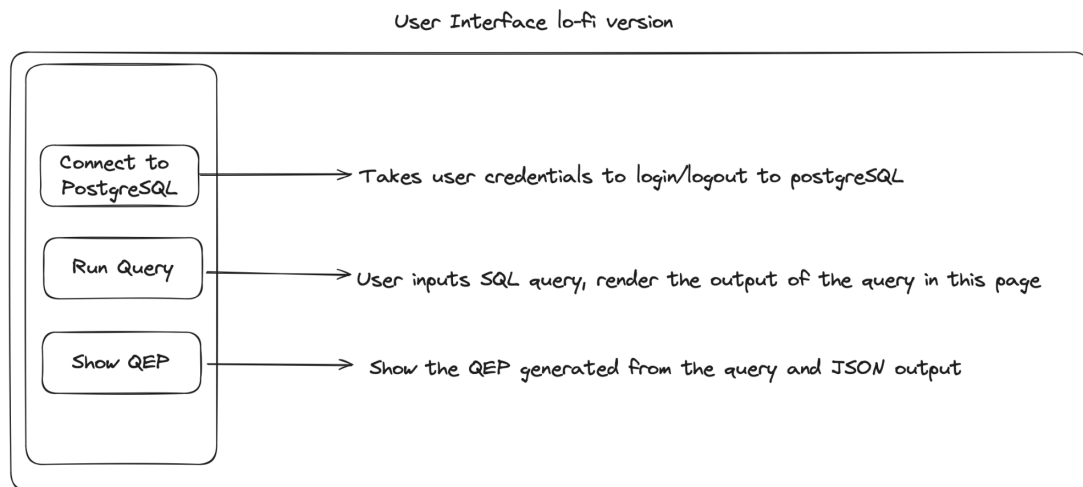
```
select c_mktsegment, o_orderpriority, count(DISTINCT o_orderkey) as
num_orders, sum(l_extendedprice * (1 - l_discount)) as revenue,
avg(l_quantity) as avg_quantity, avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_discount from customer c join
orders
o ON c_custkey = o_custkey join lineitem l ON o_orderkey = l_orderkey
where
o_orderdate >= DATE '1995-01-01' AND o_orderdate <
DATE '1995-01-01' + INTERVAL '3 month'
AND l_shipdate > DATE
'1995-01-01' group by c_mktsegment, o_orderpriority order by revenue DESC,
c_mktsegment;
```

```
select * from lineitem l inner join supplier s on l.l_suppkey = s.s_suppkey
where l.l_suppkey = 10;
```

```
select l_returnflag, l_linestatus, sum(l_quantity) as total_quantity,
avg(l_extendedprice) as avg_price, avg(l_discount) as avg_discount,
count(*) as count_orders from lineitem where
l_shipdate <=
DATE '1998-12-01' - INTERVAL '90 day' group by l_returnflag, l_linestatus
order by
l_returnflag, l_linestatus;
```

```
select sum(l_tax * l_quantity) as totaltax from lineitem where l_quantity >
20;
```

B: Lo-fi mockups for user interface



C: Usage of LLMs

Our group declares that Large Language Models (LLMs) or AI tools were utilized for the primary purpose of the ideation process and information gathering during the brainstorming phase of the project, in no way the associated codebase or report was generated using any LLM. The entire development process done by us on both the report (including analysis) and code are done solely by our manual effort.