SC3020: Database System Principles

Project 2 Report

Application URL: https://woonyee28-what-if-query-plan-interface-iz2bjd.streamlit.app/

Name	**Matriculation Number**
Alex Khoo Shien How	U2220791B
Chua Ming Ru	U2140945D
Ng Woon Yee	U2120622C
Tay Zhi Xian	U2220099F
Yang Yichen	U2121540A

Table of Contents

1 Introduction

1.1 Project Overview

In SC3020 Project 2, our task is to analyze and compare optimized query execution plans (QEPs) with alternative query plans (AQPs) by enabling users to modify various components of QEP, such as join types, join orders and scan methods. This will enable us to explore the "what-if" scenarios and evaluate how different physical operators and configurations impact the estimated execution cost. Through the analysis, we will gain insights into the cost trade-offs and effectiveness of various query structures within relational databases.

To enhance usability, we also incorporate a Large Language Model (LLM) to generate an explanation for the "what-if" scenarios, enabling users to better understand the implications of their modifications. Additionally, this project includes a cloud-hosted PostgreSQL database for user testing, offering hands-on experience. At the same time, users are also provided with an option to connect our application to their own PostgreSQL database.

2 Project Architecture

2.1 Database

In our database, we utilized the TPC-H dataset, a decision support benchmark that consists of a suite of business-oriented ad-hoc queries and concurrent data modifications. The TPC-H dataset consists of eight separate and individual tables: part, supplier, partsupp, customer, lineitem, orders, nation and region. The dataset is initialized using the TPCH-dbgen tool, and we wrote a preprocessing script to clean the .csv files and auto populate it into our cloud-hosted database.

To connect our database and application, we use Psycopg, the popular PostgreSQL database adapter for Python. Additionally, we acknowledge that PostgreSQL is enabling parallel execution per executor node by default, which will lead to a parallel cost estimation instead of sequential cost estimation. Hence, we *SET max_parallel_workers_per_gather = 0 in the cloud server* to ensure accurate cost estimation.

2.1.1 Cloud

We hosted the TPC-H dataset in cloud to provide an option for users who do not have PostgreSQL installed to experience our application. We rented a cloud instance from Amazon Web Services, which has 2 CPU cores, 1GB RAM and 5GB Storage for PostgreSQL usage.

2.1.2 Local

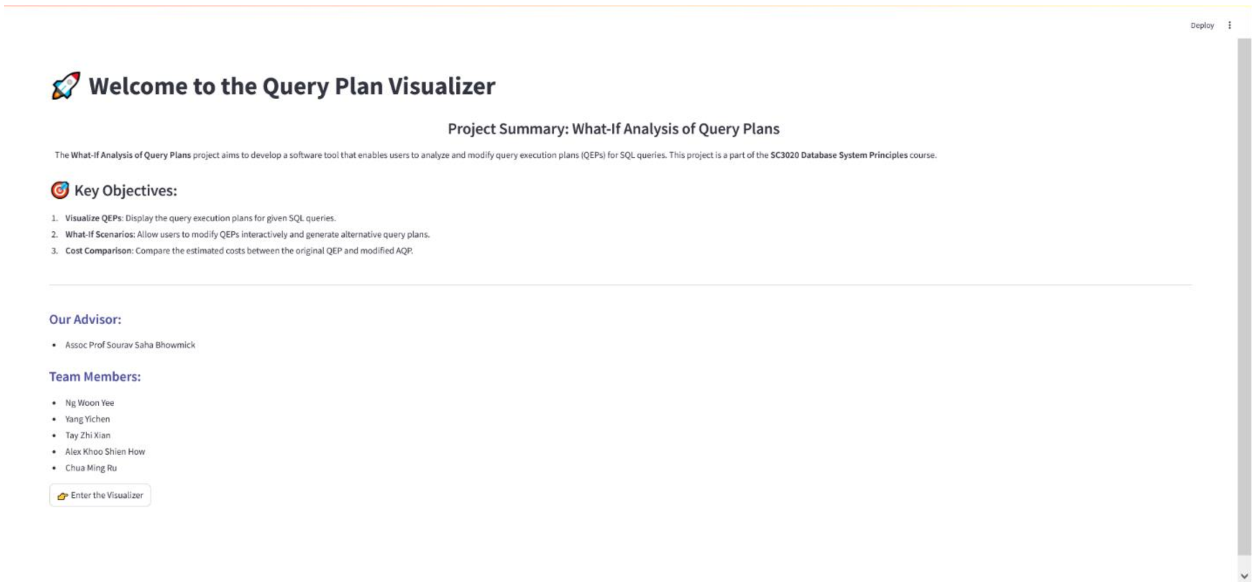
We also provide our users with the option to connect to their own PostgreSQL dataset, which needs their local database credentials like username, password, database name, local host IP address, port, and SSL Mode. We assume that the dataset has already been loaded into the database and our program does not provide any functionality to load database into Postgres.

2.2 Front-End

The front-end of our project was developed using Streamlit, an open-source Python framework primarily used for designs of data applications. Streamlit also facilitates deployment to a cloud platform, providing users with easier access to our application.

2.2.1 Welcome Page

At startup, the application first displays the welcome page, this interface provides a brief explanation of what our project involves and lists the members of the team together with our advisor.


Welcome to the Query Plan Visualizer

Project Summary: What-If Analysis of Query Plans

The **What-If Analysis of Query Plans** project aims to develop a software tool that enables users to analyze and modify query execution plans (QEPs) for SQL queries. This project is a part of the **SC3020 Database System Principles** course.

Key Objectives:

1. **Visualize QEPs**: Display the query execution plans for given SQL queries.
2. **What-If Scenarios:** Allow users to modify QEPs interactively and generate alternative query plans.
3. **Cost Comparison**: Compare the estimated costs between the original QEP and modified AQP.

Our Advisor:

- Assoc Prof Sourav Saha Bhowmick

Team Members:

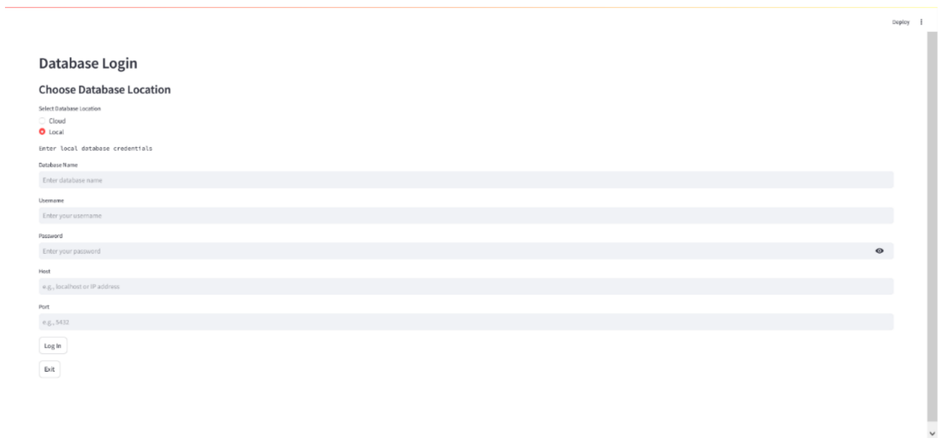
- Ng Woon Yee
- Yang Yichen
- Tay Zhi Xian
- Alex Khoo Shien How
- Chua Ming Ru

Fig 1: Screenshot of the welcome page

Clicking on "Enter the Visualiser" will bring the user to the login page.

2.2.2 Login Page

The login page provides the user with 2 key options for database connection: cloud and local. Cloud connection connects to our database hosted on a cloud platform, which includes the dataset we are working with for this project (TPC-H). Alternatively, local connection provides users with the option to manually configure the connection from their end, allowing the application to read different datasets the user presents. To do so, the user would need to input the following parameters: Database Name, Username, Password, Host and Port Number. Examples of the input parameters are also displayed in the input fields.


Database Login

Choose Database Location

Select Database Location

Cloud [✓ Local]

```
Enter local database credentials
```

Database Name: Enter database name

Username: Enter your username

Password: Enter your password

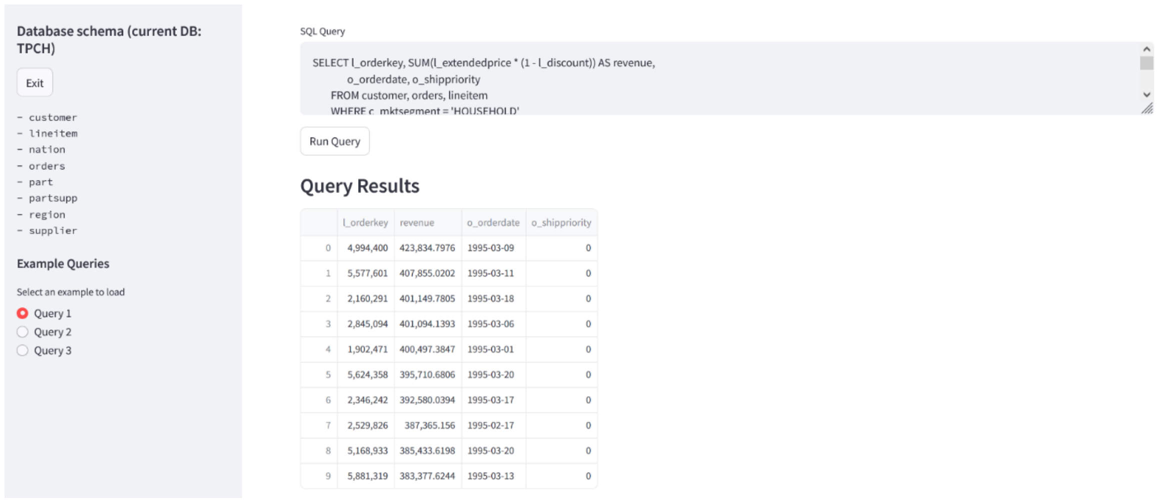
Host: e.g., localhost or IP address

Port: e.g., 5432

Fig 2: Screenshot of the database login page for local

2.2.3 Main Application Page

Upon successful connection to the database, the main application page is displayed. The side bar of the main application page displays the current database schema used by the application. The user may also select example queries for the application to execute. Depending on the example query set, the “SQL Query” input field is filled with different example queries to be run. Alternatively, the user may also manually input a **valid** SQL query to be run by the application. Selecting “Run Query” will execute the query in PostgreSQL and retrieve the output table of the query.



```
- customer
- lineitem
- nation
- orders
- part
- partsupp
- region
- supplier
```

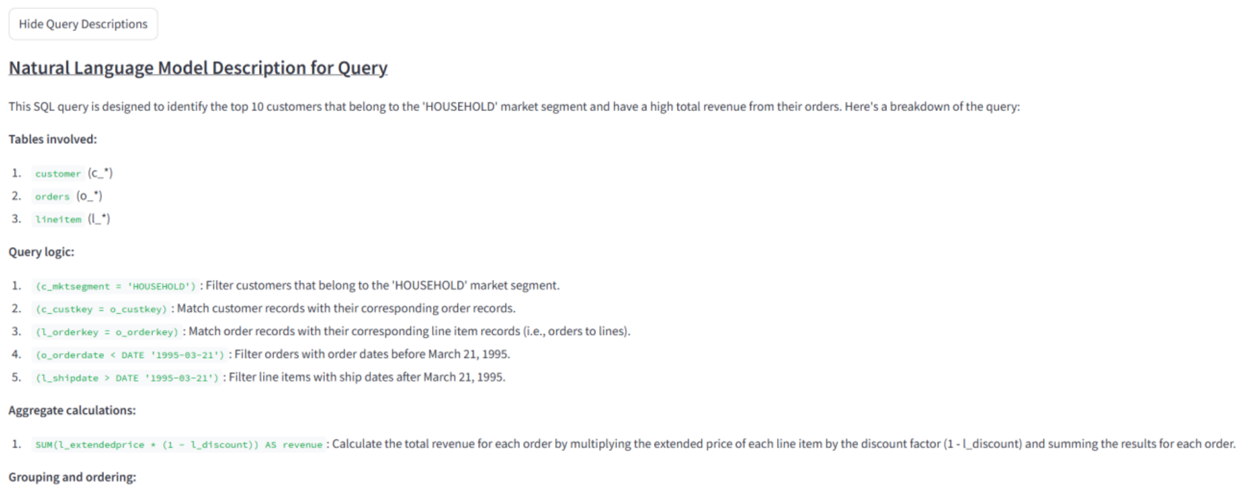
```
SELECT l_orderkey, SUM(l_extendedprice * (1 - l_discount)) AS revenue,
       o_orderdate, o_shippriority
   FROM customer, orders, lineitem
   WHERE c_mktsegment = 'HOUSEHOLD'
```

	l_orderkey	revenue	o_orderdate	o_shippriority
0	4,994,400	423,834.7976	1995-03-09	0
1	5,577,601	407,855.0202	1995-03-11	0
2	2,160,291	401,149.7805	1995-03-18	0
3	2,845,094	401,094.1393	1995-03-06	0
4	1,902,471	400,497.3847	1995-03-01	0
5	5,624,358	395,710.6806	1995-03-20	0
6	2,346,242	392,580.0394	1995-03-17	0
7	2,529,826	387,365.156	1995-02-17	0
8	5,168,933	385,433.6198	1995-03-20	0
9	5,881,319	383,377.6244	1995-03-13	0

Fig

Figure 3: Screenshot of SQL query and query result

Users can also toggle the query description button to read the natural language description of a query.



Natural Language Model Description for Query

This SQL query is designed to identify the top 10 customers that belong to the 'HOUSEHOLD' market segment and have a high total revenue from their orders. Here's a breakdown of the query:

Tables involved:

1. `customer` (c_*)
2. `orders` (o_*)
3. `lineitem` (l_*)

Query logic:

1. `(c_mktsegment = 'HOUSEHOLD')` : Filter customers that belong to the 'HOUSEHOLD' market segment.
2. `(c_custkey = o_custkey)` : Match customer records with their corresponding order records.
3. `(l_orderkey = o_orderkey)` : Match order records with their corresponding line item records (i.e., orders to lines).
4. `(o_orderdate < DATE '1995-03-21')` : Filter orders with order dates before March 21, 1995.
5. `(l_shipdate > DATE '1995-03-21')` : Filter line items with ship dates after March 21, 1995.

Aggregate calculations:

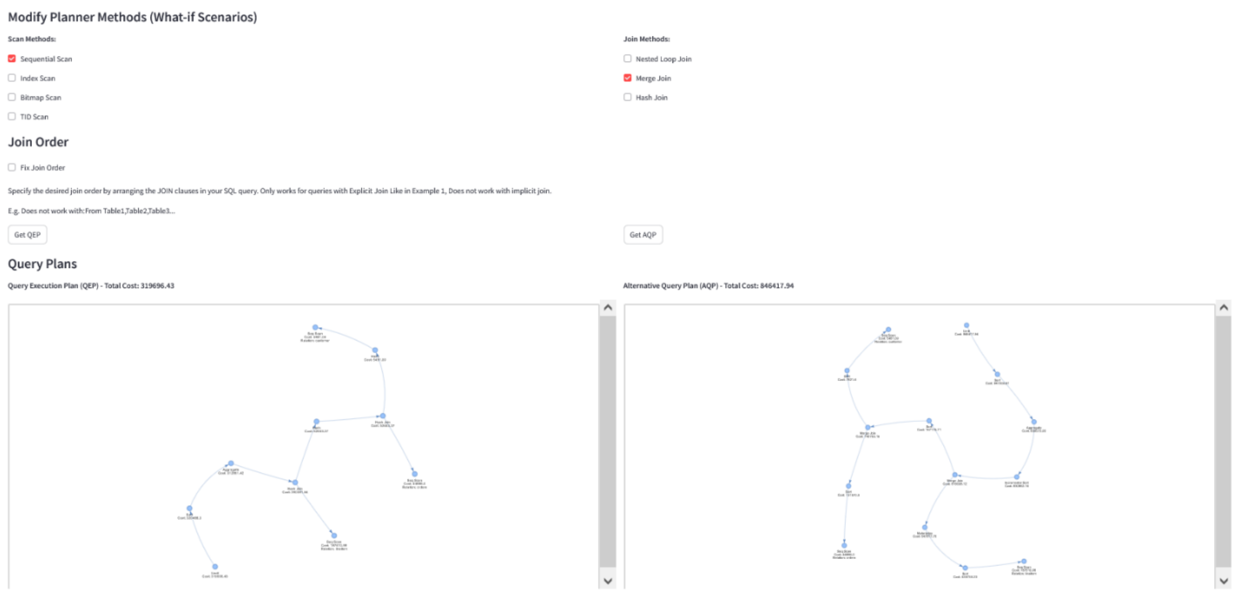
1. `SUM(l_extendedprice * (1 - l_discount)) AS revenue` : Calculate the total revenue for each order by multiplying the extended price of each line item by the discount factor (1 - l_discount) and summing the results for each order.

Grouping and ordering:

Fig 4: Screenshot of the description of query by Natural language model

To display the QEP utilised by PostgreSQL when executing the query, the user may select "Get QEP". The application will visualise the QEP in the form of a visual tree and display its cost.

To answer the project's "what-if" questions, the user may select the checkboxes under "Modify Planner Methods" which dictates the scan and join methods for the AQP. Selecting "Get AQP" visualises the AQP and displays the cost, similarly to the QEP. The output data for both QEP and AQP and placed beside each other for easier comparison.



Modify Planner Methods (What-if Scenarios)

Scan Methods:

- [✓ Sequential Scan]
- Index Scan
- Bitmap Scan
- TID Scan

Join Methods:

- Nested Loop Join
- [✓ Merge Join]
- Hash Join

Join Order

- Fix Join Order

Specify the desired join order by arranging the JOIN clauses in your SQL query. Only works for queries with Explicit Join Like in Example 1, Does not work with implicit join.

E.g. Does not work with:From Table1,Table2,Table3...

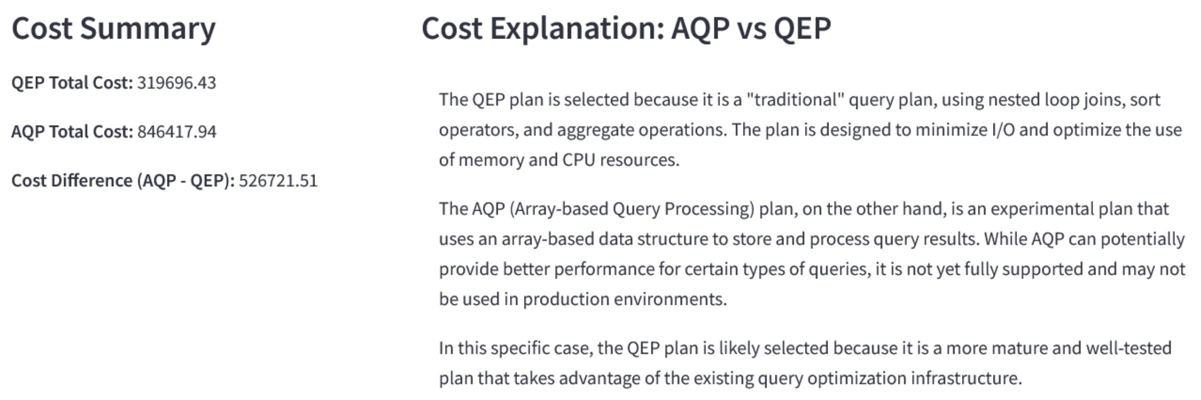
Query Plans

Query Execution Plan (QEP) - Total Cost: 319696.43

Alternative Query Plan (AQP) - Total Cost: 846417.94

Fig 5: Screenshot of the possible what-if scenarios and the graphs generated

A summary of the cost of both execution plans is displayed under “Cost Summary” which additionally includes the difference between the cost of the QEP and AQP. Under “Cost Explanation: AQP vs QEP”, advantages and disadvantages of both query plans are evaluated, with explanation as to why QEP is generally favoured.



Cost Summary

QEP Total Cost: 319696.43

AQP Total Cost: 846417.94

Cost Difference (AQP - QEP): 526721.51

Cost Explanation: AQP vs QEP

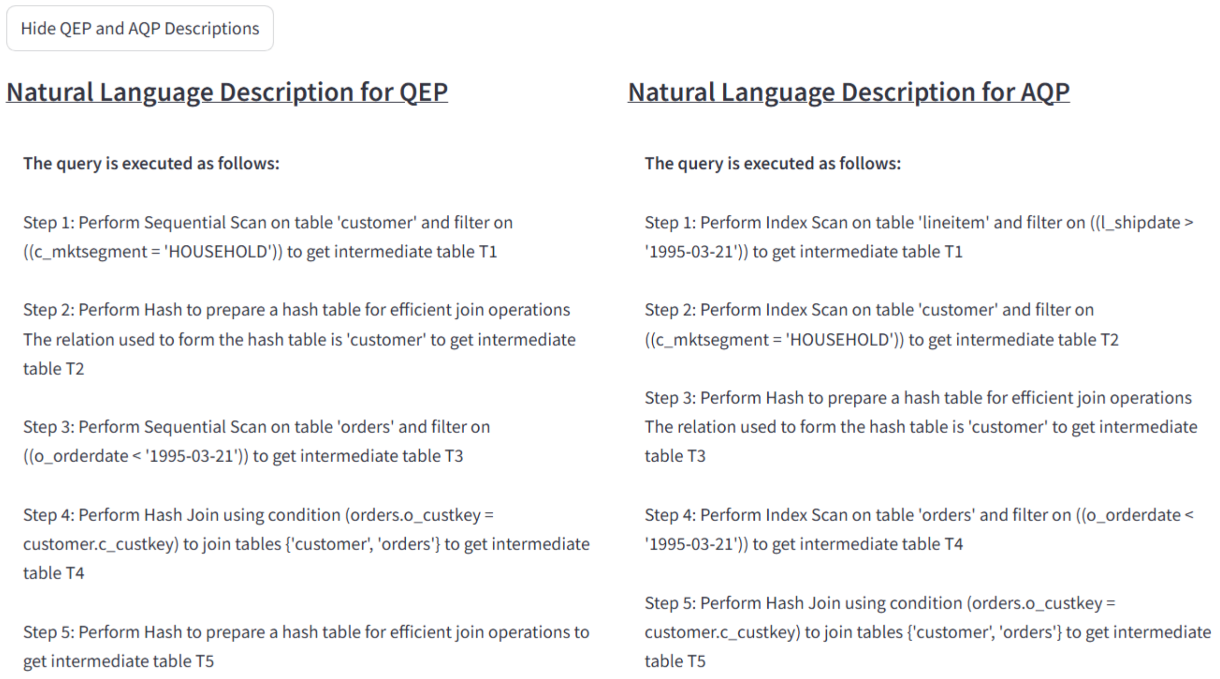
The QEP plan is selected because it is a "traditional" query plan, using nested loop joins, sort operators, and aggregate operations. The plan is designed to minimize I/O and optimize the use of memory and CPU resources.

The AQP (Array-based Query Processing) plan, on the other hand, is an experimental plan that uses an array-based data structure to store and process query results. While AQP can potentially provide better performance for certain types of queries, it is not yet fully supported and may not be used in production environments.

In this specific case, the QEP plan is likely selected because it is a more mature and well-tested plan that takes advantage of the existing query optimization infrastructure.

Fig 6: Screenshot of cost explanation

The LLM generation output of cost explanation for **both plans** is refreshed whenever the AQP plan or QEP plan of a query is updated or modified, which is whenever AQP button or QEP button is pressed by the user.



Natural Language Description for QEP

The query is executed as follows:

Step 1: Perform Sequential Scan on table 'customer' and filter on ((c_mktsegment = 'HOUSEHOLD')) to get intermediate table T1

Step 2: Perform Hash to prepare a hash table for efficient join operations The relation used to form the hash table is 'customer' to get intermediate table T2

Step 3: Perform Sequential Scan on table 'orders' and filter on ((o_orderdate < '1995-03-21')) to get intermediate table T3

Step 4: Perform Hash Join using condition (orders.o_custkey = customer.c_custkey) to join tables {'customer', 'orders'} to get intermediate table T4

Step 5: Perform Hash to prepare a hash table for efficient join operations to get intermediate table T5

Natural Language Description for AQP

The query is executed as follows:

Step 1: Perform Index Scan on table 'lineitem' and filter on ((l_shipdate > '1995-03-21')) to get intermediate table T1

Step 2: Perform Index Scan on table 'customer' and filter on ((c_mktsegment = 'HOUSEHOLD')) to get intermediate table T2

Step 3: Perform Hash to prepare a hash table for efficient join operations The relation used to form the hash table is 'customer' to get intermediate table T3

Step 4: Perform Index Scan on table 'orders' and filter on ((o_orderdate < '1995-03-21')) to get intermediate table T4

Step 5: Perform Hash Join using condition (orders.o_custkey = customer.c_custkey) to join tables {'customer', 'orders'} to get intermediate table T5

Fig 7: Screenshot of the natural language description for a QEP and AQP

By toggling ***QEP and AQP explanation button***, users can view the natural language steps explanation for both Query Explanation Plan and Alternative Query Plan. The **natural language steps explanation for both plans are refreshed** whenever the button is toggled by updating the content of session state variables such as st.session_state.last_aqp_plan.

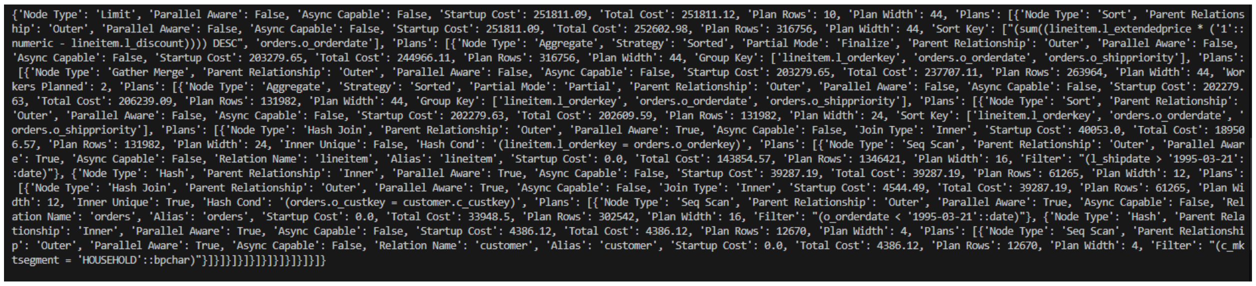
3 Preprocessing

In this section, we will explain the functions provided in preprocessing.py.

3.1 Query Plan Visualization

When the "Get QEP" or "Get AQP" button is clicked, the corresponding function (get_qep or get_aqp) is triggered. This function executes the SQL query against the database and retrieves a query execution plan (QEP) or an alternative query plan (AQP) in JSON format. This JSON contains nested nodes, each representing a step in the query execution process. Each node includes attributes such as:

- **Node Type**: The type of operation (e.g., Seq Scan, Hash Join, Aggregate).
- **Total Cost**: The estimated cost associated with this operation.
- **Relation Name**: The specific table (or relation) involved in the operation
- **Index Name**: The name of the index being used

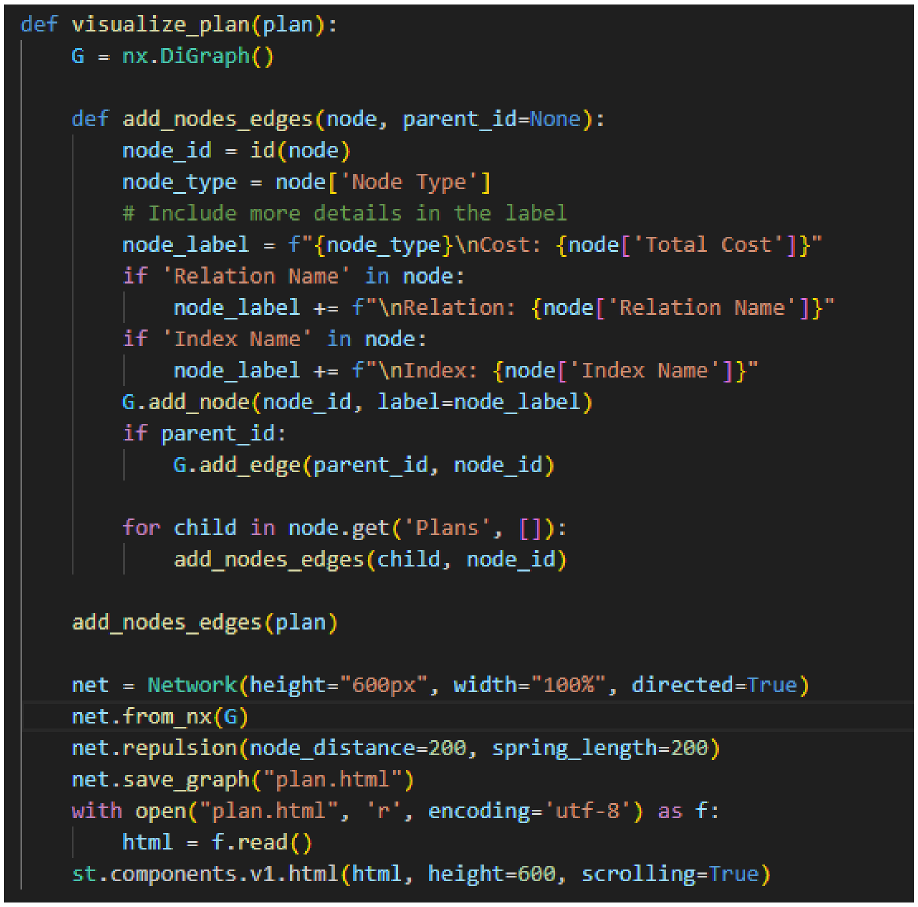


```
{'Node Type': 'Limit', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 251811.09, 'Total Cost': 251811.12, 'Plan Rows': 10, 'Plan Width': 44, 'Plans': [{'Node Type': 'Sort', 'Parent Relations
hip': 'Outer', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 251811.09, 'Total Cost': 252602.98, 'Plan Rows': 316756, 'Plan Width': 44, 'Sort Key': ["(sum((lineitem.l_extendedprice * ('1'::
numeric - lineitem.l_discount)))) DESC", 'orders.o_orderdate'], 'Plans': [{'Node Type': 'Aggregate', 'Strategy': 'Sorted', 'Partial Mode': 'Finalize', 'Parent Relationship': 'Outer', 'Parallel Aware': False,
'Async Capable': False, 'Startup Cost': 203279.65, 'Total Cost': 244966.11, 'Plan Rows': 316756, 'Plan Width': 44, 'Group Key': ['lineitem.l_orderkey', 'orders.o_orderdate', 'orders.o_shippriority'], 'Plans':
 [{'Node Type': 'Gather Merge', 'Parent Relationship': 'Outer', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 203279.65, 'Total Cost': 237707.11, 'Plan Rows': 263964, 'Plan Width': 44, 'Wor
kers Planned': 2, 'Plans': [{'Node Type': 'Aggregate', 'Strategy': 'Sorted', 'Partial Mode': 'Partial', 'Parent Relationship': 'Outer', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 202279.
63, 'Total Cost': 206239.09, 'Plan Rows': 131982, 'Plan Width': 44, 'Group Key': ['lineitem.l_orderkey', 'orders.o_orderdate', 'orders.o_shippriority'], 'Plans': [{'Node Type': 'Sort', 'Parent Relationship':
'Outer', 'Parallel Aware': False, 'Async Capable': False, 'Startup Cost': 202279.63, 'Total Cost': 202609.59, 'Plan Rows': 131982, 'Plan Width': 24, 'Sort Key': ['lineitem.l_orderkey', 'orders.o_orderdate', '
orders.o_shippriority'], 'Plans': [{'Node Type': 'Hash Join', 'Parent Relationship': 'Outer', 'Parallel Aware': True, 'Async Capable': False, 'Join Type': 'Inner', 'Startup Cost': 40053.0, 'Total Cost': 18950
6.57, 'Plan Rows': 131982, 'Plan Width': 24, 'Inner Unique': False, 'Hash Cond': '(lineitem.l_orderkey = orders.o_orderkey)', 'Plans': [{'Node Type': 'Seq Scan', 'Parent Relationship': 'Outer', 'Parallel Awar
e': True, 'Async Capable': False, 'Relation Name': 'lineitem', 'Alias': 'lineitem', 'Startup Cost': 0.0, 'Total Cost': 143854.57, 'Plan Rows': 1346421, 'Plan Width': 16, 'Filter': "(l_shipdate > '1995-03-21':
:date)"}, {'Node Type': 'Hash', 'Parent Relationship': 'Inner', 'Parallel Aware': True, 'Async Capable': False, 'Startup Cost': 39287.19, 'Total Cost': 39287.19, 'Plan Rows': 61265, 'Plan Width': 12, 'Plans':
 [{'Node Type': 'Hash Join', 'Parent Relationship': 'Outer', 'Parallel Aware': True, 'Async Capable': False, 'Join Type': 'Inner', 'Startup Cost': 4544.49, 'Total Cost': 39287.19, 'Plan Rows': 61265, 'Plan Wi
dth': 12, 'Inner Unique': True, 'Hash Cond': '(orders.o_custkey = customer.c_custkey)', 'Plans': [{'Node Type': 'Seq Scan', 'Parent Relationship': 'Outer', 'Parallel Aware': True, 'Async Capable': False, 'Rel
ation Name': 'orders', 'Alias': 'orders', 'Startup Cost': 0.0, 'Total Cost': 33948.5, 'Plan Rows': 302542, 'Plan Width': 16, 'Filter': "(o_orderdate < '1995-03-21'::date)"}, {'Node Type': 'Hash', 'Parent Rela
tionship': 'Inner', 'Parallel Aware': True, 'Async Capable': False, 'Startup Cost': 4386.12, 'Total Cost': 4386.12, 'Plan Rows': 12670, 'Plan Width': 4, 'Plans': [{'Node Type': 'Seq Scan', 'Parent Relationshi
p': 'Outer', 'Parallel Aware': True, 'Async Capable': False, 'Relation Name': 'customer', 'Alias': 'customer', 'Startup Cost': 0.0, 'Total Cost': 4386.12, 'Plan Rows': 12670, 'Plan Width': 4, 'Filter': "(c_mk
tsegment = 'HOUSEHOLD'::bpchar)"}]}]}]}]}]}]}]}]}]}]}
```

Fig 8: Example of query plan in JSON format

After obtaining the JSON plan, it is passed to the visualize_plan function, which uses the NetworkX library to create a directed graph (nx.DiGraph()). Within this function, the add_nodes_edges helper function recursively processes each node in the JSON structure. For each node, it extracts key information such as Node Type, Total Cost, and Relation Name, and creates nodes in the NetworkX graph. The function then recursively adds directed edges between each node and its child nodes, constructing the complete hierarchical structure of the query plan.

To enhance the visualization, the Pyvis Network class is used to format the graph, applying layout and spacing adjustments for a more readable and interactive display. The resulting graph is saved as an HTML file and then embedded into the Streamlit application, where it is displayed on the main page. This interactive visualization allows users to explore and understand the structure and cost of each step in the query execution plan.

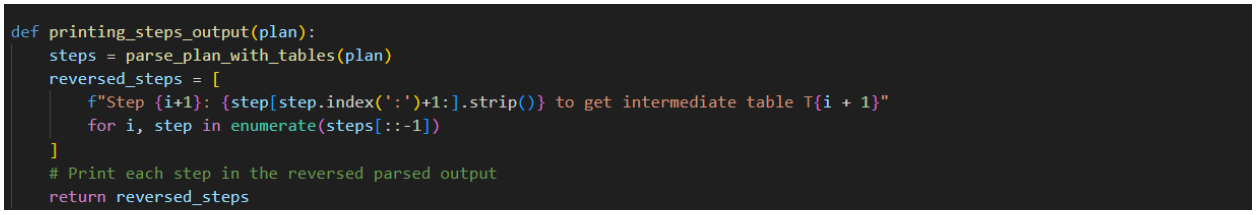


```python
def visualize_plan(plan):
    G = nx.DiGraph()

    def add_nodes_edges(node, parent_id=None):
        node_id = id(node)
        node_type = node['Node Type']
        # Include more details in the label
        node_label = f"{node_type}\nCost: {node['Total Cost']}"
        if 'Relation Name' in node:
            node_label += f"\nRelation: {node['Relation Name']}"
        if 'Index Name' in node:
            node_label += f"\nIndex: {node['Index Name']}"
        G.add_node(node_id, label=node_label)
        if parent_id:
            G.add_edge(parent_id, node_id)

        for child in node.get('Plans', []):
            add_nodes_edges(child, node_id)

    add_nodes_edges(plan)

    net = Network(height="600px", width="100%", directed=True)
    net.from_nx(G)
    net.repulsion(node_distance=200, spring_length=200)
    net.save_graph("plan.html")
    with open("plan.html", 'r', encoding='utf-8') as f:
        html = f.read()
    st.components.v1.html(html, height=600, scrolling=True)
```

Fig 9: Code for the visualize_plan function found in preprocessing.py

3.2 Natural Language Explanation

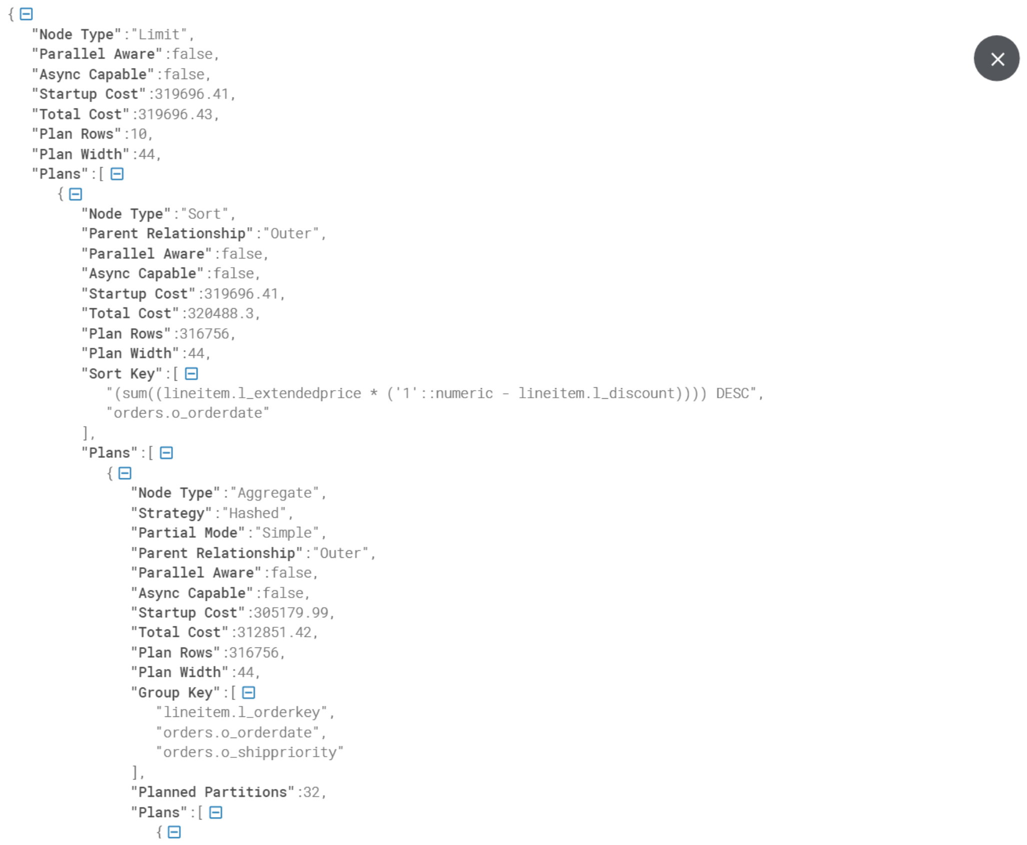
Using the JSON file generated, text preprocessing steps are applied to describe the steps involved in the Query Explanation Plan (QEP) and Alternative Query Plan (AQP) by analysing the nodes in the JSON format plan. In this project, custom natural language explanations have been made for scan operations e.g. ["Sequential Scan", "Bitmap Heap Scan", "Index Scan", "Index Only Scan", "Bitmap Index Scan", "Tid Scan", "Sample Scan"], join operations e.g. ["Nested Loop", "Merge Join", "Hash Join"], sort operations e.g. [“Sort”, "Incremental Sort"], aggregate operations e.g. ["Aggregate", "Limit"] , and others e.g. ["Hash" and "Memoize"] using the variable node_types in the plan.



```python
def printing_steps_output(plan):
    steps = parse_plan_with_tables(plan)
    reversed_steps = [
        f"Step {i+1}: {step[step.index(':')+1:].strip()} to get intermediate table T{i + 1}"
        for i, step in enumerate(steps[::-1])
    ]
    # Print each step in the reversed parsed output
    return reversed_steps
```

Fig 10: Code for printing steps for natural language description

Considering the Query Explanation Plan and Alternative Query Plan steps generated by Postgrad SQL are in a bottom-up fashion, the **printing_steps_output** function will align the query plan steps in a top-down manner for more intuitive understanding of the query execution flow.



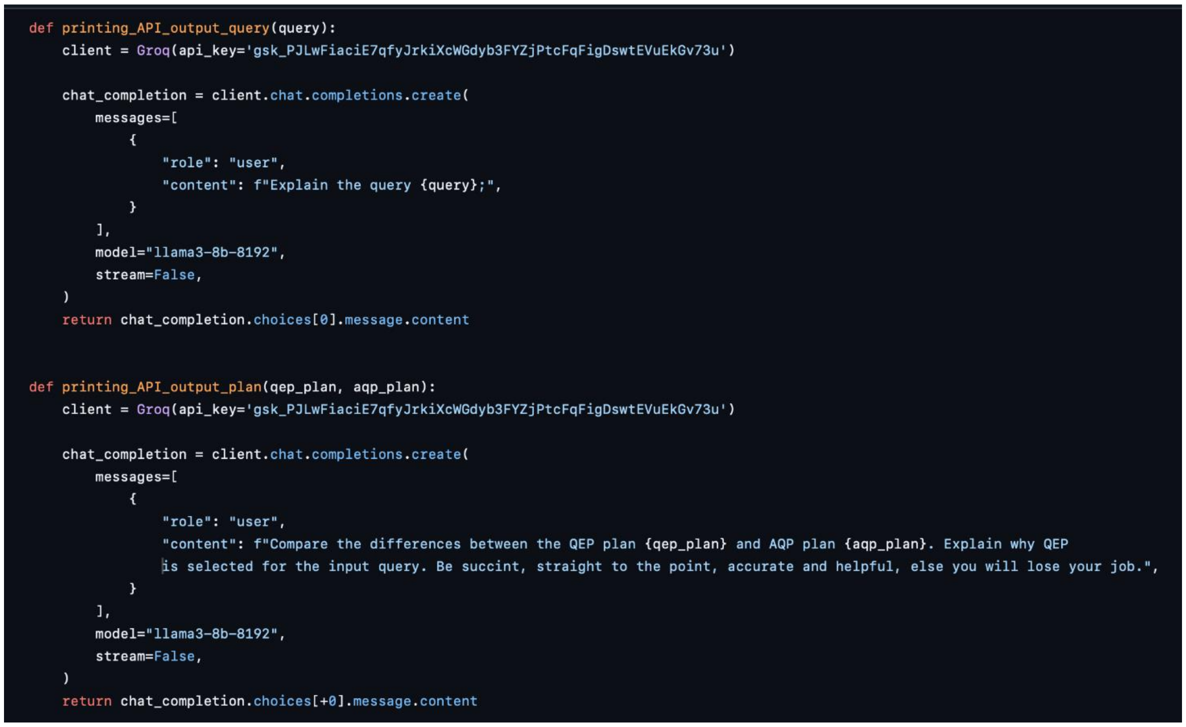
```
{
  "Node Type":"Limit",
  "Parallel Aware":false,
  "Async Capable":false,
  "Startup Cost":319696.41,
  "Total Cost":319696.43,
  "Plan Rows":10,
  "Plan Width":44,
  "Plans":[
    {
      "Node Type":"Sort",
      "Parent Relationship":"Outer",
      "Parallel Aware":false,
      "Async Capable":false,
      "Startup Cost":319696.41,
      "Total Cost":320488.3,
      "Plan Rows":316756,
      "Plan Width":44,
      "Sort Key":[
        "(sum((lineitem.l_extendedprice * ('1'::numeric - lineitem.l_discount)))) DESC",
        "orders.o_orderdate"
      ],
      "Plans":[
        {
          "Node Type":"Aggregate",
          "Strategy":"Hashed",
          "Partial Mode":"Simple",
          "Parent Relationship":"Outer",
          "Parallel Aware":false,
          "Async Capable":false,
          "Startup Cost":305179.99,
          "Total Cost":312851.42,
          "Plan Rows":316756,
          "Plan Width":44,
          "Group Key":[
            "lineitem.l_orderkey",
            "orders.o_orderdate",
            "orders.o_shippriority"
          ],
          "Planned Partitions":32,
          "Plans":[
            {
```

Fig 11: JSON file format of a query plan

3.3 Using Large Language Model for cost explanation

A Large Language Model (LLM) llama3-8b-8192 model has been deployed to generate natural language explanation for the SQL query and the cost comparison between AQP plan and QEP plan such that more detailed insights of the query and comparable explanations of both plans can be generated for the user. This is done by parsing the query plan (in JSON format) to the LLM to get the LLM to output the reason for the cost difference.

This is done as Large Language Models can interpret query plans and large volume of queries more effectively in the context of query optimization. Furthermore, Large Language Models can also automate the analysis of queries and query plans by considering the broader context which would otherwise require manual interpretation, strong domain knowledge and sophisticated text parsing techniques due to the different contexts of varied queries with relation to different databases incorporated.



```python
def printing_API_output_query(query):
    client = Groq(api_key='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')

    chat_completion = client.chat.completions.create(
        messages=[
            {
                "role": "user",
                "content": f"Explain the query {query};",
            }
        ],
        model="llama3-8b-8192",
        stream=False,
    )
    return chat_completion.choices[0].message.content


def printing_API_output_plan(qep_plan, aqp_plan):
    client = Groq(api_key='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')

    chat_completion = client.chat.completions.create(
        messages=[
            {
                "role": "user",
                "content": f"Compare the differences between the QEP plan {qep_plan} and AQP plan {aqp_plan}. Explain why QEP
                is selected for the input query. Be succint, straight to the point, accurate and helpful, else you will lose your job.",
            }
        ],
        model="llama3-8b-8192",
        stream=False,
    )
    return chat_completion.choices[+0].message.content
```

Fig 12: Code for printing output

4 What If

This section describes the functions provided in whatif.py.

4.1 Algorithms

In the whatif.py file, there exists 2 functions – to generate the optimal query plan (QEP) and to generate the alternate query plan (AQP).

To generate the QEP (in the **get_qep** function), we will append “EXPLAIN (FORMAT JSON) ” to the front of the SQL query. The function will then send the query to Postgres SQL for Postgres to generate the query execution plan in JSON format and the estimated total cost needed to execute the plan. We will store these 2 results to be used by preprocessing.py to generate the graphs.

To generate the AQP (in the **get_aqp** function), we will check each planner setting and turn setting on if the checkbox is ticked, else we turn the setting off. For example, if the user chose to disable merge_sort, the algorithm would tell Postgres to “SET merge_sort TO 'off’;” In the case where ‘fixed join order’ is chosen, the algorithm will tell Postgres to “SET join_collapse_limit TO 1;" to force explicit join order. After enabling/disabling the settings chosen by the user, we will append “EXPLAIN (FORMAT JSON) ” to the front of the SQL query to find the AQP and the total cost of the plan. Reset of settings will be performed after the SQL query to ensure that subsequent queries in the same session will use the correct settings.

5 Limitations and Assumptions

Our model employs a simplified version of PostgreSQL's cost estimation formulas. PostgreSQL's query planner uses a complex and detailed cost model that accounts for numerous factors, including CPU costs, I/O costs, memory usage and parallelism overhead. In our model, the parameters: seq_page_cost, random_page_cost, cpu_tuple_cost, cpu_index_tuple_cost, cpu_operator_cost are set to the default configuration from PostgreSQL. Our model simplifies these factors, focusing on the primary components and using generalized formulas. This simplification means we may not capture the nuanced cost contributions of specific operations, leading to different total cost estimates compared to more comprehensive tools like Mocha and Lantern.

Our model cannot directly alter individual steps within the QEP to use alternative algorithms. PostgreSQL does not provide a mechanism to directly edit or replace specific operations within a QEP. For instance, we cannot change a sequential scan to an index scan at a particular step in the plan through direct modification. There is no built-in interface or API to manually adjust individual operators within the execution plan.

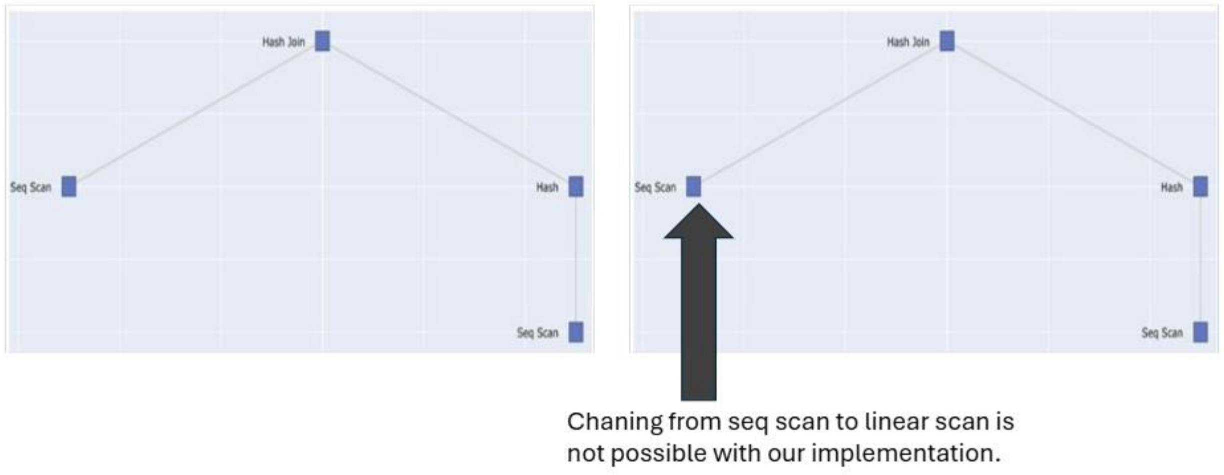
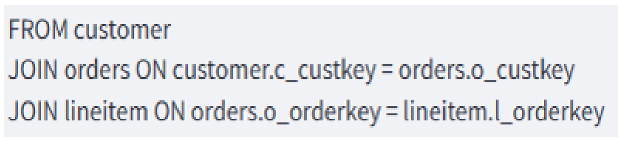

Hash Join
Seq Scan
Hash
Seq Scan
Hash Join
Seq Scan
Hash
Seq Scan
Chaning from seq scan to linear scan is not possible with our implementation.

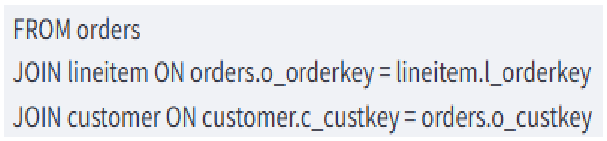
Fig 13: (left) shows a possible QEP and the right shows an impossible configuration.

Additionally, PostgreSQL's query planner and optimizer are designed to autonomously determine the most efficient execution plan based on available statistics, indexes, and configurations. Due to the limitations of PostgreSQL, we do not have precise control over specific plan steps. This restricts the depth of what-if analysis we can perform. In our model, we can only alter the parameters to include or exclude a particular algorithm entirely from the execution plan (If Index scan is set to off, the planner will not use index scan in any part of the AQP).

Next, the feature to change join order, implemented with the SET join_collapse_limit parameter, requires users to manually rewrite and restructure their queries with explicit JOIN clauses. This constraint arises from PostgreSQL's query planner, which relies on explicit joins for enforcing join orders, and there are no other built-in parameters that are available to influence the planner's decision on join order. With the current implementation, changing of join order is only possible with changing the order of explicit join clauses in the query. Due to the limitations of the PostgreSQL's built-in planner, our model lacks the ability to let users visually reorder joins or manipulate the execution plan through a more user-friendly graphical interface.



```
FROM customer
JOIN orders ON customer.c_custkey = orders.o_custkey
JOIN lineitem ON orders.o_orderkey = lineitem.l_orderkey
```



```
FROM orders
JOIN lineitem ON orders.o_orderkey = lineitem.l_orderkey
JOIN customer ON customer.c_custkey = orders.o_custkey
```

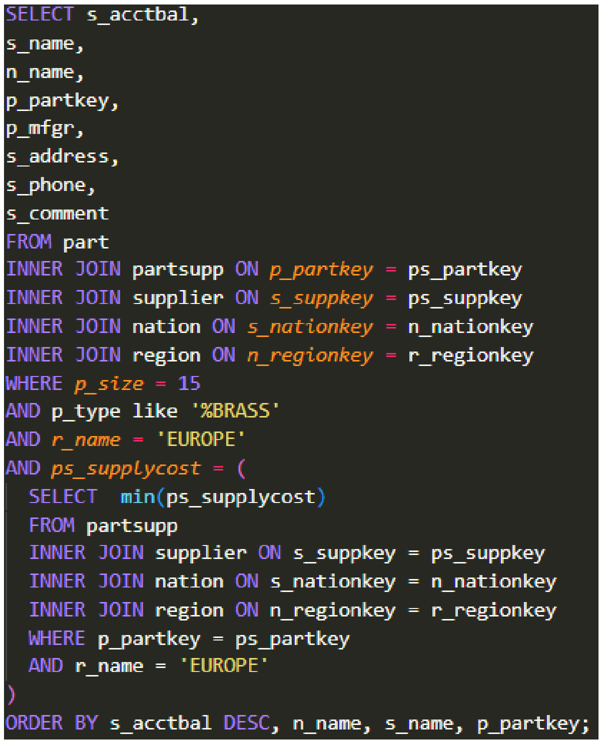
Fig 14 shows the changing of join order by reordering the input query.

The feature to be able to change join order by using the operation '**geqo_seed**' is not supported by our system as it is only applicable for large number of table joins (>=12), which our example queries do not use. Hence, it is not implemented as it is not useful in most SQL queries.

6 Example query and output

In this section, we will give an example of a query and the results shown in the User Interface (UI).

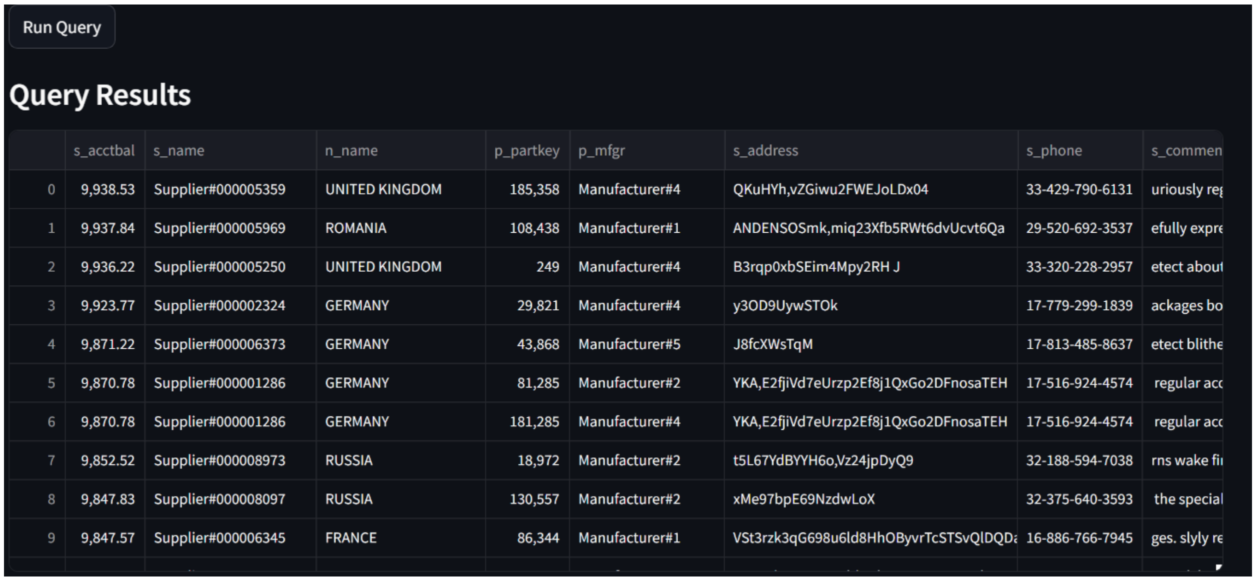
The query used will be the example query 2.



```sql
SELECT s_acctbal,
s_name,
n_name,
p_partkey,
p_mfgr,
s_address,
s_phone,
s_comment
FROM part
INNER JOIN partsupp ON p_partkey = ps_partkey
INNER JOIN supplier ON s_suppkey = ps_suppkey
INNER JOIN nation ON s_nationkey = n_nationkey
INNER JOIN region ON n_regionkey = r_regionkey
WHERE p_size = 15
AND p_type like '%BRASS'
AND r_name = 'EUROPE'
AND ps_supplycost = (
  SELECT  min(ps_supplycost)
  FROM partsupp
  INNER JOIN supplier ON s_suppkey = ps_suppkey
  INNER JOIN nation ON s_nationkey = n_nationkey
  INNER JOIN region ON n_regionkey = r_regionkey
  WHERE p_partkey = ps_partkey
  AND r_name = 'EUROPE'
)
ORDER BY s_acctbal DESC, n_name, s_name, p_partkey;
```

Fig 15: Example query 2

By pressing the run query button, the UI will display the output results.

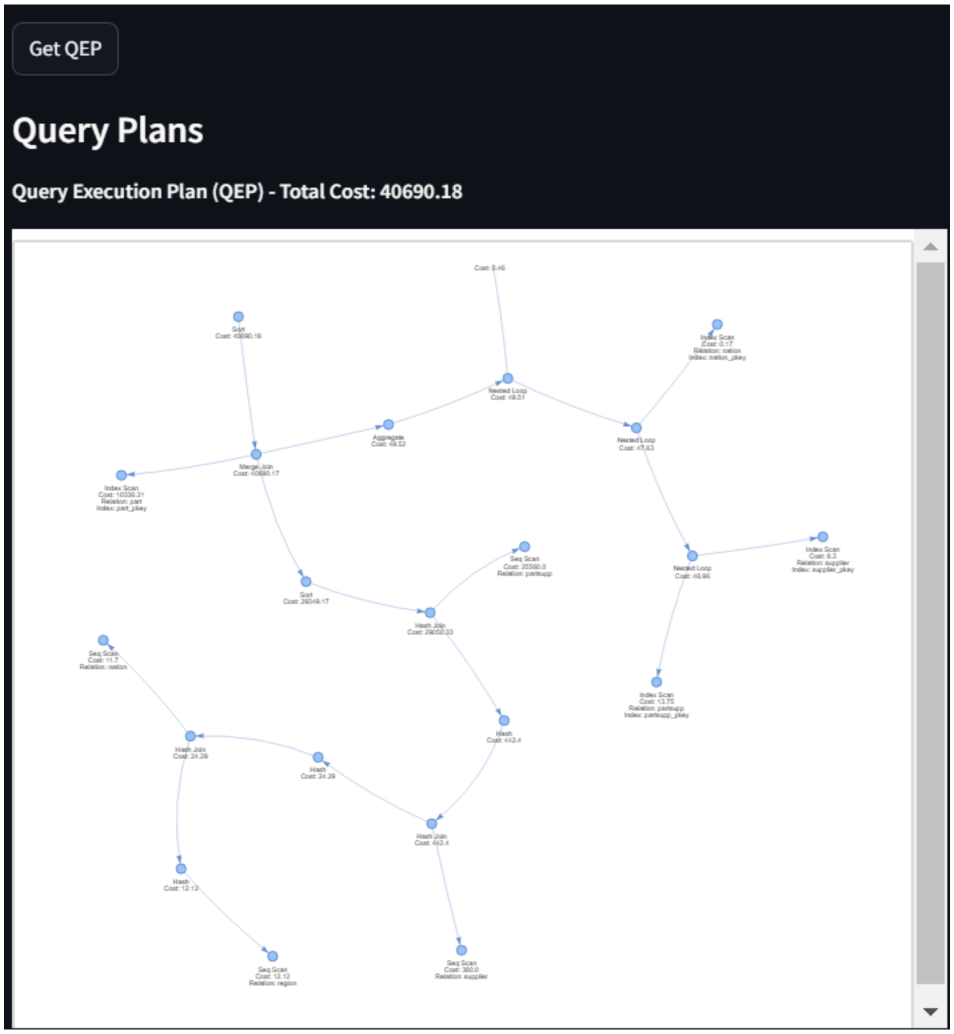


Query Results

	s_acctbal	s_name	n_name	p_partkey	p_mfgr	s_address	s_phone	s_commen
0	9,938.53	Supplier#000005359	UNITED KINGDOM	185,358	Manufacturer#4	QKuHYh,vZGiwu2FWEJoLDx04	33-429-790-6131	uriously reg
1	9,937.84	Supplier#000005969	ROMANIA	108,438	Manufacturer#1	ANDENSOSmk,miq23Xfb5RWt6dvUcvt6Qa	29-520-692-3537	efully expre
2	9,936.22	Supplier#000005250	UNITED KINGDOM	249	Manufacturer#4	B3rqp0xbSEim4Mpy2RH J	33-320-228-2957	etect about
3	9,923.77	Supplier#000002324	GERMANY	29,821	Manufacturer#4	y3OD9UywSTOk	17-779-299-1839	ackages bo
4	9,871.22	Supplier#000006373	GERMANY	43,868	Manufacturer#5	J8fcXWsTqM	17-813-485-8637	etect blithe
5	9,870.78	Supplier#000001286	GERMANY	81,285	Manufacturer#2	YKA,E2fjiVd7eUrzp2Ef8j1QxGo2DFnosaTEH	17-516-924-4574	regular acc
6	9,870.78	Supplier#000001286	GERMANY	181,285	Manufacturer#4	YKA,E2fjiVd7eUrzp2Ef8j1QxGo2DFnosaTEH	17-516-924-4574	regular acc
7	9,852.52	Supplier#000008973	RUSSIA	18,972	Manufacturer#2	t5L67YdBYYH6o,Vz24jpDyQ9	32-188-594-7038	rns wake fi
8	9,847.83	Supplier#000008097	RUSSIA	130,557	Manufacturer#2	xMe97bpE69NzdwLoX	32-375-640-3593	the special
9	9,847.57	Supplier#000006345	FRANCE	86,344	Manufacturer#1	VSt3rzk3qG698u6ld8HhOByvrTcSTSvQlDQD	16-886-766-7945	ges. slyly re

Fig 16: Query 2 results

Pressing the QEP button will display the query execution plan graph where each node will represent the query action and the cost associated with the step.

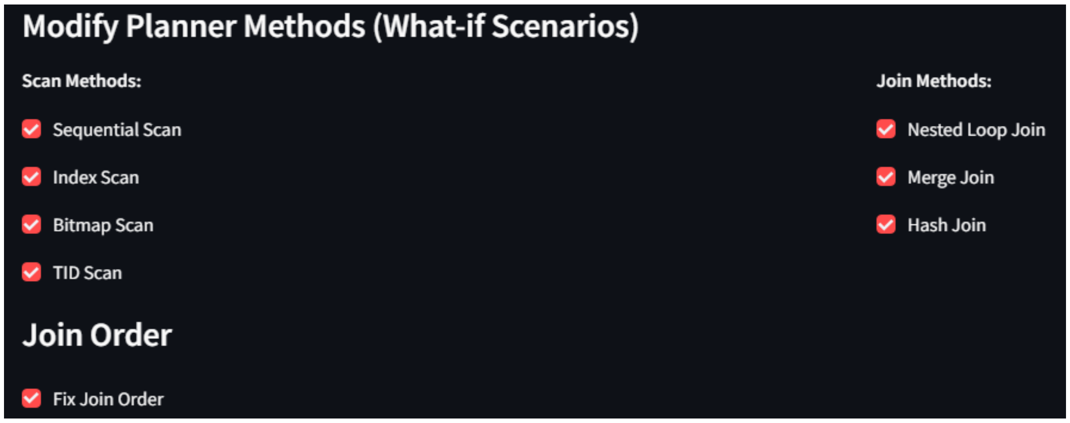

Get QEP

Query Plans

Query Execution Plan (QEP) - Total Cost: 40690.18

Fig 17: Query execution plan (QEP) of Query 2

A possible configuration for an alternative query plan is to fix the join order. To do this, in the settings parameter, check the box to fix join order.



Modify Planner Methods (What-if Scenarios)

Scan Methods:

- ✓ Sequential Scan
- ✓ Index Scan
- ✓ Bitmap Scan
- ✓ TID Scan

Join Methods:

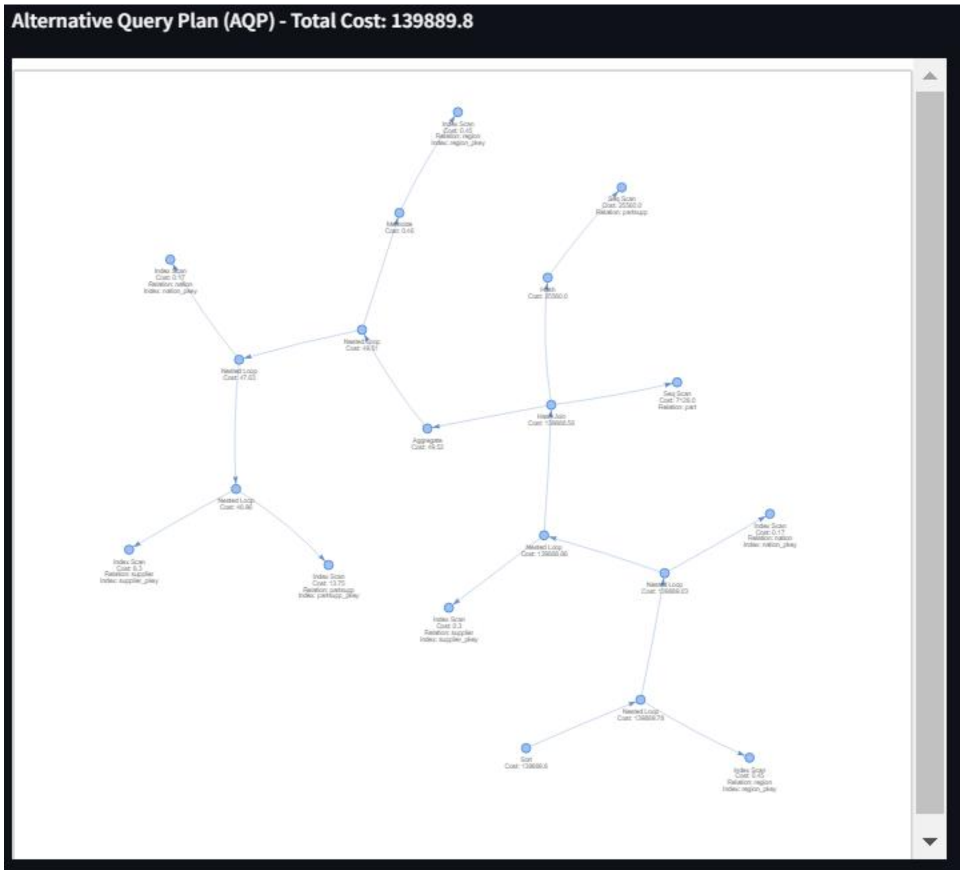
- ✓ Nested Loop Join
- ✓ Merge Join
- ✓ Hash Join

Join Order

- ✓ Fix Join Order

Fig 18: Example of a possible modification to query – fixing join order

Clicking on the ‘Get AQP button’ using the modification shown above will generate the following graph for the AQP.


Alternative Query Plan (AQP) - Total Cost: 139889.8

Index Scan Cost: 0.45 Relation: region Index: region_pkey
Materialize Cost: 0.46
Seq Scan Cost: 25560.0 Relation: partsupp
Index Scan Cost: 0.17 Relation: nation Index: nation_pkey
Hash Cost: 25560.0
Nested Loop Cost: 49.51
Nested Loop Cost: 47.03
Seq Scan Relation: part
Hash Join
Aggregate
Nested Loop
Index Scan Cost: 0.17 Relation: nation Index: nation_pkey
Index Scan Cost: 0.3 Relation: supplier Index: supplier_pkey
Index Scan Cost: 13.75 Relation: partsupp Index: partsupp_pkey
Nested Loop
Index Scan Cost: 0.3 Relation: supplier Index: supplier_pkey
Nested Loop
Nested Loop
Sort
Index Scan Cost: 0.45 Relation: region Index: region_pkey

Fig 19: AQP of query 2 by fixing join order

7 Conclusion

In conclusion, this project developed a user-friendly interface for analyzing and comparing query execution plans (QEPs) with alternative query plans (AQPs). The application also provides insights into the reasons why the AQP have a higher cost than QEP usually by using a large langauge model to explain the cost differences. Despite some limitations, the project may serve as a practical tool for education in query optimisation.

Annex A

A.1 Installation Guide

```Bash
$ git clone https://github.com/woonyee28/What-If-Query-Plan.git

$ cd ./What-If-Query-Plan

$ python -m venv venv

$ source venv/bin/activate

$ pip install poetry

$ poetry install

$ poetry shell

$ python project.py

Local URL: http://localhost:8501
```