NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Nanyang Technological University

SC3020/CZ4031 Group Project 2

Understanding Cost Estimation in RDBMS

Luo Yihang: U2022279G

Mao Yiyun: U2022609J

Zhou Yuxuan: U2022079B

Zhang Danxu: U2120845K

Group 18 - April, 2024

School of Computer Science and Engineering

Acknowledgments

This SC3020/CZ4031 Database System Principle course project has been an enriching and insightful experience for our team. We would firstly like to express our appreciation to the course instructors and assistants for providing such a meaningful task for us. This project marks the end of our undergraduate journey for most members of our team, as we are nearing graduation. It has afforded us the opportunity to apply the knowledge gained from our courses and delve into various intriguing aspects of QEP cost.

Additionally, we extend our gratitude to Generative AI technology for its invaluable assistance in refining and polishing this report. The utilization of advanced generative AI tools has greatly facilitated the enhancement of language clarity, and the optimization of data presentations.

Contents

1 Introduction

This project includes the design and implementation of software aimed at estimating the computational process behind estimating costs associated with Query Execution Plans (QEPs) for given input Structured query language (SQL) queries. Our team has invested substantial time delving into database concepts. Specifically, our efforts encompass:

1. Researching the **source code** of Postgres to replicate **concise cost estimation algorithms** for multiple query operations.
2. Providing **comprehensive natural language explanations** within the report for these queries. Given the limited resources explaining exact cost estimation algorithms in Postgres, our team has conducted research on step cost estimation plans **by our own**.
3. Implementing a **Depth-First Search (DFS)-based pipeline** to extract Query Estimation Plans from Postgres.
4. Developing a **user-friendly UI** using Qt, allowing user to input a query and get a **visualization** for cost explaination of QEPs.

Through these endeavors, our software aims to investigate QEP cost estimation, offering users valuable insights into query cost calculation.

2 Project Architecture

In this section, we will explain our project's code design architecture and instruct how to start our application.

2.1 Database

For our project, we utilize the TPC-H dataset, a decision support benchmark comprising eight tables: customer, lineitem, nation, orders, part, partsupp, region, and supplier. The TPC-H dataset is initialized according to the TPCH-dbgen tool (`https://github.com/electrum/tpch-dbgen`). To facilitate the use of the database in PostgreSQL, we provide a Dockerfile to initialize a Docker image from scratch. Additionally, we run a `VACUUM ANALYZE` to clean and optimize the database, and we execute `SET max_parallel_workers_per_gather = 0` to disable parallel cost estimation, ensuring accurate cost estimations.

2.2 Backend Structure

The backend is responsible for receiving input queries and returning the explanation of the QEP for those queries. It is implemented in Python. Our backend is designed in following pipeline with the following features:

2.2.1 Preprocessing a Query from User Input

For an input query, we first establish a connection to the PostgreSQL database using the psycopg2 Python API interface. Then, we validate the SQL query and parse the QEP in JSON format using the following command:

```
EXPLAIN (ANALYZE, VERBOSE, FORMAT JSON) ${YOUR_SQL_QUERY}
```

The QEP is generated in a tree format due to its hierarchical nature. This format provides a clear representation of the sequence of operations involved in query execution. Each node in the tree represents an operation. After parsing the JSON, we convert it into operation tree in our backend. We refer to each tree node as an operation node. Subsequently, we estimate the costs for each operation node.

2.2.2 Processing Operation Nodes

To process the Operation Tree obtained from the previous step, we employ a depth-first search (DFS) based recursive approach. Here's a pseudocode representation:

```
def explain_node(self, node: dict):
    # Logic to process the node
    if "Plans" in node.keys():
        subplans: list = node["Plans"]
        for subplan in subplans:
            self.explain_node(subplan)
```

Listing 1: DFS-based Node Explanation

To estimate the cost of processing each Operation Node, we analyze the PostgreSQL source code for concise cost estimation. We explore the codebase of the relational query processor to gather information related to cost estimation. Some frequently used predefined constants are shown in Table 1.

PostgreSQL constants	**Meaning**	**Value**
seq_page_cost	Cost to access a sequential page from disk	1
random_page_cost	Cost to access a random page from disk	4
cpu_tuple_cost	Cost for CPU to process each row during a query	0.01
cpu_index_tuple_cost	Cost for CPU to process each index entry during a query	0.005
cpu_operation_cost	Cost for CPU to process each operator or function executed during a query	0.0025
heap_tuple_header_size	Size of heap tuple header in bytes	32
work_mem	Maximum amount of memory to be used by a query operation in bytes	4096
block_size	Size of block in bytes	8192

Table 1: Frequently Used PostgreSQL Constants.

While this constants is either getting during runtime or from the PostgreSQL code. The SQL command to getting the constant is:

```
SHOW ${PARAM_NAME}
```

While we also need to retrieve the page size tuples number or index tuples, index pages information from the original relations in PostgreSQL. The SQL commands to retrieve the relation information are:

```
SELECT relname, reltuples, relpages
FROM pg_class
WHERE relkind = 'r';  -- Filter to include only ordinary tables
```

and the SQL commands to retrieve the index information are:

```
SELECT relname, reltuples, relpages
FROM pg_class
WHERE relkind = 'i';  -- Filter to include only index tables
```

After gathering relevant information related to the cost, we calculate the cost for each node based on its type using specific algorithms.

2.2.3 Postprocessing Explanation for Visualization

After obtaining the cost estimation and explanation from the previous step, we reconstruct an explanation tree in the same format as the operation trees. Each node of the explanation tree contains:

1. Node Type
2. Startup Cost
    (a) Natural Language Explanation
    (b) Formula and Calculation Result
3. Runtime Cost
    (a) Natural Language Explanation
    (b) Formula and Calculation Result
4. Total Cost (sum of Startup Cost and Runtime Cost)

This structure facilitates frontend visualization.

2.3 Frontend Structure

We implemented a user-friendly frontend UI using Qt Python, featuring drag-and-drop functionality. The frontend UI is shown in Figure 1.

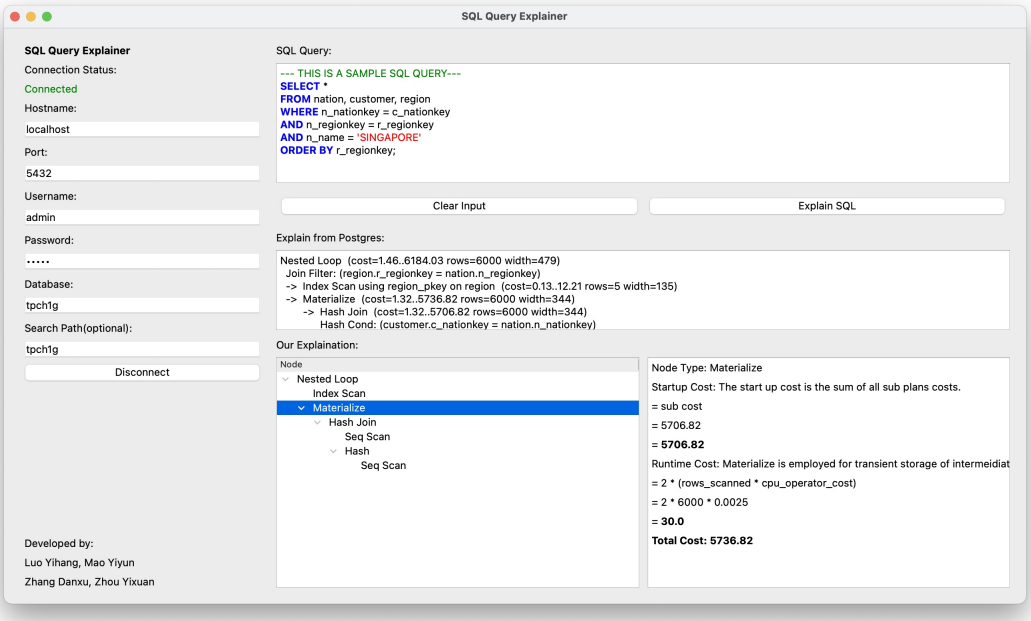

Figure 1: Front End UI Page.

2.3.1 Database Login Component

Instead of hardcoding the database connection logic, we allow users to connect to the database using our frontend UI. The UI can also display the connection status, as shown in the left box of Figure 1.

2.3.2 SQL Input Component

Users can input SQL queries in the designated input box. Additionally, we implemented SQL grammar highlighting for easier query input, as shown in the top right box of Figure 1.

2.3.3 Explanation Component

As the backend returns the explanation in a tree structure, we display the explanation of nodes in a tree-like manner. Users can freely expand or collapse explanations for each node and drag the area to view the output content, as shown in the bottom right box of Figure 1.

2.3.4 Error Handling

We have implemented robust error handling in the frontend. In case of failure to connect to the database or input of an invalid SQL query, appropriate responses are provided to the users, as shown in Figure 2 and Figure 3.

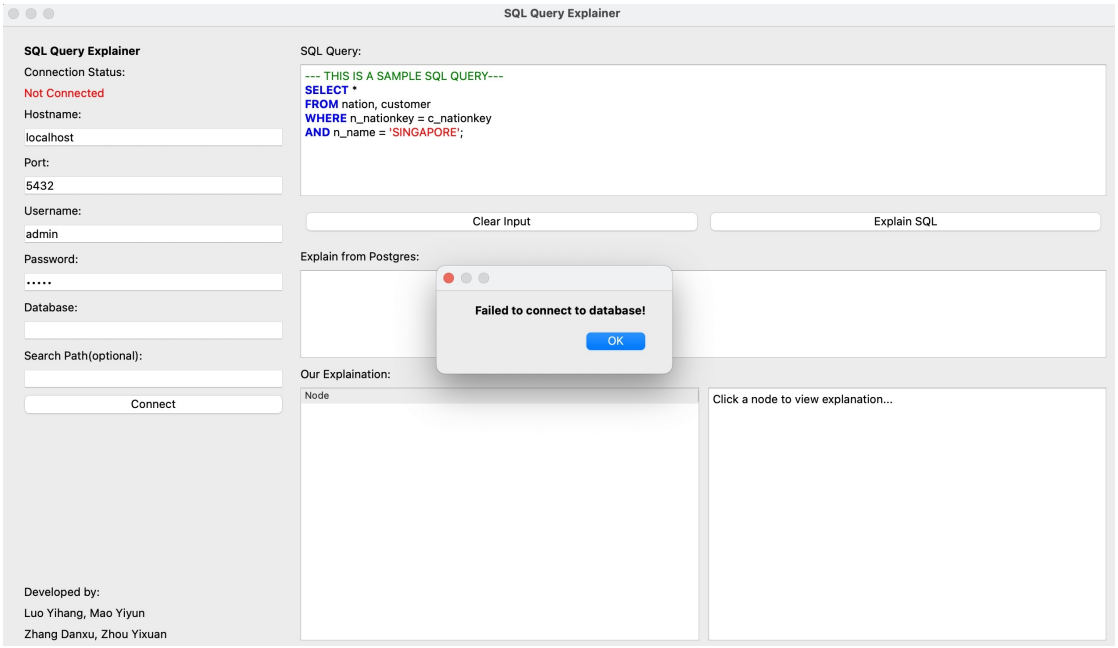

Figure 2: Front End UI Page Error Handling for Unsuccessful DB Connection.

2.4 Get Started

In this section, we will guide you on setting up and starting our software from the source code provided in our code folder.

2.4.1 Environments

Before getting started, please ensure that you have the following environment:

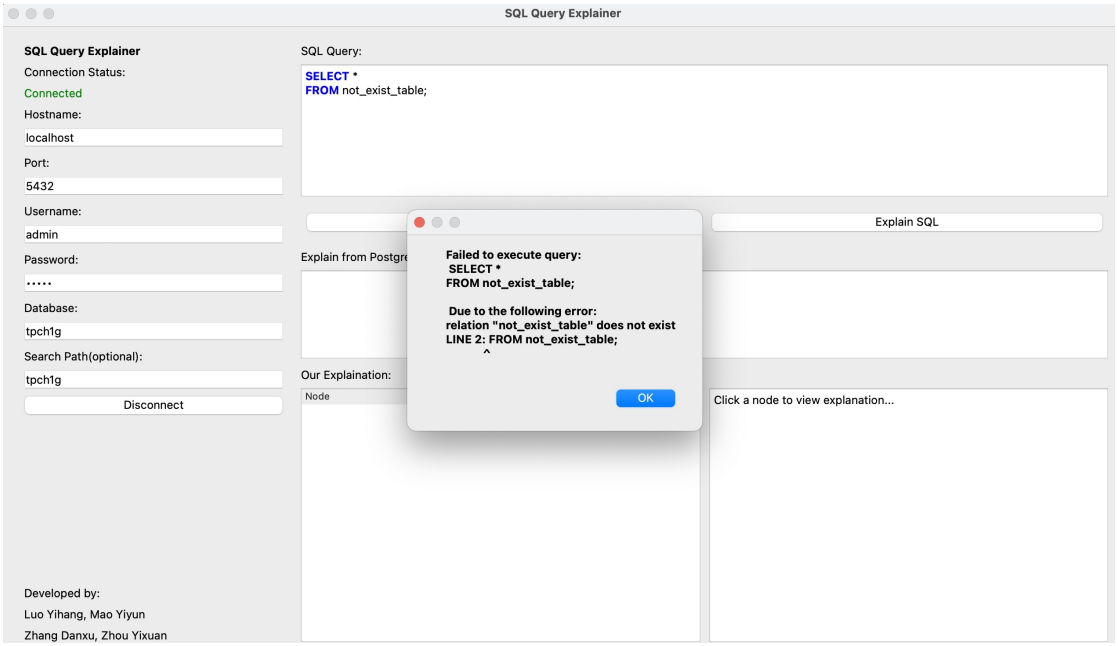

Figure 3: Front End UI Page Error Handling for Invalid SQL Query.

- Windows 10/11
  - Postgres Client 16.2
- Python $\geq$ 3.11
  - psycopg2 2.9.9
  - numpy 1.26.4
  - PyQt5 5.15.10
- Docker (Optional)

Install python environments:

```
pip install -r requirements.txt
```

2.4.2 Initialize Database

You can skip this step if you already have a PostgreSQL database with TPC-H database data. In the setup folder, follow these steps:

Build Image:

```
docker build -t tpch-db .
```

Run Container:

```
docker run -d \
  -e POSTGRES_DB=tpch1g \
  -e POSTGRES_USER=admin \
  -e POSTGRES_PASSWORD=admin \
  -v tpch_pgdata:/var/lib/postgresql/data \
  --name tpch_postgres \
  -p 5432:5432 \
  tpch-db:latest
```

Connect to DB:

```
psql -h localhost -U admin -d tpch1g
```

Run in PostgreSQL Prompt:

```
set search_path to tpch1g;
```

Explore Tables:

```
SELECT * FROM nation LIMIT 5;
```

2.4.3 Rrun Application

Our `src` folder contains three Python files: `interface.py`, `explain.py`, and `project.py`. `interface.py` contains the code for the GUI, `explain.py` contains the code for generating explanations, and `project.py` is the main file that invokes all necessary procedures.

Simply run the following command under this folder:

```
python ./src/project.py
```

This will launch our application on your screen.

Enter Database Info:

For our docker setup, the database name is `tpch1g` and search path is `tpch1g`.

You may need to change if you are using other setup.

Click `connect` button to connect, once connected, the connection status label will change to green 'Connected'.

Enter SQL Queries:

Enter your SQL you'd like to explain in the `SQL Query` text box, click `Explain SQL` button to explain.

Check Explanation from PostgreSQL:

The middle section of the right hand side is the output from PostgreSQL `EXPLAIN`.

Check Our Explanation:

The bottom section of the right hand side is our explanation. You can view the tree-structure of the QEP on the left panel, click a node and view our explanation for that node on the right panel.

3 Cost Estimation

In this section we will discuss our implementation of cost estimation. For this project, our main goal is to design and implement a software to explain the computation of the estimated cost associated with a QEP for a given input SQL query, where the QEPs and estimated costs are obtained using the `EXPLAIN` feature of PostgreSQL. We have gained valuable knowledge regarding various cost estimations in this course, where we use I/O costs and intermediate tuple numbers as cost estimations when selecting QEPs for any queries. However, when we investigated the documentations and source code of PostgreSQL, we found out PostgreSQL uses much more complicated algorithms and many planner cost constants to calculate the costs.

Also, the total cost of any node is separated into startup cost and run cost in PostgreSQL. The startup cost is the time expended before the output phase can begin, e.g., time to do the sorting in a sort node. The total cost is stated on the assumption that the plan node is run to completion, i.e., all available rows are retrieved, and the run cost is simply the difference between total cost and startup cost. It's impractical to provide credible explanations and satisfactory estimations for PostgreSQL `EXPLAIN` outputs without considering these fundamental concepts. Therefore, in this report, while acknowledging the cost estimation methods we learned in this course, we will focus more on explaining the cost calculation mechanisms we learned from investigating PostgreSQL source code, and why there are differences between our estimations and the estimations from PostgreSQL. Our estimations are simplified due to the following reasons:

1. Lack of knowledge about PostgreSQL' zonemap, histograms, and sampling, therefore, we can only rely on uniform distribution assumptions and cannot make informed guesses.

2. Some cost estimations require PostgreSQL runtime variables which are not accessible.

Even though, for many database operations, we're still able to get concise and

accurate estimations compared to the ones given by PostgreSQL' EXPLAIN query. Here, we explain the cost estimation algorithm for multiple key operations in detail.

3.1 Sequential Scan

According to PostgreSQL, sequential scan can only be applied to base relations (relations stored in database) but not to intermediate relations. Therefore, it will only appear as leaf nodes in the query execution tree and can utilize statistics of the base relations. In sequential scan, the number of pages and tuples of the base relation is used for calculating disk I/O.

According to the source code from PostgreSQL, we calculate the cost of Sequential Scan as follows:

1. Initialize *Startup Cost* and *Run Cost*. As sequential sort will always be leaf node in query execution tree, we initialize start and run cost as 0.

$$\begin{aligned} \textit{Startup Cost} &\leftarrow 0 \\ \textit{Run Cost} &\leftarrow 0 \end{aligned}$$

2. Calculate the cost of sequential page access of the base relation using the number of pages of the base relation fetched from `pg_statistic`.

$$\begin{aligned} \textit{Run Cost} &\overset{+}{\leftarrow} \texttt{seq\_page\_cost} \times \texttt{num\_pages} \\ &= \textit{Run Cost} + \texttt{num\_pages} \end{aligned}$$

3. Calculate the CPU operation cost to process all the tuples.

$$\textit{Run Cost} \xleftarrow{+} \texttt{cpu\_tuple\_cost} \times \texttt{num\_tuples}$$
$$= 0.01 \times \texttt{num\_tuples}$$

4. Calculate the CPU operation cost of processing `WHERE` clause conditions. For simplification, we assume that all `WHERE` clause conditions require `cpu_operator_cost` per tuple.

$$\textit{Run Cost} \xleftarrow{+} \texttt{cpu\_operator\_cost} \times \texttt{num\_tuples} \times \texttt{num\_conditions}$$
$$= 0.0025 \times \texttt{num\_tuples} \times \texttt{num\_conditions}$$

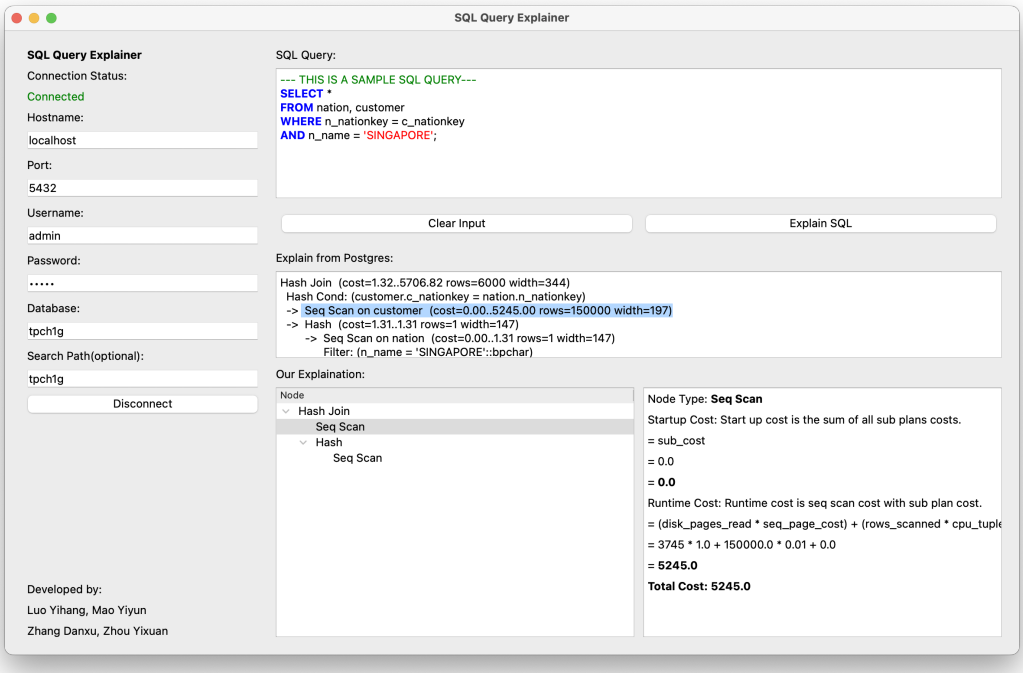

Figure 4: Example of Sequential Scan.

3.2 Sort

3 types of sort algorithms are implemented by PostgreSQL, which are External Merge Sort, Heap Sort and Quicksort. External Merge Sort is applied when the estimated output relation size is larger than the allocated memory for sorting. Heap sort is then selected when input relation size is larger than the allocated memory or number of output tuples is less than half of input tuples (usually triggered by `LIMIT` operation). Quicksort is applied in the rest of circumstances.

We implement cost estimation for all 3 sorting algorithms. As we do not know the allocated memory for sorting, we use PostgreSQL `EXPLAIN ANALYSE` function to get the sorting method that are actually selected in real world execution and calculate its corresponding estimated cost.

The overall process of estimating cost is shown below:

1. Initialize *Startup Cost* and *Run Cost* using the child node.

$$\textit{Startup Cost} \leftarrow \texttt{child\_total\_cost}$$

$$\textit{Run Cost} \leftarrow 0$$

2. Calculate the algorithm-specific cost.

$$\textit{Startup Cost} \overset{+}{\leftarrow} \textit{Sort Cost}$$

3. PostgreSQL charges a cost per tuple extracted, and the cost is set empirically as `cpu_operator_cost`. It uses number of input tuples instead of output tuples, as upper `LIMIT` node will ignore this run cost and produce correct result.

$$\begin{aligned} \textit{Run Cost} &\xleftarrow{+} \texttt{input\_tuples} \times \texttt{cpu\_operator\_cost} \\ &= 0.0025 \times \texttt{input\_tuples} \end{aligned}$$

3.2.1 External Merge Sort

In PostgreSQL, External Merge Sort can have more than 2 passes, as introduced in lectures. Although there is a clear way of calculating number of passes in the source code, we do not know the memory allocated for sorting during cost estimation. Therefore, we estimate the number of passes using the following condition:

$$\texttt{num\_passes} = \begin{cases} 1 & ,when\ 0 < n \leq 5 \times 10^5 \\ 2 & ,when\ 10^5 < n \leq 5 \times 10^6 \\ 3 & ,when\ 10^6 < n \leq 5 \times 10^8 \\ 5 & ,when\ n > 5 \times 10^8 \end{cases} \tag{1}$$

We calculate the cost of External Merge Sort as follows:

1. Estimate the number of pages of child in query execution tree using the function applied by PostgreSQL.

$$\begin{aligned} \texttt{input\_bytes} &= maxalign(\texttt{input\_tuple\_width}) + \texttt{heap\_tuple\_header\_size}) \\ &= maxalign(\texttt{input\_tuple\_width}) + 24 \\ \texttt{num\_pages} &= ceil(\texttt{num\_pages}/\texttt{block\_size}) \\ &= ceil(\texttt{num\_pages}/8192) \end{aligned}$$

2. Calculate the CPU operation cost of comparing two tuples. PostgreSQL assumes that there are a total of $N \log_2(N)$ comparisons with N input tuples, and assumes that each comparison takes $2\times$ `cpu_operator_cost`.

$$
\begin{aligned}
\textit{Merge Sort Cost} \leftarrow\ & 2 \times \texttt{cpu\_operator\_cost} \times \texttt{input\_tuples} \times \\
& \log_2(\texttt{input\_tuples}) \\
=\ & 0.005 \times \texttt{input\_tuples} \times \log_2(\texttt{input\_tuples})
\end{aligned}
$$

3. Calculate the disk access cost for disk I/O. In each pass External Merge Sort, the whole relation is written back and read in once. PostgreSQL assumes that 3/4 of the disk I/O is sequential access and 1/4 is random access.

$$
\begin{aligned}
\textit{Merge Sort Cost} \stackrel{+}{\leftarrow}\ & 2 \times \texttt{num\_pages} \times (0.75 \times \texttt{seq\_page\_cost} + \\
& 0.25 \times \texttt{random\_page\_cost}) \\
=\ & 2 \times \texttt{num\_pages} \times (0.75 \times 1 + 0.25 \times 4) \\
=\ & 3.5 \times \texttt{num\_pages}
\end{aligned}
$$

3.2.2 Heap Sort

Top-N Heap Sort is used in PostgreSQL when the estimated number of output tuples can fit into buffer memory. The cost of Heap sort is shown below:

$$
\begin{aligned}
\textit{Heap Sort Cost} \leftarrow\ & 2 \times \texttt{cpu\_operator\_cost} \times \texttt{input\_tuples} \times \\
& \log_2(2 \times \texttt{output\_tuples}) \\
=\ & 0.05 \times \texttt{input\_tuples} \times \log_2(2 \times \texttt{output\_tuples})
\end{aligned}
$$

3.2.3 QuickSort

QuickSort is used when the entire inpue tuples can fit into buffer memory. The constant factor of QuickSort is smaller than Heap Sort and is thus more efficient. The cost of QuickSort is shown below:

$$\begin{aligned} \textit{QuickSort Cost} \leftarrow\ & 2 \times \texttt{cpu\_operator\_cost} \times \texttt{input\_tuples} \times \\ & \log_2(\texttt{output\_tuples}) \\ =\ & 0.05 \times \texttt{input\_tuples} \times \log_2(\texttt{output\_tuples}) \end{aligned}$$

3.2.4 Limit

`LIMIT` node will restrict the number of tuples of SQL query output. We estimate the cost of `LIMIT` node by charging full start up cost from child, and the restricted percentage of child's run cost.

$$\begin{aligned} \textit{Startup Cost} &\leftarrow \texttt{child\_start\_cost} \\ \textit{Run Cost} &\leftarrow \texttt{child\_run\_cost} \times \frac{\texttt{output\_tuples}}{\texttt{input\_tuples}} \end{aligned}$$

3.3 Index Scan

PostgreSQL has many different implementations of indexes, each of them having their own cost estimation method. In TPC-H 1G database used for this project, all indexes of primary keys are based on B-Trees and there are no index built in non-primary keys. Therefore, for simplification purpose, we consider the cost of Index Scan on B-Trees.

In this section we will discuss the cost estimation of Index Only Scan and normal

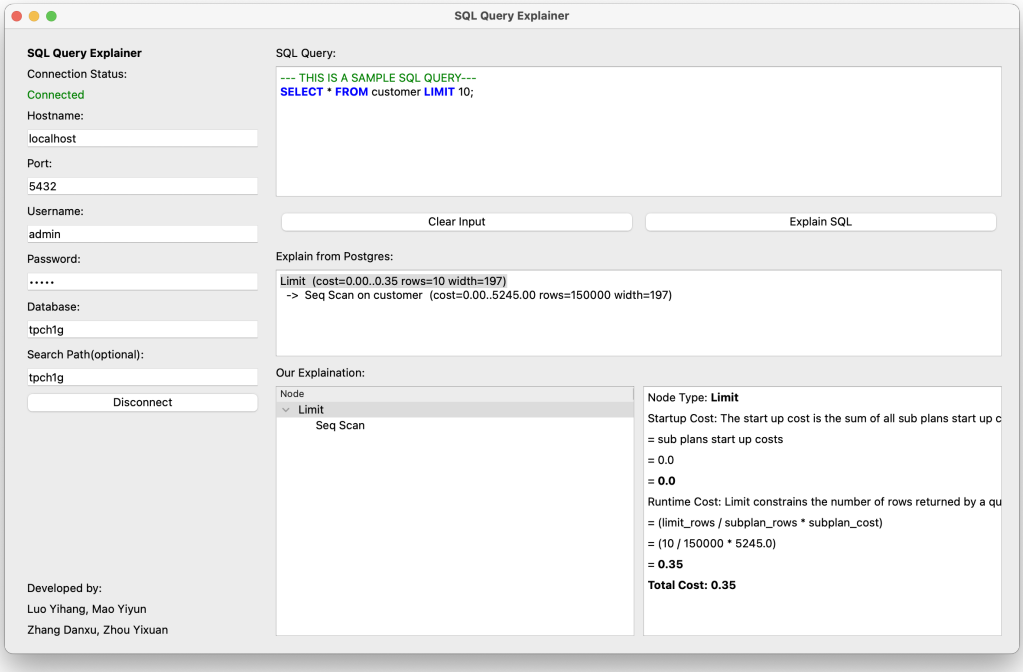

Figure 5: Example of Limit.

Index Scan. PostgreSQL uses methods like histogram and random sampling for estimating selectivity and number of output tuples. As the coarse selectivity estimation method introduced in lecture is much less precise compared to PostgreSQL's implementation, we will use output tuples provided by PostgreSQL `EXPLAIN` for cost estimation.

3.3.1 Index Only Scan

Index Only Scan is applied when all retrieved attributes are stored in Index, and no access to main relation is required. We simply the cost estimation process of PostgreSQL and the procedure is shown below:

1. Initialize *Startup Cost* and *Run Cost* using the child node.

$$\textit{Startup Cost} \leftarrow \texttt{child\_startup\_cost}$$

$$\textit{Run Cost} \leftarrow \texttt{child\_run\_cost}$$

2. Perform generic cost estimation for indexes by estimating number of disk I/Os. PostgreSQL performs selectivity estimation in this step, and assume all index pages can only be accessed randomly. We get the number of index pages and tuples in index from `pg_statistic`.

$$\begin{aligned} \texttt{idx\_pages\_sel} &\leftarrow \mathit{ceil}(\texttt{num\_tuple\_sel} \times \texttt{idx\_pages})/\texttt{idx\_tuples} \\ \mathit{Run\ Cost} &\overset{+}{\leftarrow} \texttt{idx\_pages\_sel} \times \texttt{radom\_page\_cost} \\ &= 4 \times \texttt{idx\_pages\_sel} \end{aligned}$$

3. Estimate CPU operation cost on each selected index tuples. The cost per selected tuple is `cpu_index_tuple_cost` as base cost, and one additional `cpu_per_operation` is charged per query condition.

$$\begin{aligned} \mathit{Run\ Cost} \overset{+}{\leftarrow} &\texttt{num\_tuple\_sel} \times (\texttt{cpu\_index\_tuple\_cost} + \\ &\texttt{cpu\_operator\_cost} \times \texttt{num\_conds}) \\ = &\texttt{num\_tuple\_sel} \times (0.01 + 0.0025 \times \texttt{num\_conds}) \end{aligned}$$

4. Perform index-specific cost estimation. Here we calculate the CPU operation cost to travel down B-Tree. $\log_2(\mathit{idx\_tuples})$ is expected to be compared with index condition, and `cpu_operator_cost` is charged per comparison. Moreover, PostgreSQL charges $50 \times$ `cpu_operator_cost` per B-Tree node as node processing cost. Due to lack of data (degree of B-Tree used by PostgreSQL), we are unable to directly calculate the depth of index. However, we test that index on Nation, Region and Supplier has depth of 1, and the rest of relations have depth of 2. For simplification purpose, we assume depth to be 2 for all relations.

$$\textit{Start Cost} \overset{+}{\leftarrow} \log_2(\texttt{idx\_tuples}) \times \texttt{cpu\_operator\_cost}$$

$$\textit{Start Cost} \overset{+}{\leftarrow} 50 \times (\texttt{idx\_depth} + 1) \times \texttt{cpu\_operator\_cost}$$

5. Estimate CPU operation cost on each selected tuples. The cost per selected tuple is `cpu_tuple_cost` as base cost, and one additional `cpu_per_operation` is charged per filter.

$$\begin{aligned}\textit{Run Cost} &\overset{+}{\leftarrow} \texttt{num\_tuple\_sel} \times (\texttt{cpu\_index\_tuple\_cost} + \\ &\quad \texttt{cpu\_operator\_cost} \times \texttt{num\_conds}) \\ &= \texttt{num\_tuple\_sel} \times (0.01 + 0.0025 \times \texttt{num\_filters})\end{aligned}$$

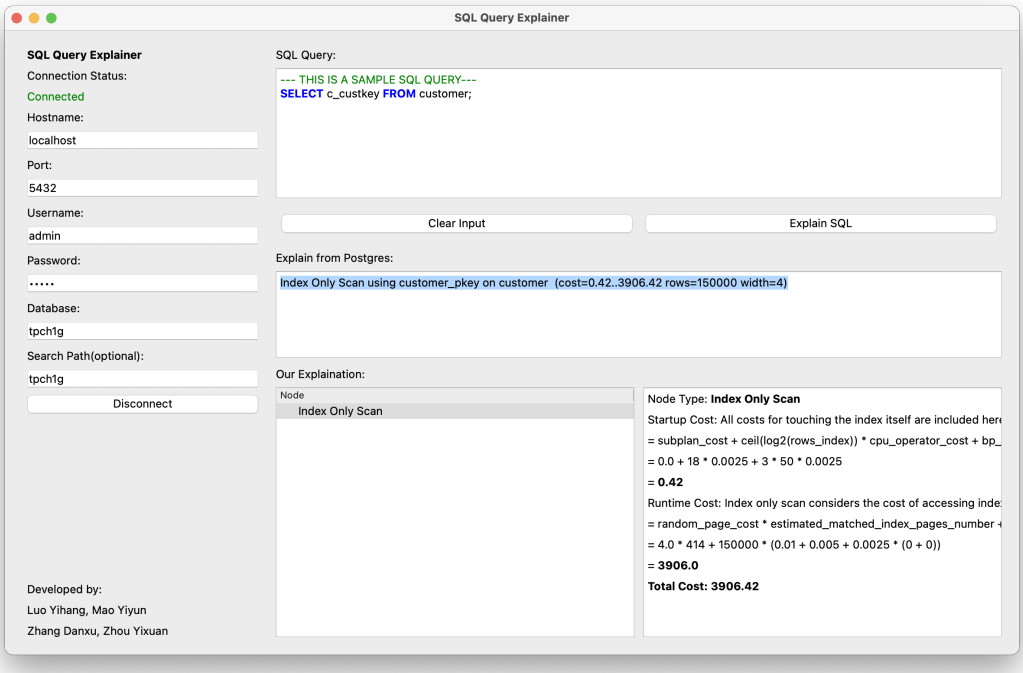

Figure 6: Example of Index Only Scan.

3.3.2 Normal Index Scan

Normal Index Scan will need to fetch pages from heap, which incurs extra costs compare to Index Only Scan. We attempt to follow the implementation of PostgreSQL, where the Mackert and Lohman formula is used for estimating number of pages fetched from heap, but it leads to huge deviation. One explanation is that Mackert and Lohman formula is a very pessimistic estimation where it assumes the attribute in the relation is not sorted, and all pages can only accessed randomly. This is apparently not true for B-Tree index built on primary key of relations in TPC-H.

Therefore, we assume that PostgreSQL has it's optimization making use of properties of B-Tree. However, we are unable to locate the related piece of code, and we will produce our own estimation method to estimate cost of fetching tables from heap. This will unavoidably lead to deviation from PostgreSQL estimation, but the idea should be similar.

The procedure of Normal Index Scan is shown below:

1. Calculate *Startup Cost* and *Run Cost* following the procedure of Index Only Scan.

2. Calculate the estimated number of pages fetched from main memory. For simplification, and also considering that the indexes are built on primary keys, we think all target pages are clustered in heap. We charge `random_page_cost` for the first page fetched, and `seq_page_cost` for the rest of the pages.

$$\begin{aligned} \texttt{pages\_fetched} &\leftarrow \mathit{ceil}(\texttt{num\_tuple\_sel} \times \texttt{rel\_pages})/\texttt{rel\_tuples} \\ \mathit{Run\ Cost} &\overset{+}{\leftarrow} \texttt{radom\_page\_cost} + \texttt{seq\_page\_cost} \times (\texttt{pages\_fetched} - 1) \\ &= \texttt{pages\_fetched} + 3 \end{aligned}$$

3. For every retrieved tuple, we charge a `cpu_operator_cost` as the CPU overhead to process each tuple.

$$\begin{aligned} \textit{Run Cost} &\xleftarrow{+} \texttt{cpu\_operator\_cost} \times \texttt{num\_tuple\_sel} \\ &= 0.0025 \times \texttt{num\_tuple\_sel} \end{aligned}$$

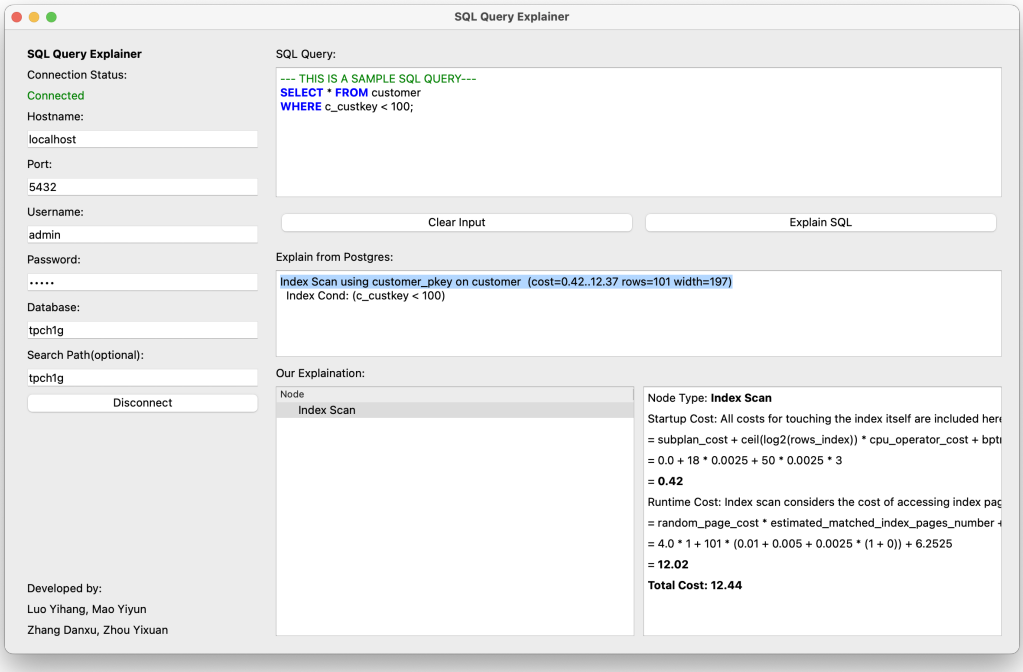

Figure 7: Example of Normal Index Scan.

3.4 Bitmap Scan

Bitmap scans make use of a bitmap to record and fetch the relation pages that fulfill the query conditions. By performing bit operation on bitmaps, we can select the target pages efficiently. In PostgreSQL, there are a total of 4 nodes that are related to Bitmap Scan, and all of them will be introduced in this section.

3.4.1 Bitmap Index Scan

Bitmap Index Scan in PostgreSQL will perform a Index Only Scan on the index. However, it does not need to save all the index tuples as intermediate. Therefore,

the estimated cost of Bitmap Index Scan is simply the cost after the first 4 steps in Index Only Scan, excluding the cost of processing selected index tuples.

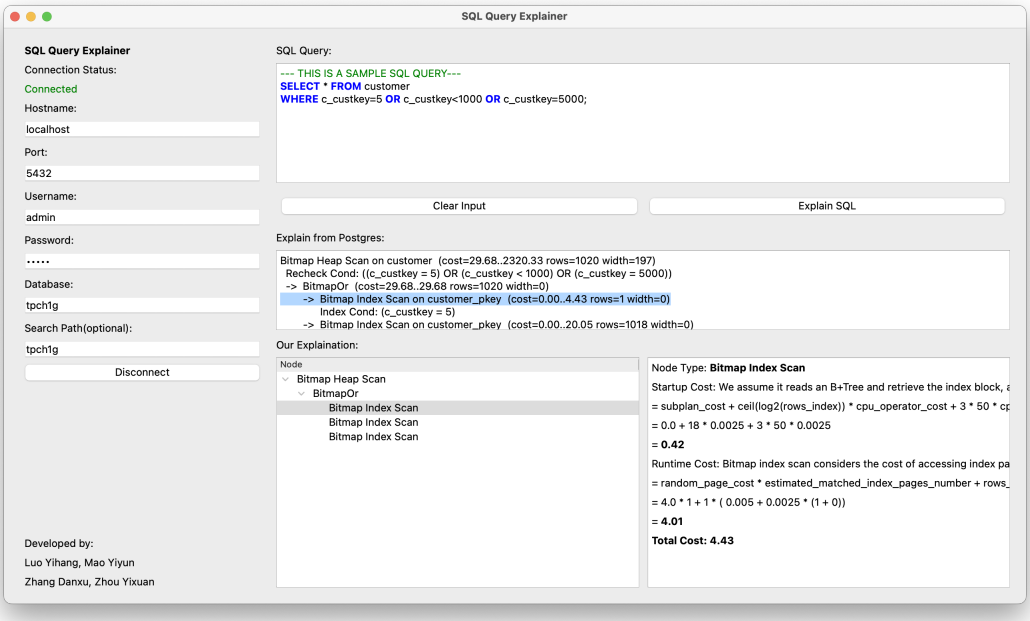

Figure 8: Example of Bit Map Index Scan.

3.4.2 Bitmap Or and Bitmap And

Bitmap Index Or and Bitmap And node performs bit operation on all their children's bitmaps. We follow the PostgreSQL's procedure of estimation, but due to the complexity of their algorithm, the cost estimation on this part will have a deviation from actual cost (less than 5% for most cases).

1. Initialize *Startup Cost* and *Run Cost* using the total costs from all child nodes.

$$\textit{Startup Cost} \leftarrow \sum \texttt{child\_total\_cost}$$

$$\textit{Run Cost} \leftarrow 0$$

2. For all Bitmap Index Scan children, charge a slightly smaller CPU operation

cost for setting up bits in bitmap.

$$\begin{aligned} \textit{Run Cost} &\xleftarrow{+} 0.1 \times \texttt{cpu\_operator\_cost} \times \sum_{isBmIdxScan} \texttt{child\_output\_tuples} \\ &= 0.00025 \times \sum_{isBmIdxScan} \texttt{child\_output\_tuples} \end{aligned}$$

3. Charge 100 `cpu_operator_cost` as cost of bit operation between 2 child bitmaps.

$$\begin{aligned} \textit{Run Cost} &\xleftarrow{+} 100 \times \texttt{cpu\_operator\_cost} \times (\texttt{num\_chilren} - 1) \\ &= 0.25 \times (\texttt{num\_chilren} - 1) \end{aligned}$$

4. Calculate the of I/O of accessing pages from base relation. As there are more pages fetched, the percentage of sequential read increases. PostgreSQL designs a function to show this trend clearly.

$$\begin{aligned} \texttt{io\_per\_page} &\leftarrow \texttt{random\_page\_cost} - (\texttt{random\_page\_cost} - \texttt{seq\_page\_cost}) \times \\ &\qquad \sqrt{\texttt{num\_pages\_fetched}/\texttt{num\_rel\_pages}} \\ &= 4 - 3\sqrt{\texttt{num\_pages\_fetched}/\texttt{num\_rel\_pages}} \\ \textit{Run Cost} &\xleftarrow{+} \texttt{io\_per\_page} \times \texttt{num\_pages\_fetched}) \end{aligned}$$

5. Estimate CPU operation cost on each selected tuples on heap. The cost per selected tuple is `cpu_tuple_cost` as base cost, and one additional `cpu_per_operation` is charged for every distinct condtion appears in Recheck Condition or Filter.

$$
\begin{aligned}
\textit{Run Cost} &\overset{+}{\leftarrow} \texttt{num\_tuple\_sel} \times (\texttt{cpu\_tuple\_cost} + \\
&\quad \texttt{cpu\_operator\_cost} \times \texttt{num\_conds}) \\
&= \texttt{num\_tuple\_sel} \times (0.01 + 0.0025 \times \texttt{num\_conds})
\end{aligned}
$$

3.4.3 Bitmap Heap Scan

Bitmap Heap Scan will fetch relation pages from heap using the bitmap.

The procedure of Bitmap Heap Scan cost estimation is shown below:

1. Initialize *Startup Cost* and *Run Cost* using the child node.

$$
\begin{aligned}
\textit{Startup Cost} &\leftarrow \texttt{child\_total\_cost} \\
\textit{Run Cost} &\leftarrow 0
\end{aligned}
$$

2. Estimate number of pages fetched by bitmap. PostgreSQL uses the Mackert and Lohman formula for the case $\texttt{num\_rel\_pages} \leq \texttt{buffer\_pages\_available}$ in this step.

$$
\begin{aligned}
\texttt{num\_pages\_fetched} &\leftarrow \textit{ceil}(\frac{2 \times \texttt{num\_rel\_pages} \times \texttt{input\_tuples}}{2 \times \texttt{num\_rel\_pages} + \texttt{input\_tuples}}) \\
\texttt{num\_pages\_fetched} &\leftarrow \textit{min}(\texttt{num\_rel\_pages},\ \texttt{num\_pages\_fetched})
\end{aligned}
$$

3. Calculate the of I/O of accessing pages from base relation. As there are more pages fetched, the percentage of sequential read increases. PostgreSQL designs a function to show this trend clearly.

$$\texttt{io\_per\_page} \leftarrow \texttt{random\_page\_cost} - (\texttt{random\_page\_cost} - \texttt{seq\_page\_cost}) \times \sqrt{\texttt{num\_pages\_fetched}/\texttt{num\_rel\_pages}}$$
$$= 4 - 3\sqrt{\texttt{num\_pages\_fetched}/\texttt{num\_rel\_pages}}$$
$$\textit{Run Cost} \overset{+}{\leftarrow} \texttt{io\_per\_page} \times \texttt{num\_pages\_fetched})$$

4. Estimate CPU operation cost on each selected tuples on heap. The cost per selected tuple is `cpu_tuple_cost` as base cost, and one additional `cpu_per_operation` is charged for every distinct condtion appears in Recheck Condition or Filter.

$$\textit{Run Cost} \overset{+}{\leftarrow} \texttt{num\_tuple\_sel} \times (\texttt{cpu\_tuple\_cost} + \texttt{cpu\_operator\_cost} \times \texttt{num\_conds})$$
$$= \texttt{num\_tuple\_sel} \times (0.01 + 0.0025 \times \texttt{num\_conds})$$

3.5 Nested Loop Join

Cost estimation of Nested Loop Join in PostgreSQL is similar to the introduced method in lecture, just that PostgreSQL considers cost of CPU operations.

The procedure of estimating cost of Nested Loop Join is shown below:

1. Initialize *Startup Cost* and *Run Cost* using the child node.

$$\textit{Startup Cost} \leftarrow \texttt{outer\_startup\_cost} + \texttt{inner\_startup\_cost}$$
$$\textit{Run Cost} \leftarrow \texttt{outer\_run\_cost} + \texttt{inner\_run\_cost}$$

2. As inner relation will need to be read for `outer_tuples` times, we need add the cost of rescaning the inner relation. PostgreSQL provides a function for different child node types, and we implement that function with

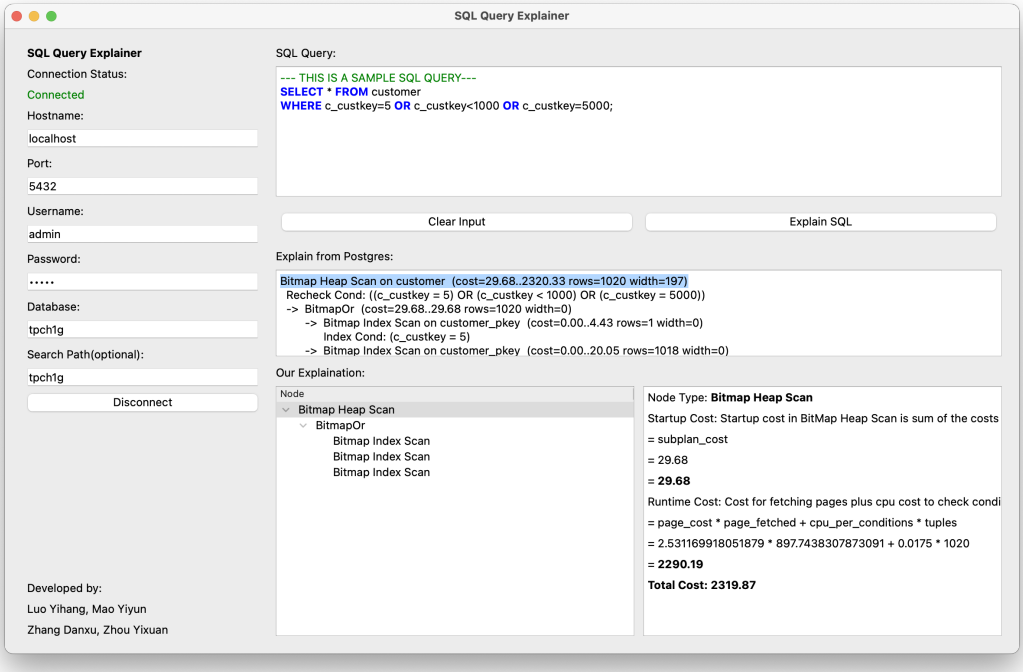

Figure 9: Example of Bit Map Heap Scan.

some simplification. Note that rescan cost for Materialization would contain `_inner_pages` sequential Disk I/O cost if inner relation is too large to fit in buffer memory.

$$\texttt{inner\_rescan\_cost} = \begin{cases} \texttt{cpu\_operator\_cost} \times \texttt{inner\_tuples} \\ \qquad\qquad\qquad\qquad , \ for\ Materialization \\ \texttt{child\_run\_cost} \quad , \ for\ single{-}batch\ Hash\ Join \\ \texttt{child\_total\_cost} \ , \ by\ default \end{cases} \tag{2}$$

$$Run\ Cost \overset{+}{\leftarrow} \texttt{sinner\_rescan\_cost} \times (\texttt{outer\_tuples} - 1)$$

3. Calculate cost of comparing every pairs of inner and outer tuples. Additional `cpu_operator_cost` is charged for every join condition.

$$
\begin{aligned}
\textit{Run Cost} &\xleftarrow{+} \texttt{inner\_tuples} \times \texttt{outer\_tuples} \times (\texttt{cpu\_tuple\_cost} + \\
&\quad \texttt{num\_conds} \times \texttt{cpu\_operator\_cost}) \\
&= \texttt{inner\_tuples} \times \texttt{outer\_tuples} \times (0.01 + 0.0025 \times \texttt{num\_conds})
\end{aligned}
$$

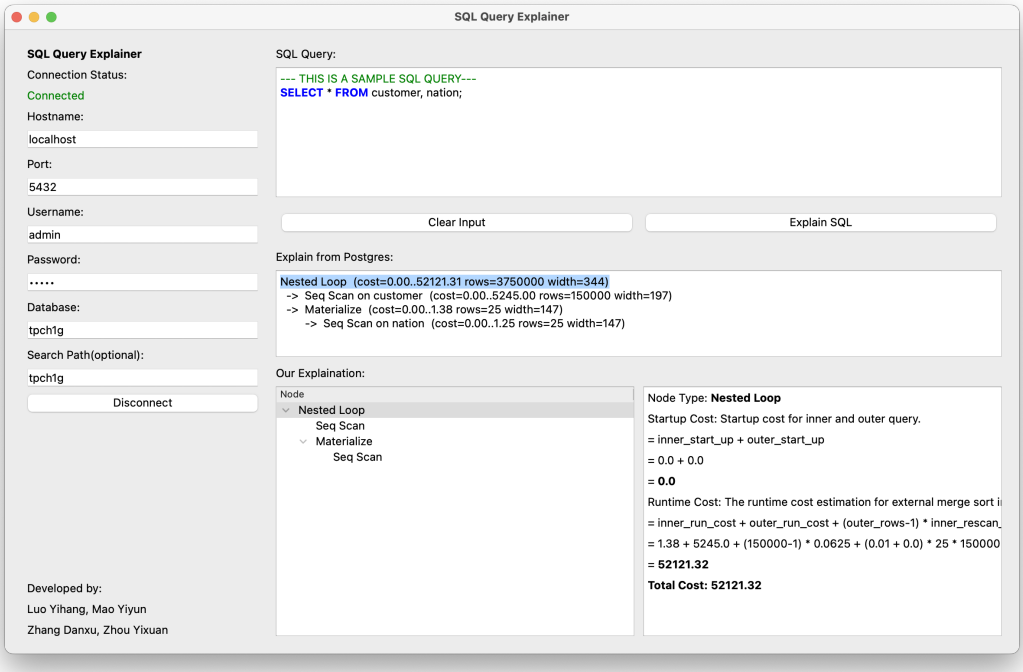

Figure 10: Example of Nested Loop.

3.6 Hash Join

When two relations are joined on non-index attributes, Hash Join is most likely to be the most efficient Join method. In PostgreSQL, complicated algorithms are applied to calculate important parameters like number of buckets of hash function, batches of hash join, and selectivity. The calculation process of these previous parameters involves too many unavailable parameters (e.g., memory allocated, random sampling results, etc.). Moreover, calculation of cost itself is already complex.

We first try to apply the estimation using the methods taught in lecture, but it

turns out to have huge differences in orders of magnitude. Therefore, we switch to using `EXPLAIN ANALYSE` method to get the value of the mentioned parameters that are applied in real execution. We also try to follow all observable instructions in source code. Deviations is still unavoidable, but we have attempted to produce close and reasonable cost estimation steps.

3.6.1 Hash

In PostgreSQL, a Hash node is always a child a Hash Join node. The calculation overhead of hash function is postponed to the Hash Join node. Therefore, Hash node does not change its child's *Startup Cost* and *Run Cost*.

3.6.2 Hash Join

1. Initialize startup cost using the child nodes.

$$\textit{Startup Cost} \leftarrow \texttt{outer\_start\_cost} + \texttt{inner\_total\_cost}$$
$$\textit{Run Cost} \leftarrow \texttt{outer\_run\_cost}$$

2. Estimate the cost of computing hash functions. Every inner and outer tuple should be hashed, so we charge a `cpu_operator_cost` on each of them. We also charge a `cpu_tuple_cost` per inner tuple to estimate the cost to insert inner tuples into hash tables.

$$
\begin{aligned}
\textit{Start Cost} &\overset{+}{\leftarrow} \texttt{inner\_tuples} \times (\texttt{cpu\_operator\_cost} + \texttt{cpu\_tuple\_cost}) \\
&= 0.0125 \times \texttt{inner\_tuples} \\
\textit{Run Cost} &\overset{+}{\leftarrow} \texttt{outer\_tuples} \times \texttt{cpu\_operator\_cost} \\
&= 0.0025 \times \texttt{outer\_tuples}
\end{aligned}
$$

3. If number of hash batches is greater than 1, which means we cannot fit in the content of 1 hash bucket into the memory, we need to write back and read the inner and outer tuples once per batch. We therefore estimate Disk I/O during this process.

$$
\begin{aligned}
&If\ \texttt{num\_batches} > 1: \\
&\qquad \textit{Startup Cost} \overset{+}{\leftarrow} \texttt{inner\_tuples} \times \texttt{seq\_page\_cost} \\
&\qquad\qquad = \texttt{inner\_tuples} \\
&\qquad \textit{Run Cost} \overset{+}{\leftarrow} \texttt{inner\_tuples} \times \texttt{seq\_page\_cost} + \texttt{outer\_tuples} \times \\
&\qquad\qquad 2 \times \texttt{seq\_page\_cost} \\
&\qquad\qquad = \texttt{inner\_tuples} + 2 \times \texttt{outer\_tuples}
\end{aligned}
$$

4. Estimate total tuple comparison costs. PostgreSQL assumes that half of tuples in inner relation will be compared to a single output tuple.

$$
\begin{aligned}
\textit{Run Cost} &\overset{+}{\leftarrow} \frac{\texttt{cpu\_operator\_cost} \times \texttt{outer\_tuples} \times \texttt{inner\_tuples}}{\texttt{num\_buckets} \times 2} \\
&= \frac{0.0025 \times \texttt{outer\_tuples} \times \texttt{inner\_tuples}}{\texttt{num\_buckets} \times 2}
\end{aligned}
$$

5. For each output tuple, we charge `cpu_tuple_cost` as processing cost.

$$\begin{aligned} \textit{Run Cost} &\xleftarrow{+} \texttt{cpu\_tuple\_cost} \times \texttt{output\_tuples} \\ &= 0.01 \times \texttt{output\_tuples} \end{aligned}$$

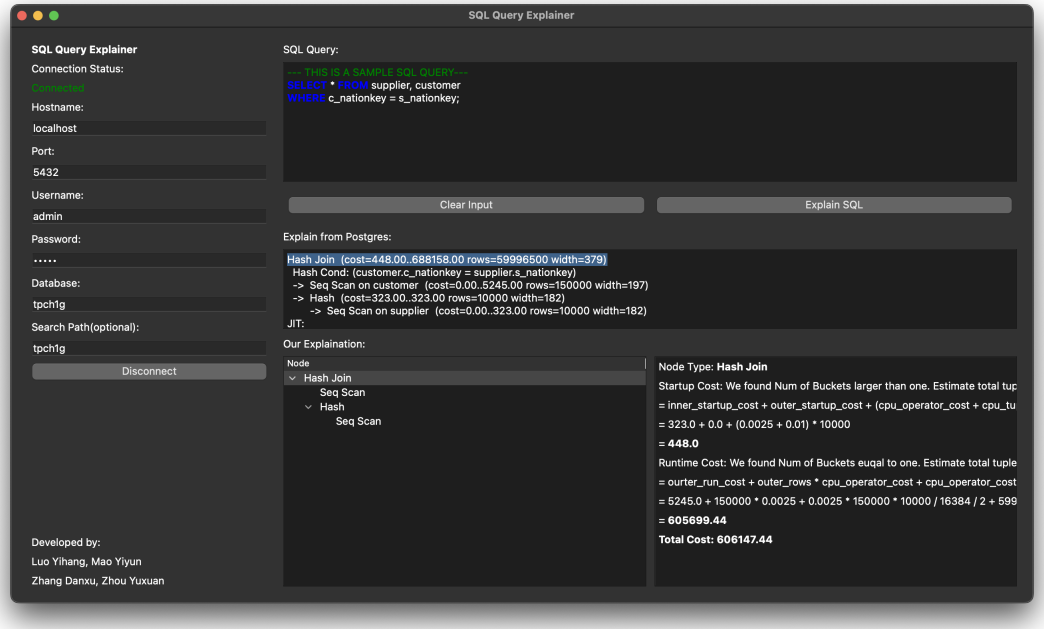

Figure 11: Example of Hash Join.

3.7 Merge Join

In PostgreSQL, each relation is sorted on the join attributes before merge join starts. Then the two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is attractive because each relation has to be scanned only once. The required sorting might be achieved either by an explicit sort step, or by scanning the relation in the proper order using an index on the join key.

In this section, we will explain how PostgreSQL calculates costs for merge join. As mentioned before, PostgreSQL has mechanisms of estimating selectivity and the values are retrieved from cache when needed. As we do not have access to these

cache contents when using the `EXPLAIN` feature, we are going to come up with our own simplified estimations in our cost calculation.

1. Initialize startup cost using the child nodes: Outer input and inner input. Some rows will be skipped before the first join, which should be factored into startup cost, as shown in the second formula below. However, as PostgreSQL read the start_selectivity and end_selectivity parameters from cache which we have no access to, in our code implementation we just set start_selectivity to be 0.

$$\begin{aligned} \textit{Startup Cost} &\leftarrow \texttt{outer\_startup\_cost} + \texttt{inner\_startup\_cost} \\ \textit{Startup Cost} &\overset{+}{\leftarrow} \texttt{outer\_run\_cost} \times \texttt{outer\_start\_selectivity} \\ &\quad + \texttt{inner\_run\_cost} \times \texttt{inner\_start\_selectivity} \end{aligned}$$

2. Initialize run cost using the child nodes: Outer input and inner input. Similar to startup cost, we need to get selectivity parameters from cache to calculate the fraction of child node run cost we need to include for the merge join node.

$$\begin{aligned} \textit{Run Cost} \leftarrow \texttt{outer\_run\_cost} \times (&\texttt{outer\_end\_selectivity} - \\ &\texttt{outer\_start\_selectivity}) + \texttt{inner\_run\_cost} \times \\ &(\texttt{inner\_end\_selectivity} - \texttt{inner\_start\_selectivity}) \end{aligned}$$

   In our code implementation, we use the number of rows of the input nodes and the output to estimate the end_selectivity values. The estimation works better when there are no duplicate values.

$$\texttt{end\_selectivity} \leftarrow min(\texttt{output\_rows/input\_rows}, 1)$$

3. Add `cpu_operator_cost` if materialization is applied on inner input.

$$If\ \texttt{materialize\_inner} == True:$$
$$Run\ Cost \overset{+}{\leftarrow} \texttt{cpu\_operator\_cost} \times \texttt{inner\_row}$$

4. Add cpu cost for the number of tuple comparisons needed, which is approximately number of outer rows plus number of inner rows. For each tuple that gets through the merge join node, charge `cpu_tuple_cost`.

$$\begin{aligned} Startup\ Cost \overset{+}{\leftarrow} & \texttt{num\_conds} \times \texttt{cpu\_operator\_cost} \times (\texttt{outer\_row} \times \\ & \texttt{outer\_start\_selectivity} + \texttt{inner\_row} \times \\ & \texttt{inner\_start\_selectivity}) \\ Run\ Cost \overset{+}{\leftarrow} & \texttt{num\_conds} \times \texttt{cpu\_operator\_cost} \times (\texttt{outer\_row} \times \\ & (1 - \texttt{outer\_start\_selectivity}) + \texttt{inner\_row} \times \\ & (1 - \texttt{inner\_start\_selectivity})) \\ Run\ Cost \overset{+}{\leftarrow} & \texttt{cpu\_tuple\_cost} \times \texttt{output\_rows} \end{aligned}$$

3.8 Materialize

A materialize node means the output of its child node is materialized into memory before the upper node is executed. This is done when the outer node needs a source

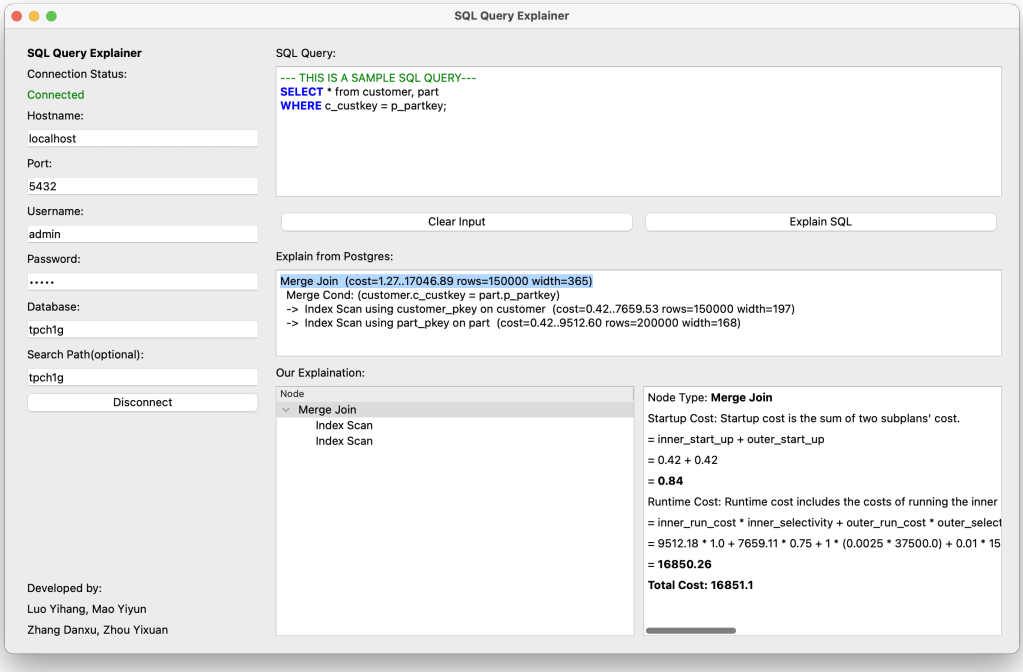

Figure 12: Example of Merge Join.

that it can re-scan later. The materialization process creates pure overheads, but it saves costs for upper operator nodes.

The *Startup Cost* for materialization is the Total Cost of its child node.

$$\textit{Startup Cost} \leftarrow \texttt{child\_total\_cost}$$

We then estimate the memory size needed for materialization. If the needed memory exceeds the `work_mem`, we will need to write the materialized content to disk, resulting in different cost estimations.

$$\texttt{estimated\_mem} \leftarrow \texttt{input\_rows} \times (\texttt{tuple\_width} + \texttt{heap\_tuple\_header\_size})$$

$$\texttt{disk\_pages} \leftarrow \mathit{ceil}(\texttt{estimated\_mem}/\texttt{block\_size})$$

$$\mathit{Run\ Cost} = \begin{cases} 2 \times \texttt{input\_rows} \times \texttt{cpu\_operation\_cost} \\ \qquad\qquad\qquad\qquad\qquad\qquad , if\ \texttt{estimated\_mem} \leq \texttt{work\_mem} \\ \texttt{disk\_pages} \times \texttt{seq\_page\_cost} + 2 \times \texttt{input\_rows} \times \\ \texttt{cpu\_operation\_cost} \qquad\qquad , if \texttt{estimated\_mem} > \texttt{work\_mem} \end{cases} \tag{3}$$

3.9 Aggregate

For aggregation, the cost estimation is related to the number of conditions and also the types of the conditions. `cpu_operation_cost` is charged per input rows on every distinct filter conditions, and `cpu_tuple_cost` is charged for all output rows, similar to many nodes explained above.

$$\mathit{Startup\ Cost} \leftarrow \texttt{child\_total\_cost} + \texttt{cpu\_operation\_cost} \times \texttt{input\_rows} \times \texttt{number\_of\_aggregation\_operations}$$

$$\mathit{Run\ Cost} \leftarrow \texttt{cpu\_tuple\_cost} \times \texttt{output\_rows}$$

4 Conclusion

In conclusion, our project represents a comprehensive investigation into the database QEP and cost estimation. Through in-depth examination of PostgreSQL source code and the development of precise cost estimation algorithms, we have significantly enhanced our understanding of query computational costs. We have implemented a robust and user-friendly application designed to explain the cost of a QEP based on user-input SQL queries. We believe our efforts extend beyond a course project, creating valuable resource for helping people deepen their understanding of QEP. We plan to release our code in public after the end of the project.

A Appendix

A.1 Text UI

In this project, we additionally provide a Text-UI for users to run our application in the terminal. The Text-UI is also well formatted. You can start Text-UI via:

`python ./src/explain.py`

For example, for input query:

`SELECT * FROM nation, customer;`

Output Text:

```
1  *************
2  | Node type: Nested Loop
3  | Startup cost: 0.0 Total cost: 52176.31
4  |
5  | -Startup Cost-
6  | Explaination: Startup cost for inner and outer query
7  |
8  | estimated_cost
9  | = inner_start_up + outer_start_up
10 | = 0.0 + 0.0
11 | = 0.0
12 |
13 | -Runtime Cost-
14 | Explaination: The runtime cost estimation for external merge
       sort in PostgreSQL accounts for the costs of
15 |  processing the initial and subsequent runs, rescanning inner
       runs, and the per-tuple processing and quality restriction
16 |  costs for all combinations of inner and outer tuples
17 |
18 | estimated_cost
19 | = inner_run_cost + outer_run_cost + (outer_rows-1) *
       inner_rescan_cost + (cpu_tuple_cost + restrict quality cost) *
       inner_tuples
20 |  * outer_tuples
```

```
| = 1.38 + 5300.0 + (150000-1) * 0.0625 + (0.01 + 0.0) * 25 *
   150000
|
| = 52176.32
|
| Total Cost: 52176.32
*************

    *************
    | Node type: Seq Scan
    | Startup cost: 0.0 Total cost: 5300.0
    |
    | -Startup Cost-
    | Explaination: Start up cost is the sum of all sub plans
   costs.
    |
    | estimated_cost
    | = sub_cost
    | = 0.0
    | = 0.0
    |
    | -Runtime Cost-
    | Explaination: Runtime cost is seq scan cost with sub plan
   cost
    |
    | estimated_cost
    | = (disk_pages_read * seq_page_cost) + (rows_scanned *
   cpu_tuple_cost) + sub_cost
    | = 3800 * 1.0 + 150000.0 * 0.01 + 0.0
    | = 5300.0
    |
    | Total Cost: 5300.0
    *************

    *************
    | Node type: Materialize
    | Startup cost: 0.0 Total cost: 1.38
```

```
 |
 | -Startup Cost-
 | Explaination: The start up cost is the sum of all sub plans costs.
 |
 | estimated_cost
 | = sub cost
 | = 1.25
 | = 1.25
 |
 | -Runtime Cost-
 | Explaination: Materialize is employed for transient storage of intermeidiate results. Charging the cpu_operator_cost by 2x per
 |  tuple to account for the additional bookkeeping overhead.
 |
 | estimated_cost
 | = 2 * (rows_scanned * cpu_operator_cost)
 | = 2 * 25 * 0.0025
 | = 0.12
 |
 | Total Cost: 1.38
 *************

     *************
     | Node type: Seq Scan
     | Startup cost: 0.0 Total cost: 1.25
     |
     | -Startup Cost-
     | Explaination: Start up cost is the sum of all sub plans costs.
     |
     | estimated_cost
     | = sub_cost
     | = 0.0
     | = 0.0
     |
```

```
87        | -Runtime Cost-
88        | Explaination: Runtime cost is seq scan cost with sub
    plan cost
89        |
90        | estimated_cost
91        | = (disk_pages_read * seq_page_cost) + (rows_scanned *
    cpu_tuple_cost) + sub_cost
92        | = 1 * 1.0 + 25.0 * 0.01 + 0.0
93        | = 1.25
94        |
95        | Total Cost: 1.25
96        **************
```

Listing 2: Text-UI Explaination Output.

A.2 Individual Contribution Claims

In this project, our team operates with a highly efficient organizational structure. Our collaborative efforts commence with thorough research on software design, algorithm research, code implementation and report writing. To foster regular communication and knowledge exchange, we convene weekly group meetings, during which we collectively share insights into the methodologies employed, analyze experiment results, and engage in comprehensive discussions on these methodologies.

Name	Contribution
Luo Yihang	Backend Development, Algorithm Implementation.
Mao Yiyun	Frontend Development, Algorithm Implementation.
Zhou Yuxuan	Backend Development, Algorithm Implementation.
Zhang Danxu	Frontend Development, Algorithm Implementation.

Table 2: Contribution Table.

** Each member in our team equally contributes to the project.*