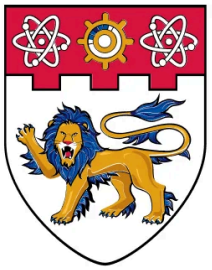


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC3020 Database System Principles

School of Computer Science and Engineering
AY23/24 Semester 2

Project 2

Group No. : 7

Jeffrey Lim Yi Ren (U2120937B)
Karishein Chandran (U2122012G)
Chen Hongpo (U2022097E)
Don Seo Kang Rui (U2120866H)

Due On: 21 April 2024, 11:59 pm

Contribution of each group member:

Name	Contribution
Jeffrey Lim Yi Ren	Report, Program
Karishein Chandran	Report, GUI Interface
Chen Hongpo	Report, Program
Don Seo Kang Rui	Report, Program

Table of Contents

1 Introduction

The goal of our report is to explain our implementation of a Query Execution Plan (QEP) where our program explains the computation of the estimated cost and other relevant statistics when given an SQL query. The program will display a graph for visualisation of each of the steps taken in order.

2 TPC_H Dataset

Visual Studio was used to build *.tbl* files for each table from the *tpch.vcproj* file within the downloaded zip file. The dataset provided for use to test the queries are as follows:

Table	**Description**	**Cardinality**
region	Continents the countries are located in	5
nation	Countries supported by the store	25
part	Part's details (name, manufacturer, brand, type …)	200000
supplier	Supplier's details (name, address, phone …)	10000
partsupp	Supply of the parts details (available quantities, supply cost, …)	800000
customer	Customer's details (name, address, phone …)	150000
orders	Order details (order status, order date, order priority …)	1500000
lineitem	Item in order detail (quantity, extended price, discount, …)	6001215

<Figure 1: Information about tables and size of tables.>

3 Setup of PostgreSQL Database

The PostgreSQL database was set up with the PostgresSQL application using the command line *CREATE DATABASE TPC_H*; and accessed using pgAdmin 4. The database was created with the name TPC_H on default port 5432. The required tables (region, nation, part, supplier, partsupp, customer, orders and lineitem) were then created using the *Create_tables.txt,* which contains the CREATE table commands of each table provided in the Project 2 brief.

A custom Python function was used to convert the built *.tbl* files to usable *.csv* files and clean up trailing '|' characters.

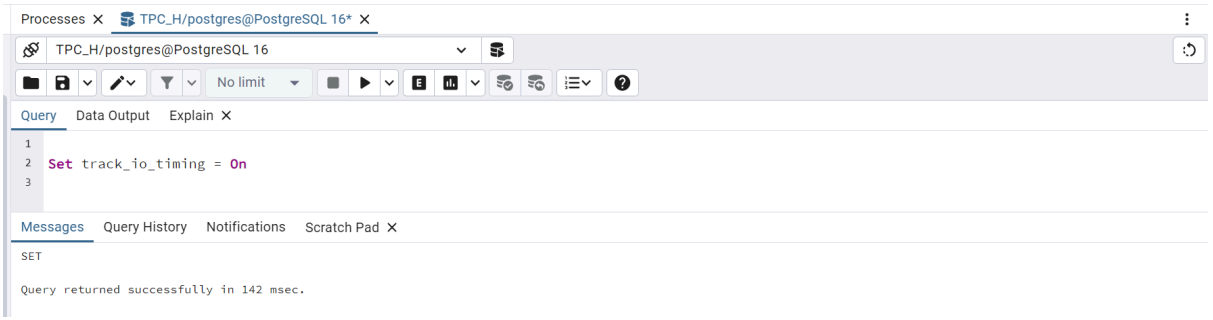
```
def TblToCsv(filename):
    csv = open("".join([path, filename, ".csv"]), "w+")
    tbl = open("".join([path, filename, ".tbl"]), "r")
    lines = tbl.readlines()
    for line in lines:
        line = line.replace("|","")
        csv.write(line)
    tbl.close()
    csv.close()
    return
```

The CSV files were then imported into the TPC_H database created in PostgreSQL with PgAdmin4. It could also be done using the SQL copy commands in *copy_tables.txt.*

Type	Server	Object	Start Time	Status	Time Taken (sec)
Import Data	PostgreSQL 16 (localhost:5432)	TPC_H/public.lineitem	4/20/2024, 3:06:53 PM	Finished	210.55
Import Data	PostgreSQL 16 (localhost:5432)	TPC_H/public.orders	4/20/2024, 3:06:20 PM	Finished	22.28
Import Data	PostgreSQL 16 (localhost:5432)	TPC_H/public.customer	4/20/2024, 3:06:00 PM	Finished	2
Import Data	PostgreSQL 16 (localhost:5432)	TPC_H/public.partsupp	4/20/2024, 3:05:39 PM	Finished	14.81
Import Data	PostgreSQL 16 (localhost:5432)	TPC_H/public.supplier	4/20/2024, 3:05:25 PM	Finished	0.23
Import Data	PostgreSQL 16 (localhost:5432)	TPC_H/public.part	4/20/2024, 3:05:03 PM	Finished	1.17
Import Data	PostgreSQL 16 (localhost:5432)	TPC_H/public.nation	4/20/2024, 3:04:50 PM	Finished	0.15
Import Data	PostgreSQL 16 (localhost:5432)	TPC_H/public.region	4/20/2024, 3:04:33 PM	Finished	0.26

<Figure 2: Uploading import status shown in pgAdmin console.>

An additional feature of **Enabling I/O tracking** was turned on, once track_io_timing is enabled and PostgreSQL is restarted, the server will start tracking I/O timing information. We can then execute queries and analyse the timing information to see detailed information about the time spent on I/O operations in the QEP result.



<Figure 3: Set track IO timing status shown in pgAdmin console.>

4 Details of the Program

Overview of File Structure

1. **csv_update.py** - Used to convert the .tbl tables that were generated from TPC-H in Visual Studio to CSV files and clean up the trailing '|' characters.
2. **project.py** - The main file used to call the other functions.
3. **interface.py** - Our main script that generates the GUI, responsible for DB credentials login, SQL querying interface, QEP Graph diagram, and QEP Explanation text box.
4. **explain.py** - The main script that handles the explanation of the query plan generated from Postgres.
5. **queries.py** - A list of queries defined in the dictionary. Used by the interface.py to provide pre-defined queries

Libraries Used

1. **customtkinter** - A customised modern version of the Python GUI library Tkinter, offering additional functionalities or modifications for creating graphical user interfaces (GUIs).
2. **graphviz** - A Python interface to Graphviz, a graph visualisation tool used to programmatically generate, manipulate, and render graphs and networks in various formats.
3. **Pillow** - A powerful Python imaging library for image processing tasks such as opening, manipulating, and saving images in various formats.
4. **psycopg2** - A PostgreSQL database adapter for Python, allowing Python applications to interact with PostgreSQL databases by executing SQL queries and managing transactions.

Installation Instruction

1. Download Python version 3.12 and above.
2. Install the requirements in the *requirements.txt* file by running the command: *pip install -r requirements.txt*
3. Download Graphviz from the following link (the latest version is graphviz-10.0.1): https://graphviz.org/download/

Note: If you are using Windows, You must add the Graphviz bin folder to your PATH environment variable so that your system can find the Graphviz executables.

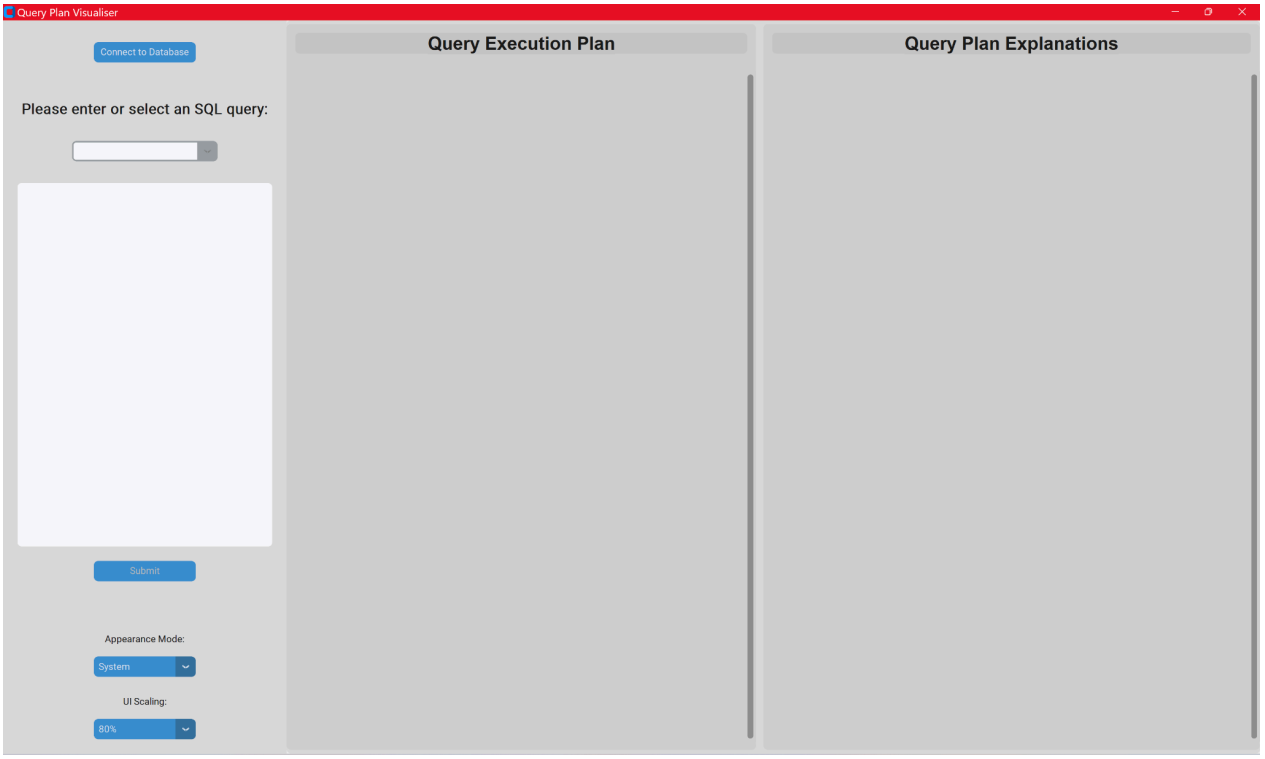
- Open the Start menu and search for "Environment Variables". Click on the "Edit the system environment variables" option that appears.
- In the System Properties window that appears, click on the "Environment Variables" button.
- In the Environment Variables window, scroll down to the "System Variables" section and find the "Path" variable. Click on the "Edit" button.
- In the Edit Environment Variable window, click on the "New" button and enter the path to the Graphviz bin folder. E.g. "C:\Program Files (x86)\Graphviz2.38\bin" (depending on your Graphviz version and installation location).
- Click "OK" on all windows to close them and save your changes.
- Once you have added the Graphviz bin folder to your PATH environment variable, restart your Windows PC. The changes will be made and you should be able to run Graphviz from the command line or from within your Python code.

4. Run the project.py file to start the program.

5 interface.py

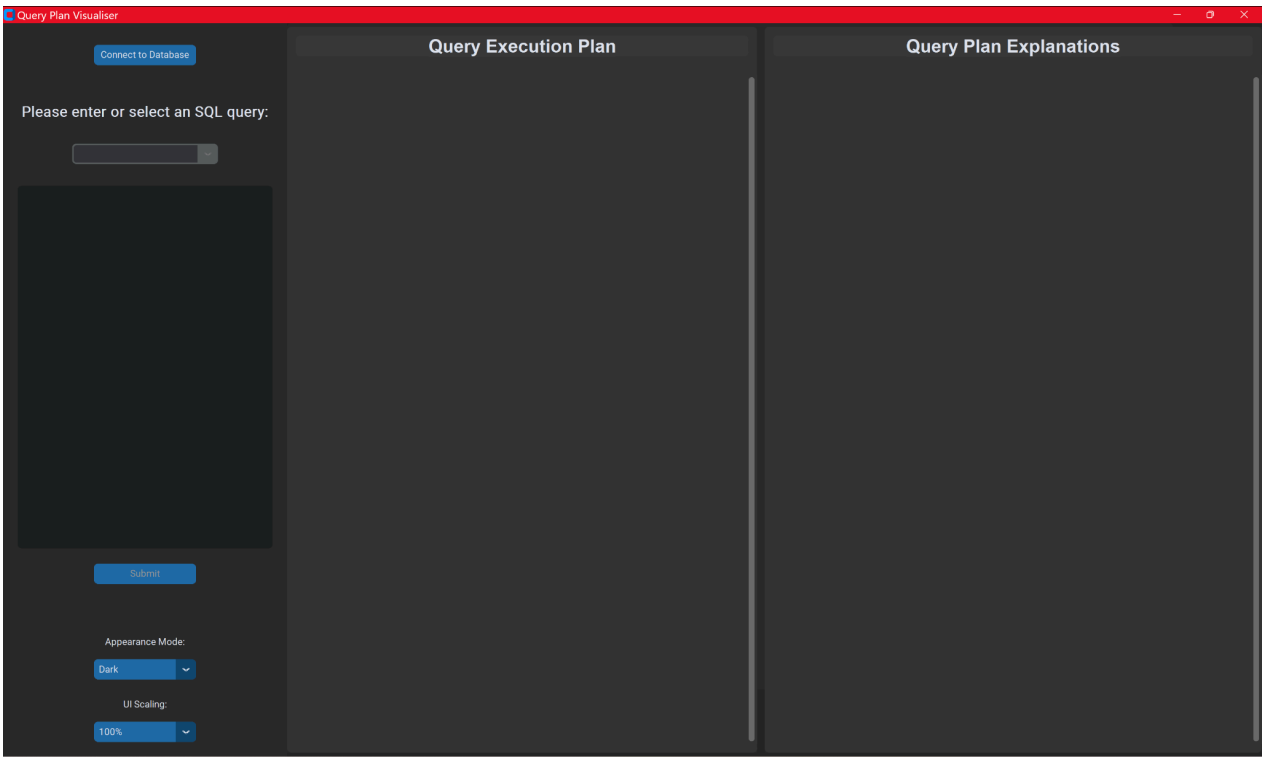
Overview of GUI

The interface file contains the main codes for the GUI. It was created using customtkinter, a Python UI library based on Tkinter that provides new, modern and fully customisable widgets. They are designed and used like standard Tkinter widgets and can be combined with the usual Tkinter elements.



<Figure 4: Query Plan Visualiser Main GUI>

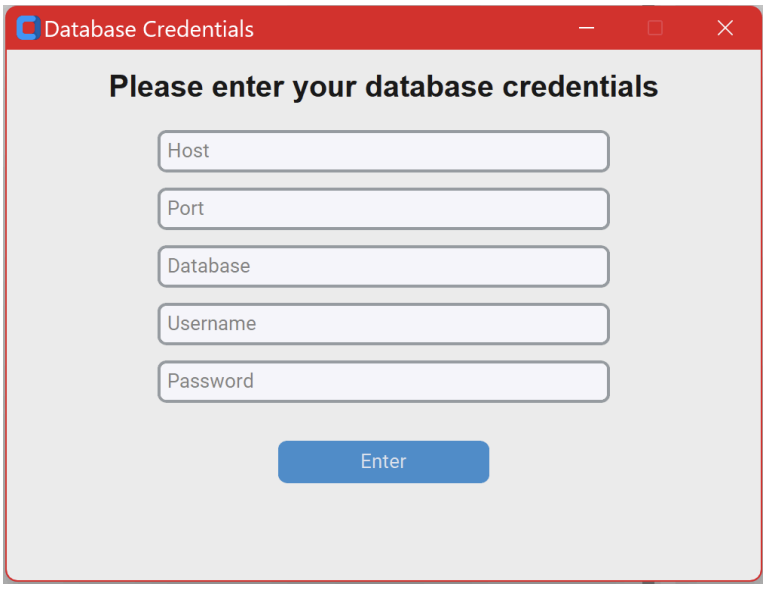
There are also other features incorporated in the GUI, such as the light, dark, and system modes, which allow the user to adjust the background colour of the GUI to suit their needs. Scaling options are also available to decrease or increase the size of the contents in the GUI.



<Figure 5: Query Plan Visualiser Main GUI in dark mode>

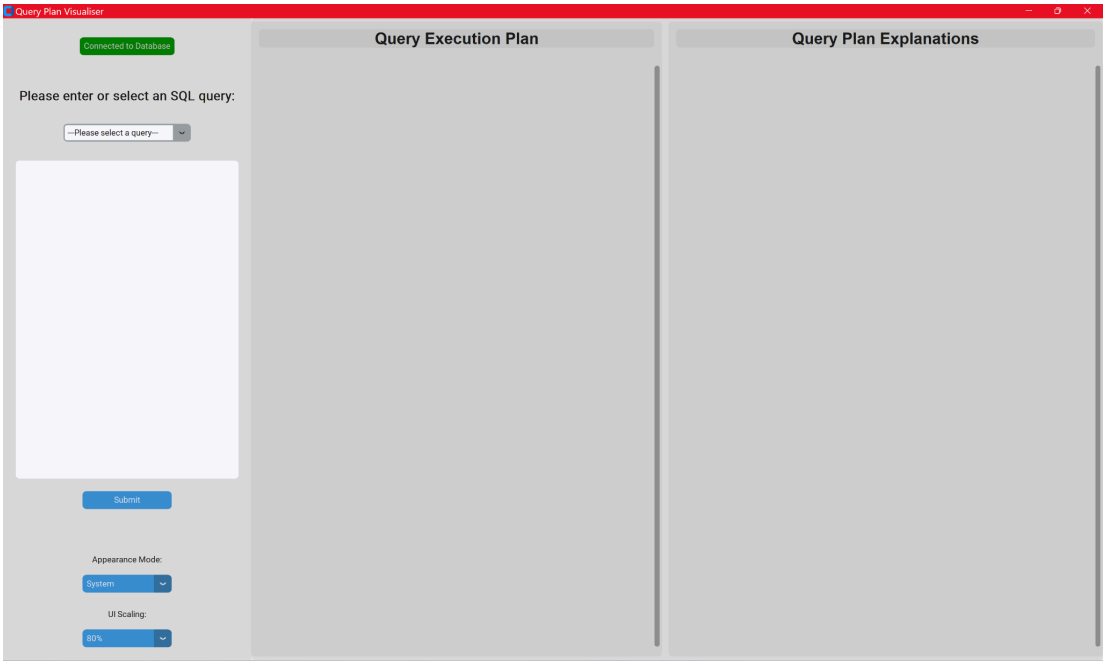
Database Login Window

We incorporated a database login window where the user must specify the credentials to connect to the database before using the system. To begin the connection, the user must click the 'Connect to Database' button. As shown in the figure below, a login window will open for the user to enter database credentials.



<Figure 6: Database Login Window>

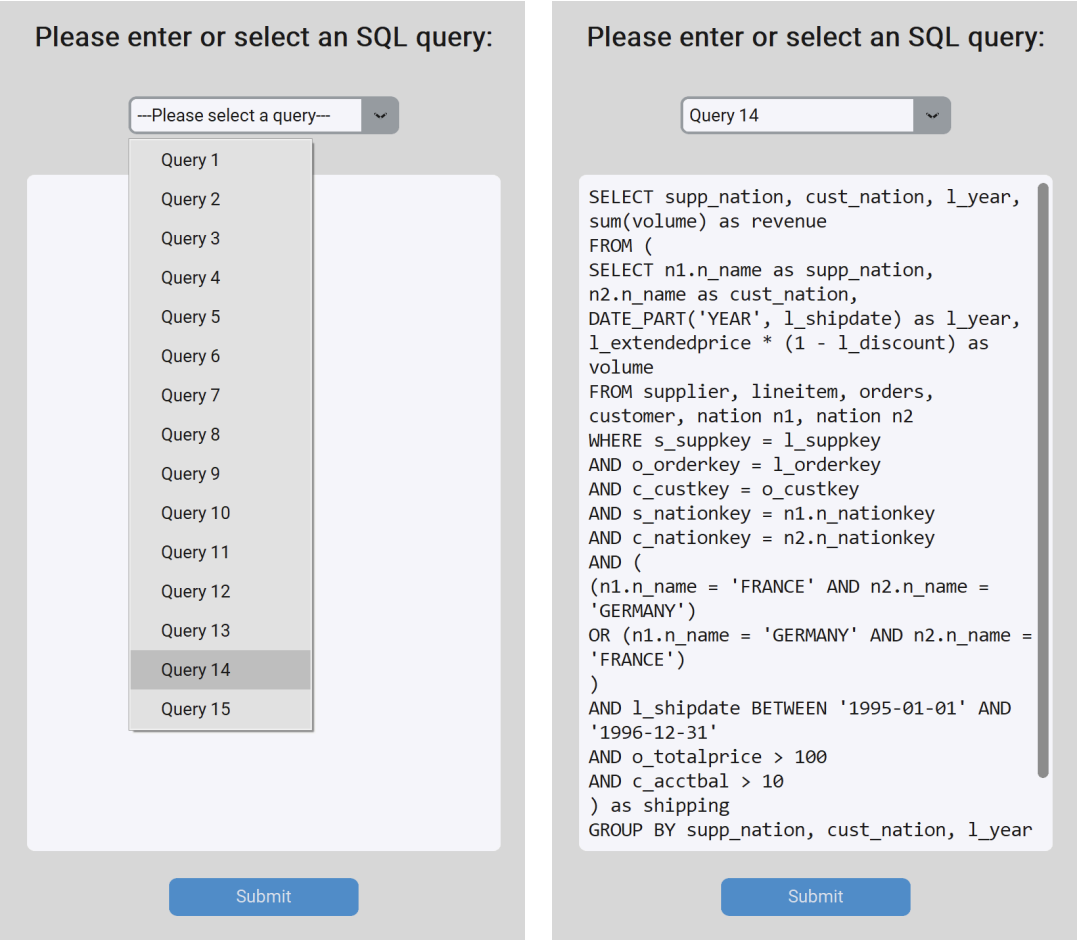
Once the login is successful, a success message will appear. The ‘Connect to Database’ button will turn green, as shown in the figure below. Users can select or manually type queries only after successfully connecting to the database. Otherwise, the query widgets will disabled.



<Figure 7: Database connected and widgets active>

Query Selection and Input

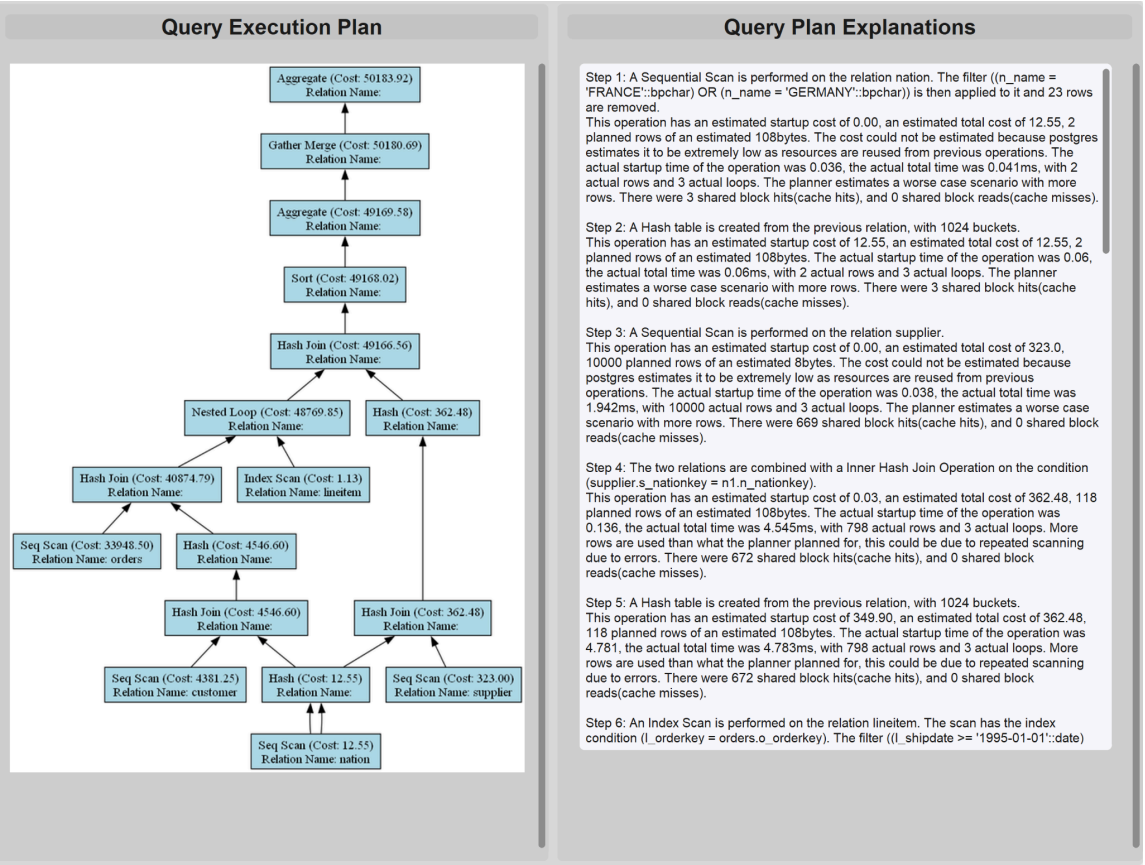
The sidebar on the left side allows the user to select pre-defined queries from the dropdown list or manually input their own queries in the text box. The user can click the ‘Submit’ button to submit the query. Any errors with the query will be flagged with appropriate warning messages.



<Figure 8: Dropdown list of pre-defined queries and queries in the text box>

Query Plans and Explanation

Once a user query is submitted, a QEP diagram will be shown in the ‘Query Execution Plan’ frame. The diagram is generated from the QEP obtained from Postgres. This will be further explained in the next part. The explanation of the QEP in natural language will be displayed in the ‘Query Plan Explanations’ frame. The QEP explanations portion will be explained further in section 6.



<Figure 9: Query Plan Explanation in the text box>

When a user submits a query, the *submit_query()* function will be called. The following statement is appended to the user query: EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON) to get the QEP plan from the PostgreSQL database. This will extract the plan from the PostgreSQL database and return the result in JSON format. The function also checks if the query submitted is error-free. The database will flag any error and incur an error message in the GUI.

Query Plan Diagram Generation

This section will explain the generation of the graph, which is used to visualise the QEP. Since the graph follows a tree structure with the direction going from bottom to top, the visualisation helps understand the data flow and operations during the query execution. We only require a few components for our tree nodes, such as the node type (operation), the total cost of the operation, and the name of the relation involved in that operation. Hence, we send the JSON results we received to the *extract_qep_plan(plan)* function to extract only the relevant information. This is a recursive function which iterates over the nested structure of the QEP JSON object. The function extracts the information needed at each recursion level, stores it in a dictionary, and returns it. The resultant dictionary, qep_plan, is sent to the

display_qep_plan_explanation(qep_plan, explanations) function, along with the explanations of the QEP plan (explained in section 6).

In this function, the Graph Generation Class generates the graph based on the provided dictionary using the Graphviz module. It consists of the *build_dot()* function, which is a recursive approach for generating the graph in the DOT format, and the *generate_graph()* function, which generates and saves the graph. (The DOT syntax is a plain text graph description language that is used to describe graphs and networks).

```
# This class is used to generate the graph based on the query execution plan
class GraphGeneration:
    def __init__(self, qep_plan):
        self.qep = qep_plan
        # Pre-defining the graph and node's visualization attributes
        graph_attribute = {'bgcolor': 'white', 'rankdir': 'BT'}
        node_attribute = {'style': 'filled', 'color': 'black', 'fillcolor': 'lightblue'}
        self.graph = Digraph(graph_attr=graph_attribute, node_attr=node_attribute)
```

<Figure 10: Code snippet of Graph Generation class and initialisation>

The *build_dot()* function takes the qep_plan dictionary as input and outputs a DOT-formatted graph representing the plan in a tree structure. The graph is generated by creating nodes for each operation in the plan and edges that connect the nodes to show the data flow between the operations. The edges are directed and represent the order in which the operations are executed.

The function first assigns a unique ID to each node in the graph based on the hash of the node's properties. The node's label will contain the node's type and cost and the relation name if the node represents a relation. The function then recursively calls itself for each child node in the plan to create the complete graph.

```
# A recursive function to build the graph
def build_dot(self, qep, parent=None):
    node_id = str(hash(str(qep)))
    label = f"{qep['Node Type']} (Cost: {qep['Total Cost']:.2f})"
    if 'Relation Name' in qep:
        label += f"\nRelation Name: {qep['Relation Name']}"
    shape = 'box'
    self.graph.node(node_id, label, shape=shape)
    if parent is not None:
        self.graph.edge(node_id, parent)
    if 'Plans' in qep:
        for plan in qep['Plans']:
            self.build_dot(plan, node_id)
```

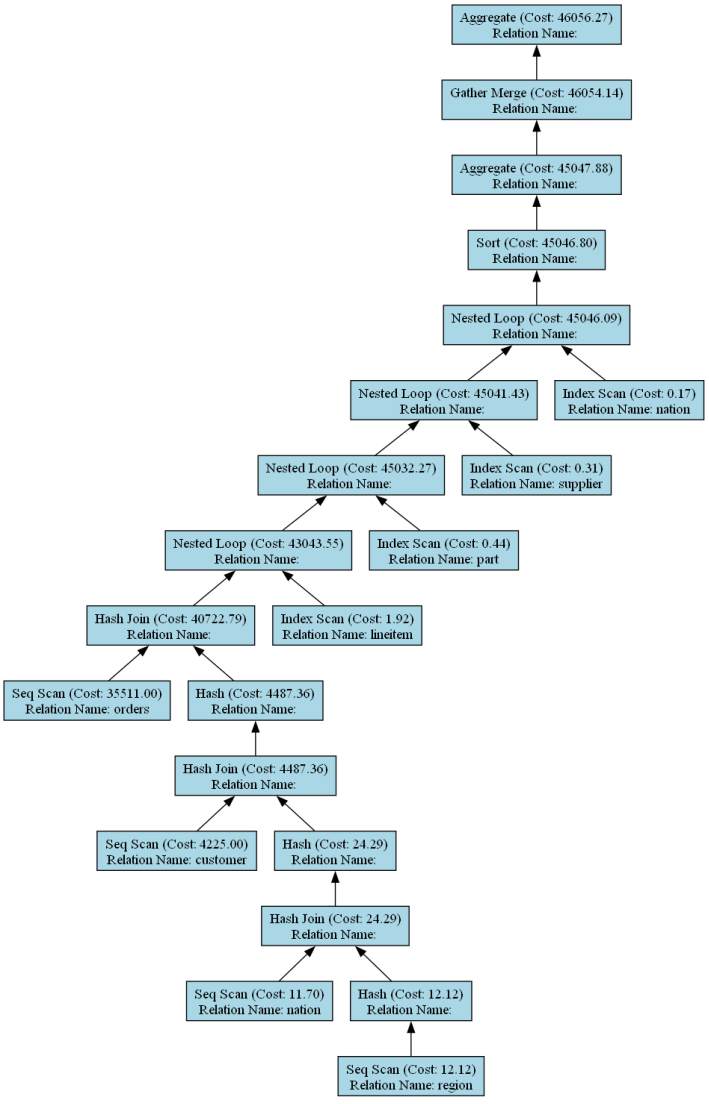
<Figure 11: Code snippet of *build_dot()* function>

The *generate_graph()* function is called after the *build_dot()* function to apply the visual properties, such as the nodes' and edges' shape and colour, to the resulting graph. This function then renders the graph in a PNG image format and passes it back to the GUI application to display the QEP plan diagram.

```
# This function is used to generate the graph into png format
def generate_graph(self, query_plan, format='png', view=True):
    for qep_node in self.qep:
        self.build_dot(qep_node)
    self.graph.attr('node', shape='box')  # set the shape of the nodes
    self.graph.render(query_plan, format=format, view=False, cleanup=True)
```

<Figure 12: Code snippet of *generate_graph()* function>

An example of a generated QEP graph can be seen in the figure below. The root node represents the last operation.



<Figure 13: QEP Tree Diagram>

6 explain.py

After getting the QEP result from the query with 'EXPLAIN ANALYZE BUFFER VERBOSE' command, the program will call the *analyze_and_explain()* function in the explain.py. This object will first be parsed using the *parse_qep()* function that converts the QEP into a structured JSON string format. It is then loaded back as a dictionary using *json.loads()*.

Analysing QEP output

After obtaining the QEP in JSON string. The program will build a graph of QEP nodes. Each QEPNode is an object that contains the Plan object from QEP, it then recursively adds child nodes to the root node by calling the *add_subplans()* function. The *add_subplans()* function iterates through each subplan in the QEP's plans and creates corresponding child nodes, linking them to the parent node. This process continues recursively until all subplans are added, resulting in a hierarchical graph representation of the QEP.

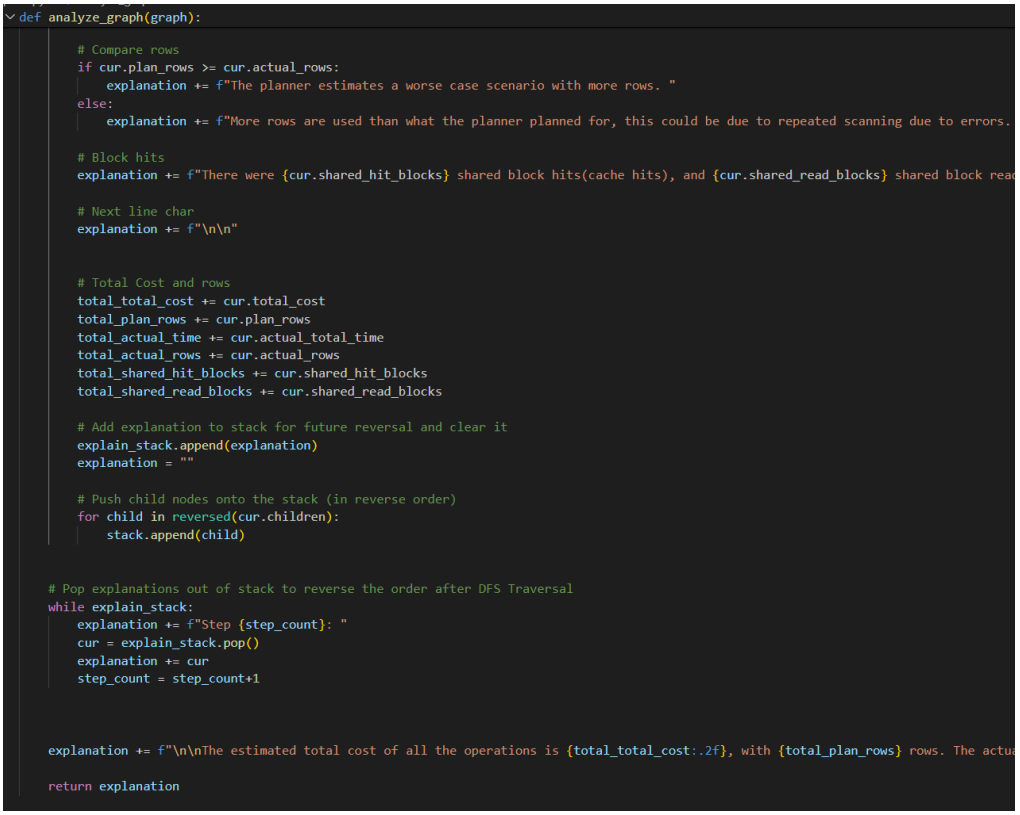
```
def build_graph(qep):
    def create_node(plan):
            node_type = plan["Node Type"],
            relation_name = relation_name,
            startup_cost = plan["Startup Cost"],
            total_cost = plan["Total Cost"],
            plan_rows = plan["Plan Rows"],
            plan_width = plan["Plan Width"],
            actual_startup_time = plan["Actual Startup Time"],
            actual_total_time = plan["Actual Total Time"],
            actual_rows = plan["Actual Rows"],
            actual_loops = plan["Actual Loops"],
            output = output,
            index_cond = index_cond,
            filter = filter,
            rows_removed = rows_removed,
            hash_cond = hash_cond,
            merge_cond = merge_cond,
            join_type = join_type,
            sort_method = sort_method,
            sort_key = sort_key,
            sort_space_used = sort_space_used,
            hash_buckets = hash_buckets,
            strategy = strategy,
            group_key = group_key,
            shared_hit_blocks = plan["Shared Hit Blocks"],
            shared_read_blocks = plan["Shared Read Blocks"],
            shared_dirtied_blocks = plan["Shared Dirtied Blocks"],
            shared_written_blocks = plan["Shared Written Blocks"],
            local_hit_blocks = plan["Local Hit Blocks"],
            local_read_blocks = plan["Local Read Blocks"],
            local_dirtied_blocks = plan["Local Dirtied Blocks"],
            local_written_blocks = plan["Local Written Blocks"],
            temp_read_blocks = plan["Temp Read Blocks"],
            temp_written_blocks = plan["Temp Written Blocks"],
        )

    def add_subplans(parent_node, subplans):
        for subplan in subplans:
            child_node = create_node(subplan)
            parent_node.children.append(child_node)
            add_subplans(child_node, subplan.get("Plans", []))

    root_node = create_node(qep[0]["Plan"])
    add_subplans(root_node, qep[0]["Plan"].get("Plans", []))
    return root_node
```

<Figure 14: code snippet of build graph for QEPNodes>

Each QEPNode is added according to the details retrieved from QEP, such as operation, operating condition, information on block reads, startup time, costs, actual startup time and more. After building the graph of QEPnodes, the program will run the *analyze_graph()* function.



```
def analyze_graph(graph):

    # Compare rows
    if cur.plan_rows >= cur.actual_rows:
        explanation += f"The planner estimates a worse case scenario with more rows. "
    else:
        explanation += f"More rows are used than what the planner planned for, this could be due to repeated scanning due to errors.

    # Block hits
    explanation += f"There were {cur.shared_hit_blocks} shared block hits(cache hits), and {cur.shared_read_blocks} shared block rea

    # Next line char
    explanation += f"\n\n"


    # Total Cost and rows
    total_total_cost += cur.total_cost
    total_plan_rows += cur.plan_rows
    total_actual_time += cur.actual_total_time
    total_actual_rows += cur.actual_rows
    total_shared_hit_blocks += cur.shared_hit_blocks
    total_shared_read_blocks += cur.shared_read_blocks

    # Add explanation to stack for future reversal and clear it
    explain_stack.append(explanation)
    explanation = ""

    # Push child nodes onto the stack (in reverse order)
    for child in reversed(cur.children):
        stack.append(child)


# Pop explanations out of stack to reverse the order after DFS Traversal
while explain_stack:
    explanation += f"Step {step_count}: "
    cur = explain_stack.pop()
    explanation += cur
    step_count = step_count+1



explanation += f"\n\nThe estimated total cost of all the operations is {total_total_cost:.2f}, with {total_plan_rows} rows. The actu

return explanation
```

<Figure 15: code snippet of analyze_graph>

Since we're looking at a bottom-up approach, we employ the use of a stack, the QEPNodes in the graph are 'pop', and each child nodes, which are the subplans, are pushed into the stack in reversed order. This is to handle logical operations such as hash on a table followed by hash join.

Node Types and Operations

For Node types, there are 18 scan types, 2 caching types, 3 join methods, 2 sorting methods, 3 union methods, 2 parallelisation and additional methods for single table functions such as hash, limit, group, and aggregate. This will be used in our explanation output.

Estimating cost in Postgres QEP

This function handles the comparison between actual cost and estimated cost driven from the node_type. Rows accessed, block hits and missed are also calculated here and provided with a short explanation of these operations.

For cost estimation, the costs are primarily hypothetical and are used for comparing different query plans to find the one that is expected to be the least expensive in terms of system resources. Postgres mainly looks at these 5 factors:

1. **Disk I/O Costs:** Estimated costs of reading the data from disk, which can be influenced by the reads if they are sequential or random. System parameters like seq_page_cost and random_page_cost are also used to model these costs.
2. **CPU Costs:** Estimates of the computational resources needed to process query operations like joins, sorts, and filters. Configurable parameters like cpu_tuple_cost, cpu_index_tuple_cost, and cpu_operator_cost quantify these costs.
3. **Row Estimates and Selectivity:** Using statistics gathered from the database (e.g., histograms, row counts), PostgreSQL estimates how many rows each operation within the query will likely process
4. **Index Usage**: The presence and type of indexes can significantly impact the cost, especially in reducing the number of disk accesses required.
5. **Network Costs:** Since our program has already been connected to the PostgresSQL database on our local machine, network costs are negligible and will not be taken into account for our project.

For row estimates and selectivity factors, statistical analysis gathered from database about the data stored in the database using the ANALYZE command, which gathers information such as:

1. Table size and number of rows.
2. Histograms of data distributions in columns.
3. Correlations between columns.
4. Frequencies of distinct values.

The command below can be used to retrieve disk pages and rows scanned of a table, where *part* is the table name.

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'part';
```

These statistics are used to estimate the selectivity of query predicates (i.e., the fraction of rows that a filter or join condition is expected to match).

These factors are combined into a cost model that estimates both the "startup cost" (the cost to begin returning the first row) and the "total cost" (the cost to complete the entire query). The formula below:

(disk pages read * **seq_page_cost**) + (rows scanned * **cpu_tuple_cost**)

where disk pages and rows scanned depend on the table while seq_page_cost defaults to 1.0 and cpu_tuple_cost defaults to 0.01.

The PostgreSQL optimiser uses these estimates to choose between multiple potential execution plans, selecting the one with the lowest estimated cost.

Cost Calculation from QEP

The below factors are used in deriving the actual cost calculation from the QEP.

Factors Involved:

1. **Actual Disk I/O:** The real number of disk pages read or written, which might differ from estimates due to factors like cache hits or system load at runtime.
2. **Actual CPU Usage:** The real CPU time consumed by the query, influenced by runtime conditions such as CPU availability, background processes, or system optimisations not accounted for in the planning phase.
3. **Concurrency and System Load:** Other processes and queries running on the system can impact the actual performance, causing discrepancies from the QEP's estimated costs.
4. **Network Delays:** Since our program is connected to the PostgresSQL database on our own local machine, network costs are negligible and will not be taken into account for our project.

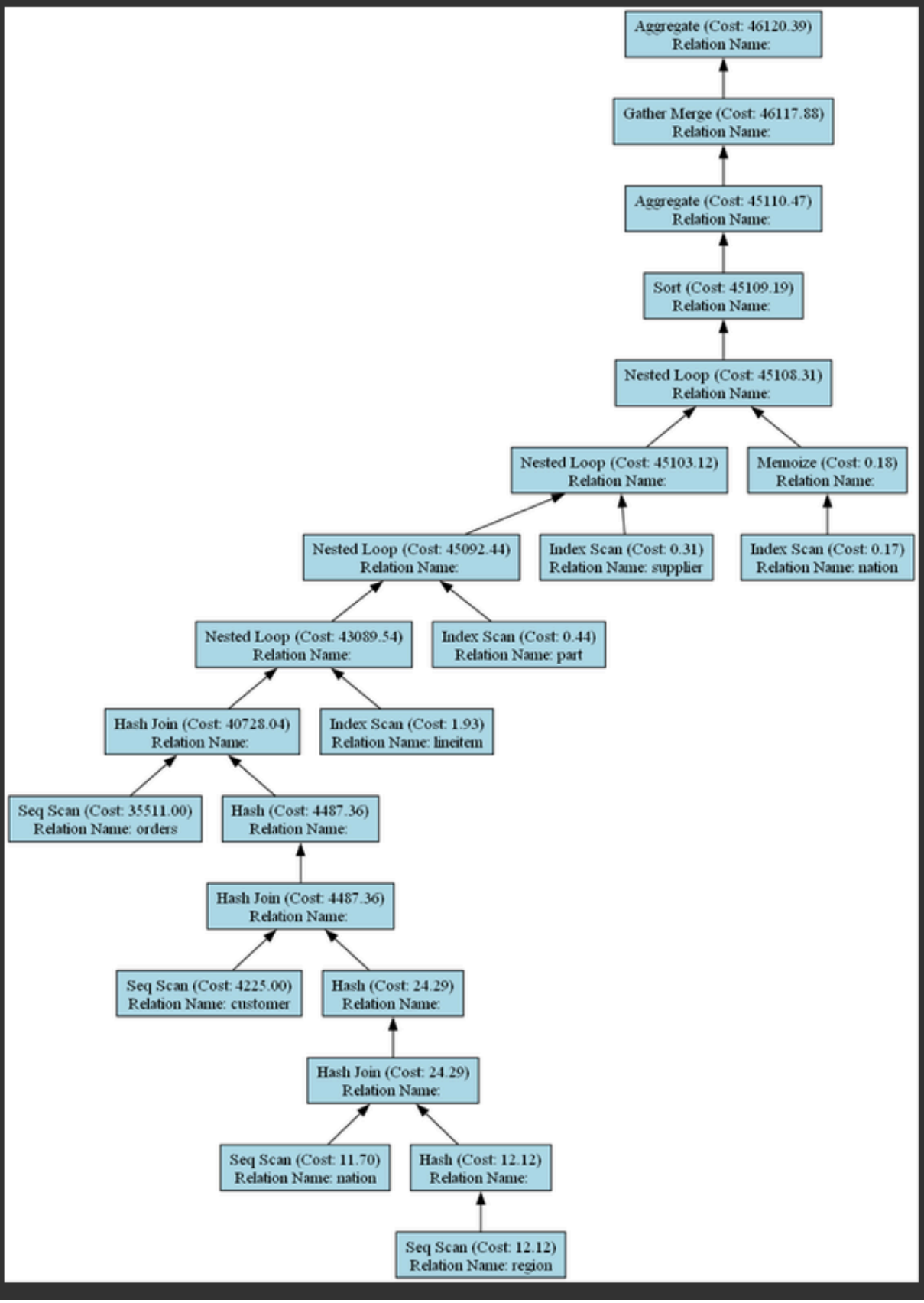
With the use of the Buffers in the Explain command, we retrieve the information on how PostgreSQL interacts with the disk and buffers.

7 Queries Output

We choose 3 random queries to report on our Analysis for Section 8 analysis. To view the output of more queries, please utilise our program to test out the pre-defined queries or create your own custom query.

Query 1 (11)	SELECT o_year, SUM(CASE WHEN nation = 'BRAZIL' THEN volume ELSE 0 END) / SUM(volume) as mkt_share FROM (SELECT DATE_PART('YEAR',o_orderdate) as o_year, l_extendedprice * (1 - l_discount) as volume, n2.n_name as nation FROM part, supplier, lineitem, orders, customer, nation n1, nation n2, region WHERE p_partkey = l_partkey AND s_suppkey = l_suppkey AND l_orderkey = o_orderkey AND o_custkey = c_custkey AND c_nationkey = n1.n_nationkey AND n1.n_regionkey = r_regionkey AND r_name = 'AMERICA' AND s_nationkey = n2.n_nationkey AND o_orderdate between '1995-01-01' and '1996-12-31' AND p_type = 'ECONOMY ANODIZED STEEL' AND s_acctbal > 10 AND l_extendedprice > 100) as all_nations GROUP BY o_year ORDER BY o_year;
QEP Diagram for Query 1	

Query Execution Plan



QEP Explanations for Query 1

Query Plan Explanations

Step 1: An Index Scan is performed on the relation nation. The scan has the index condition (n_nationkey = supplier.s_nationkey).
This operation has an estimated startup cost of 0.14, an estimated total cost of 0.17, 1 planned rows of an estimated 108bytes. The actual startup time of the operation was 0.014, the actual total time was 0.014ms, with 1 actual rows and 75 actual loops. The planner estimates a worse case scenario with more rows. There were 150 shared block hits(cache hits), and 2 shared block reads(cache misses).

Step 2: The results of lookups is stored in the cache with a Memoize Operation.
This operation has an estimated startup cost of 0.01, an estimated total cost of 0.18, 1 planned rows of an estimated 108bytes. The actual startup time of the operation was 0.002, the actual total time was 0.002ms, with 1 actual rows and 2363 actual loops. The planner estimates a worse case scenario with more rows. There were 150 shared block hits(cache hits), and 2 shared block reads(cache misses).

Step 3: An Index Scan is performed on the relation supplier. The scan has the index condition (s_suppkey = lineitem.l_suppkey). The filter (s_acctbal > '10'::numeric) is then applied to it and 0 rows are removed.
This operation has an estimated startup cost of 0.29, an estimated total cost of 0.31, 1 planned rows of an estimated 8bytes. The actual startup time of the operation was 0.008, the actual total time was 0.008ms, with 1 actual rows and 2603 actual loops. The planner estimates a worse case scenario with more rows. There were 6911 shared block hits(cache hits), and 900 shared block reads(cache misses).

Step 4: An Index Scan is performed on the relation part. The scan has the index condition (p_partkey = lineitem.l_partkey). The filter ((p_type)::text = 'ECONOMY ANODIZED STEEL'::text) is then applied to it and 1 rows are removed.
This operation has an estimated startup cost of 0.42, an estimated total cost of 0.44, 1 planned rows of an estimated 4bytes. The actual startup time of the operation was 0.003, the actual total time was 0.003ms, with 0 actual rows and 365091 actual loops. The planner estimates a worse case scenario with more rows. There were 1460365 shared block hits(cache hits), and 1 shared block reads(cache misses).

Step 5: An Index Scan is performed on the relation lineitem. The scan has the index condition (l_orderkey = orders.o_orderkey). The filter (l_extendedprice > '100'::numeric) is then applied to it and 0 rows are removed.
This operation has an estimated startup cost of 0.43, an estimated total cost of 1.93, 16 planned rows of an estimated 24bytes. The actual startup time of the operation was 0.011, the actual total time was 0.012ms, with 4 actual rows and 91179 actual loops. The planner estimates a worse case scenario with more rows. There were 287113 shared block hits(cache hits), and 88422 shared block reads(cache misses).

Step 6: A Sequential Scan is performed on the relation region. The filter (r_name = 'AMERICA'::bpchar) is then applied to it and 4 rows are removed.
This operation has an estimated startup cost of 0.00, an estimated total cost of 12.12, 1 planned rows of an estimated 4bytes. The cost could not be estimated because postgres estimates it to be extremely low as resources are reused from previous operations. The actual startup time of the operation was 0.036, the actual total time was 0.037ms, with 1 actual rows and 1 actual loops. The planner estimates a worse case scenario with more rows. There were 0 shared block hits(cache hits), and 1 shared block reads(cache misses).

Step 7: A Hash table is created from the previous relation, with 1024 buckets.
This operation has an estimated startup cost of 12.12, an estimated total cost of 12.12, 1 planned rows of an estimated 4bytes. The actual startup time of the operation was 0.044, the actual total time was 0.045ms, with 1 actual rows and 1 actual loops. The planner estimates a worse case scenario with more rows. There were 0 shared block hits(cache hits), and 1 shared block reads(cache misses).

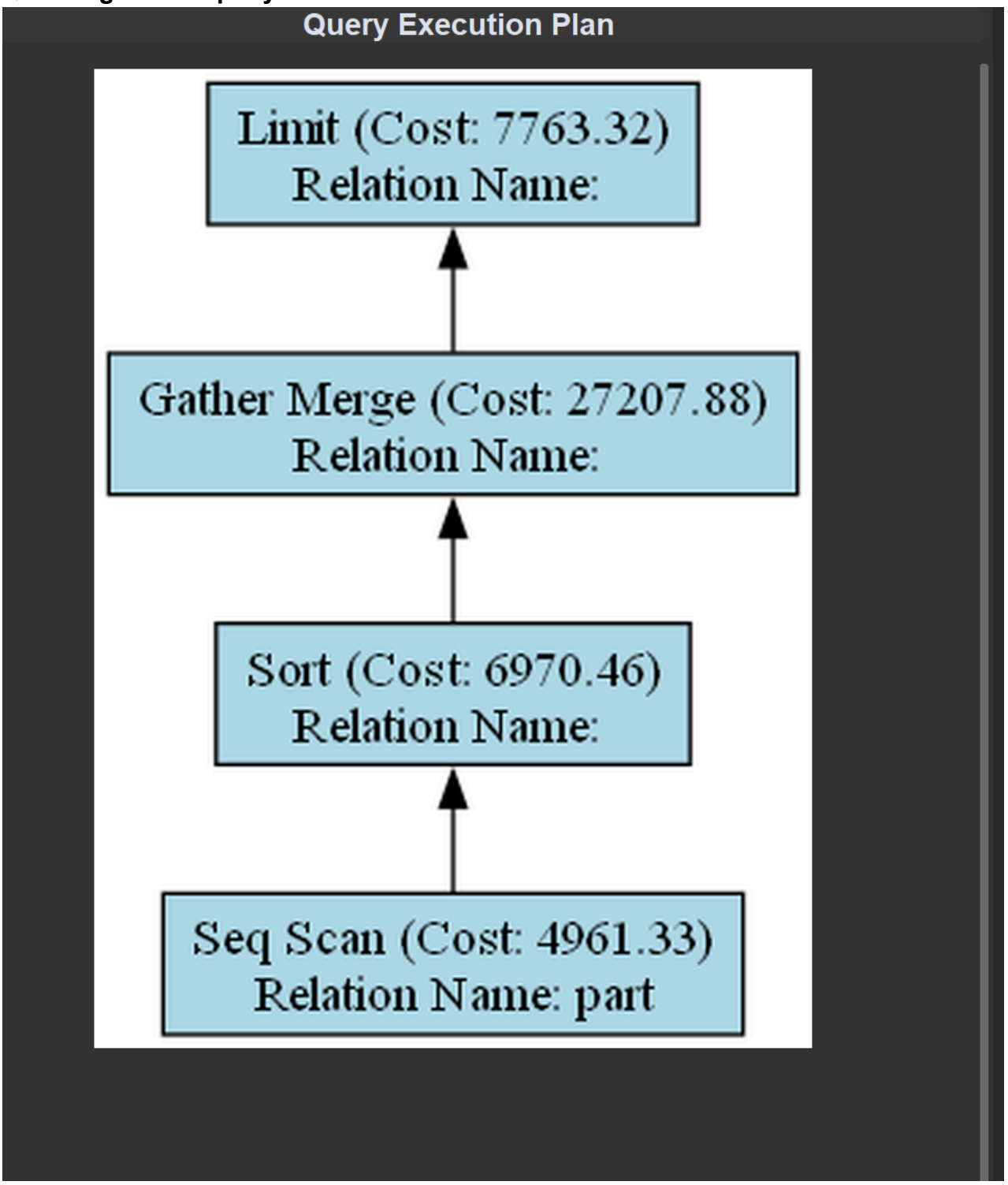
***Steps are skipped in this snippet**

Step 23: The rows are combined together with a Sorted Aggregate Operation.
This operation has an estimated startup cost of 0.00, an estimated total cost of 46120.39, 76 planned rows of an estimated 40bytes. The actual startup time of the operation was 1116.435, the actual total time was 1128.105ms, with 2 actual rows and 1 actual loops. The planner estimates a worse case scenario with more rows. There were 1754558 shared block hits(cache hits), and 119066 shared block reads(cache misses).

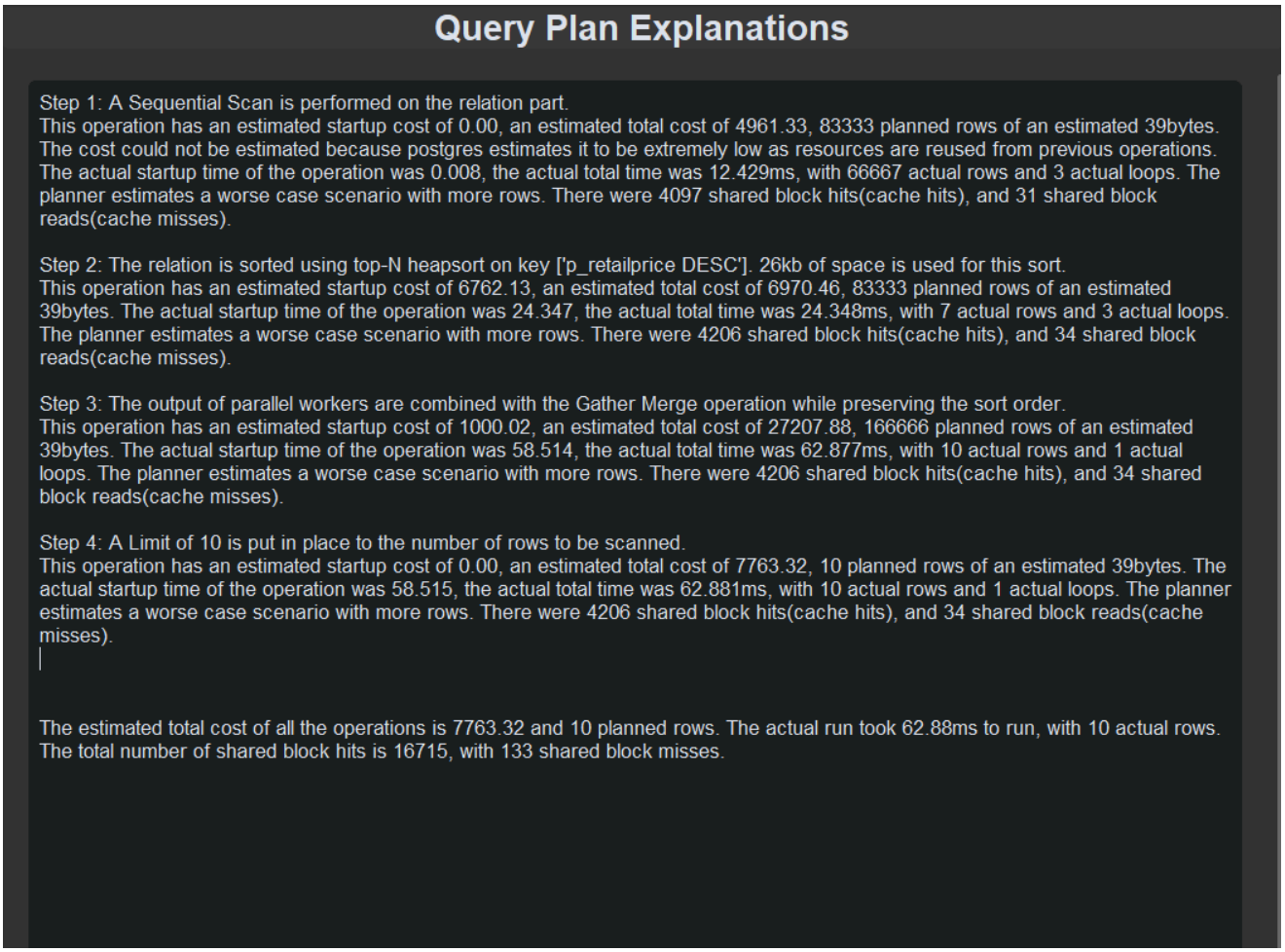
The estimated total cost of all the operations is 46120.39 and 76 planned rows. The actual run took 1128.11ms to run, with 2 actual rows. The total number of shared block hits is 14316470, with 1106721 shared block misses.

Query 2 (**4**)	SELECT p_name, p_retailprice FROM public.part ORDER BY p_retailprice DESC LIMIT 10;

QEP Diagram for query 2

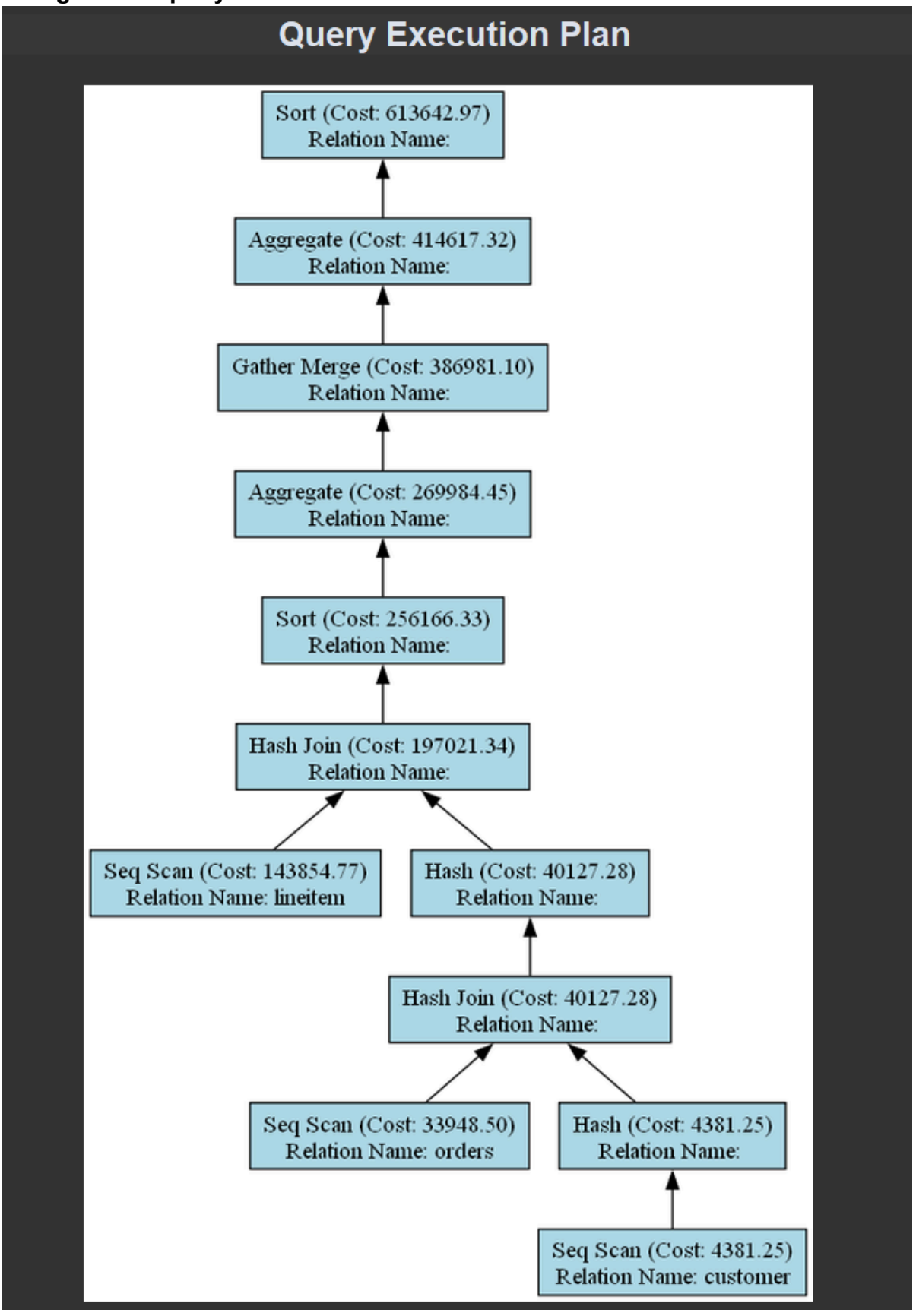


QEP Explanation for query 2



Query 3 (9)	SELECT l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority FROM customer, orders, lineitem WHERE c_mktsegment = 'BUILDING' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_totalprice > 10 AND l_extendedprice > 10 GROUP By l_orderkey, o_orderdate, o_shippriority ORDER BY revenue desc, o_orderdate;

QEP Diagram for query 3



QEP Explanation for query 3

Query Plan Explanations

Step 1: A Sequential Scan is performed on the relation customer. The filter (c_mktsegment = 'BUILDING'::bpchar) is then applied to it and 119858 rows are removed.
This operation has an estimated startup cost of 0.00, an estimated total cost of 4381.25, 12562 planned rows of an estimated 4bytes. The cost could not be estimated because postgres estimates it to be extremely low as resources are reused from previous operations. The actual startup time of the operation was 0.05, the actual total time was 40.344ms, with 30142 actual rows and 1 actual loops. More rows are used than what the planner planned for, this could be due to repeated scanning due to errors. There were 0 shared block hits(cache hits), and 3600 shared block reads(cache misses).

Step 2: A Hash table is created from the previous relation, with 32768 buckets.
This operation has an estimated startup cost of 4381.25, an estimated total cost of 4381.25, 12562 planned rows of an estimated 4bytes. The actual startup time of the operation was 15.5, the actual total time was 15.502ms, with 10047 actual rows and 3 actual loops. The planner estimates a worse case scenario with more rows. There were 0 shared block hits(cache hits), and 3600 shared block reads(cache misses).

Step 3: A Sequential Scan is performed on the relation orders. The filter (o_totalprice > '10'::numeric) is then applied to it and 0 rows are removed.
This operation has an estimated startup cost of 0.00, an estimated total cost of 33948.5, 624938 planned rows of an estimated 16bytes. The cost could not be estimated because postgres estimates it to be extremely low as resources are reused from previous operations. The actual startup time of the operation was 0.71, the actual total time was 155.222ms, with 500000 actual rows and 3 actual loops. The planner estimates a worse case scenario with more rows. There were 96 shared block hits(cache hits), and 26040 shared block reads(cache misses).

Step 4: The two relations are combined with a Inner Hash Join Operation on the condition (orders.o_custkey = customer.c_custkey).
This operation has an estimated startup cost of 157.02, an estimated total cost of 40127.28, 125612 planned rows of an estimated 12bytes. The actual startup time of the operation was 16.294, the actual total time was 286.92ms, with 101320 actual rows and 3 actual loops. The planner estimates a worse case scenario with more rows. There were 96 shared block hits(cache hits), and 29640 shared block reads(cache misses).

Step 5: A Hash table is created from the previous relation, with 524288 buckets.
This operation has an estimated startup cost of 35589.01, an estimated total cost of 40127.28, 125612 planned rows of an estimated 12bytes. The actual startup time of the operation was 315.92, the actual total time was 315.926ms, with 101320 actual rows and 3 actual loops. The planner estimates a worse case scenario with more rows. There were 96 shared block hits(cache hits), and 29640 shared block reads(cache misses).

Step 6: A Sequential Scan is performed on the relation lineitem. The filter (l_extendedprice > '10'::numeric) is then applied to it and 0 rows are removed.
This operation has an estimated startup cost of 0.00, an estimated total cost of 143854.77, 2500132 planned rows of an estimated 16bytes. The cost could not be estimated because postgres estimates it to be extremely low as resources are reused from previous operations. The actual startup time of the operation was 1.029, the actual total time was 649.936ms, with 2000405 actual rows and 3 actual loops. The planner estimates a worse case scenario with more rows. There were 4064 shared block hits(cache hits), and 108536 shared block reads(cache misses).

Step 7: The two relations are combined with a Inner Hash Join Operation on the condition (lineitem.l_orderkey = orders.o_orderkey).
This operation has an estimated startup cost of 1570.15, an estimated total cost of 197021.34, 502477 planned rows of an estimated 24bytes. The actual startup time of the operation was 318.086, the actual total time was 1310.653ms, with 404914 actual rows and 3 actual loops. The planner estimates a worse case scenario with more rows. There were 4181 shared block hits(cache hits), and 138177 shared block reads(cache misses).

***Steps are skipped in this snippet**

Step 12: The relation is sorted using external merge on key ["(sum((lineitem.l_extendedprice * ('1'::numeric - lineitem.l_discount)))) DESC", 'orders.o_orderdate']. 10096kb of space is used for this sort.
This operation has an estimated startup cost of 354717.95, an estimated total cost of 613642.97, 1205944 planned rows of an estimated 44bytes. The actual startup time of the operation was 2034.314, the actual total time was 2112.458ms, with 303959 actual rows and 1 actual loops. The planner estimates a worse case scenario with more rows. There were 4307 shared block hits(cache hits), and 138179 shared block reads(cache misses).

The estimated total cost of all the operations is 613642.97 and 1205944 planned rows. The actual run took 2112.46ms to run, with 303959 actual rows. The total number of shared block hits is 30068, with 1030128 shared block misses.

8 Observations & Analysis

Observation 1

The shared block hits (cache hits) will increase and shared block reads (cache miss) will decrease as we continuously run the same query.

Analysis:

1. PostgreSQL's shared buffer cache is designed to retain recently and frequently accessed data blocks in memory. During the first execution of a query, the necessary data may need to be retrieved from disk storage, resulting in cache misses. However, if the query is executed repeatedly and the data blocks remain in the buffer cache, PostgreSQL can quickly access this data from memory, avoiding disk reads and increasing cache hits.
2. Caching strategy is effective because accessing memory is much faster than disk I/O, significantly reducing the input/output operations needed for subsequent query executions. By keeping data in the cache after its initial retrieval, PostgreSQL will enhance the speed of future query executions.

Observation 2

In most queries, there were significant differences between the estimated row count and actual rows sorted.

Analysis:

1. This suggests that the planner did not anticipate the high selectivity of the sort operation, possibly due to an ordering condition that is highly selective
2. The discrepancy between planned and actual rows for the Gather Merge step might indicate that the planner's assumptions about the distribution of data across parallel workers need revisiting, though the actual impact on execution time is rather minimal.
3. This is a common sight in most DBMS systems, which suggests that the statistics may need to be updated to reflect the current state of the database more accurately.

Observation 3

Discrepancy between Actual costs vs Estimated costs.

Analysis:

1. Inaccurate Statistics: PostgreSQL utilizes statistical data regarding table sizes, row distributions, etc., for its estimations. The existing discrepancies suggest that the statistics for our tables are outdated, potentially leading the planner to incorrect conclusions.
2. Parallel Query Execution: When a query is executed in parallel, the complexity of the cost model increases, which can result in significant differences between the estimated and actual costs of the Query Execution Plan (QEP).
3. Cache State: If data is already loaded in memory from recent usage (cache hits), the actual execution cost will be reduced due to the minimized need for disk I/O. However, the planner's predictions may not always align with actual cache usage.

Node Types Efficiencies

Node Types and their efficiency or inefficiency under different conditions

1. **Efficiency of sequential scans:**

   For example query 3, a sequential scan on Lineitem. Despite the high number of actual rows, the actual total time is relatively low, which suggests the sequential scan was performed efficiently.

2. **Efficiency of buffer access:**

   For example query 3, a sequential scan on Lineitem, No cache hits were observed, and a very high number of block reads from disk (112600) were necessary. This points to a potential area for performance improvement, possibly by increasing the cache size or adding an index.

3. **Efficiency use of index scan**

For example, in query 3, there were multiple index scans Utilisation. This is a positive sign for query performance as index scans are generally faster than full table scans when searching for a small subset of rows.

4. **Efficiency of Memoize operation**

   Repeated Subqueries: It is particularly effective for repeated subqueries or function calls within the same query that return the same result every time they are called with the same arguments

5. **Efficiency in Parallelism**

   Parallel Query Execution: Since there is parallel execution involved (Gather Merge as seen in Query 3), the plan indicates that the database is leveraging multiple cores for the query. Ensure that the setting for max_parallel_workers_per_gather is optimised for the system's hardware and the database's typical workload.

   Choosing the Right Parallelism Mode: PostgreSQL offers different parallelism modes, such as "on" (default), "off," and "auto." By manually specifying the parallelism mode using `enable_parallel` or `force_parallel_mode` hints can be beneficial.

   Monitoring Parallel Query Performance: Efficient parallel query execution requires continuous monitoring and tuning. By utilising PostgreSQL's built-in monitoring tools, such as `pg_stat_statements` and `pg_stat_activity`, to track the performance of parallel queries, further optimisation can be done.

9 Limitations

Here are three potential limitations of the program described:

1. Dependency on PostgreSQL: The program is tightly coupled with PostgreSQL as the database backend. While this is suitable for applications specifically targeting PostgreSQL databases, it limits the program's compatibility with other database systems. If the program needs to support multiple database types, it would require significant modifications to handle different SQL dialects and database-specific features.

2. Complexity and Maintenance: Integrating multiple libraries like `psycopg2`, `customtkinter`, and `graphviz` increases the complexity of the program. Each library has its learning curve and maintenance requirements, potentially making the codebase harder to understand, maintain, and troubleshoot. Additionally, if any of these libraries receive updates or changes, it may require corresponding updates in the program to ensure compatibility and functionality.

3. Limited User Interface Flexibility: The use of `customtkinter` for displaying the query execution plan may limit the program's user interface (UI) flexibility. While `tkinter` provides a basic set of GUI components, `customtkinter` might have fewer resources and community support for customisation. This could restrict the UI's ability to adapt to different user needs or preferences. Similarly, `graphviz` for visualisation may have limitations in customisation, potentially constraining the depth or interactivity of the displayed execution plans.

10 Conclusion

Throughout the project, a comprehensive analysis of query execution plans was conducted using PostgreSQL's EXPLAIN ANALYZE tool. This deep dive into the query plans has yielded valuable insights into the performance characteristics and potential bottlenecks within PostgreSQL.

11 References

[1] “PSYCOPG2,” PyPI. [Online]. Available: https://pypi.org/project/psycopg2/.

[2]“Official documentation and tutorial | CustomTkinter.” https://customtkinter.tomschimansky.com/

[3] “PMW,” PyPI. [Online]. Available: https://pypi.org/project/Pmw/.