



## **SC3020 Database System Principles**

**Project 2 (20%)**

**Report Submission**

**Group 4**

**Prepared By:**

<b>Name</b>	<b>Matriculation Number</b>
Lohia Vardhan	U2120105F
Teoh Xi Sheng	U2120456L
Iyer Anushri	U2123122E
Pugalia Aditya Kumar	U2123212D
Mukherjee Tathagato	U2120365K

<b>Table of Contents.....</b>	<b>2</b>
<b>1. Introduction.....</b>	<b>3</b>
<b>2. TPC-H Database.....</b>	<b>3</b>
<b>3. Project Implementation.....</b>	<b>4</b>
3.1 Architecture Overview.....	4
3.2 Frontend Interface.....	5
3.3 Backend Logic.....	8
3.3.1 Retrieval of Query Execution Plan & Generation of Query Tree.....	9
3.3.2 Implementation of What-If Queries.....	11
3.3.2.1 Join Order Modification.....	11
3.3.2.2 Operator Modification.....	13
<b>4. Application Demo.....</b>	<b>15</b>
4.0 Assumptions.....	15
4.1 Database Connection.....	15
4.2 SQL Query Input.....	16
4.3 QEP Graph.....	17
4.4 QEP Join Order Modification.....	18
4.5 QEP Node Type Modification.....	19
4.6 Alternate Query Plan (AQP).....	20
<b>5. Limitations and Evaluation.....</b>	<b>22</b>

# 1. Introduction

Modern Database Management Systems execute Query Execution Plans (QEPs) to optimise SQL query execution. The QEP is selected from a large search space of Alternative Query Plans (AQPs) that are built to execute the same query in different manners. This report outlines our design and implementation of a system that can visualise QEPs extracted from PostgreSQL and allow users to perform “What-If” queries on Query Execution Plans. This allows users to modify QEP requirements by specifying operators or join orders, thus creating a specification-based AQP that can be compared for performance with the QEP.

## 2. TPC-H Database

The TPC-H database is a benchmarking tool which is developed to evaluate the performance of various database management systems (DBMS). The TPC-H dataset was put in PostgreSQL to generate a database whose relations were as follows:

TABLE	DESCRIPTION	# OF TUPLES
CUSTOMER	List of all customers with details	150,000
LINEITEM	List of all transport lines with details	6,001,215
NATION	List of all nations	25
ORDERS	List of orders with details	1,500,000
PART	List of parts and their details	200,000
PARTSUPP	List of suppliers of different parts	800,000
REGION	List of all regions	5
SUPPLIER	List of suppliers with details	10,000

## 3. Project Implementation

### 3.1 Architecture Overview

The following diagram illustrates the application's architecture overview. The application is developed using multiple modular components, each designed to perform a specific function independently.

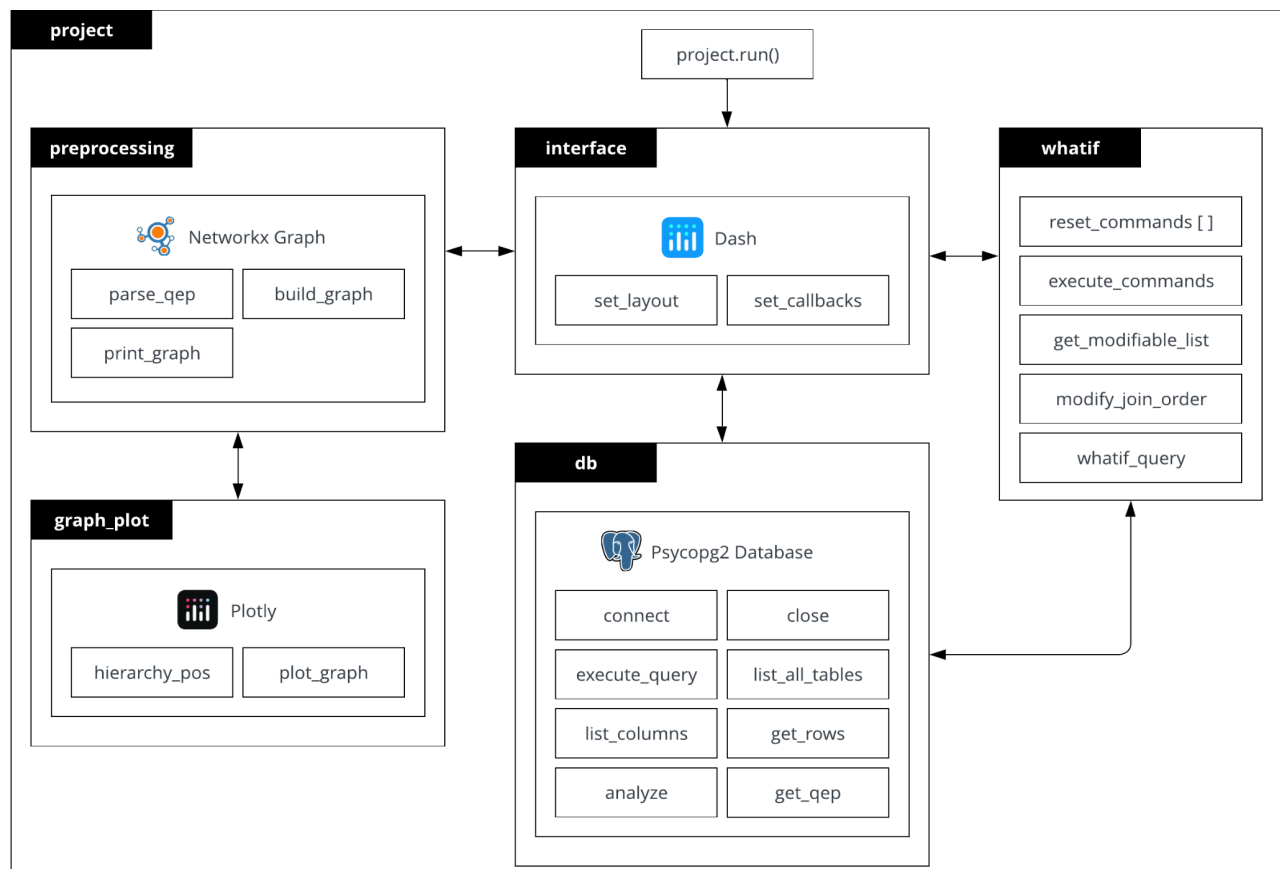


Figure 1. Application architecture diagram.

## 3.2 Frontend Interface

The frontend interface is developed using **Dash** and **Plotly** libraries to provide a user-friendly interface where the users can connect to the database, input SQL queries, modify the QEP and visualise QEP in graph format.

The logic and core components of the frontend interface include:

- 1. User Input and Feedback Handling:** The user input allows the user to perform actions such as connecting to the database, executing a query, generating the QEP, and manipulating the QEP with join ordering or node type modification. Input feedback handling was also implemented to enhance the user experience by guiding and providing visual cues of the system and operation state.



Figure 2. Disabled button to prevent the user from clicking until the user is connected to the database.

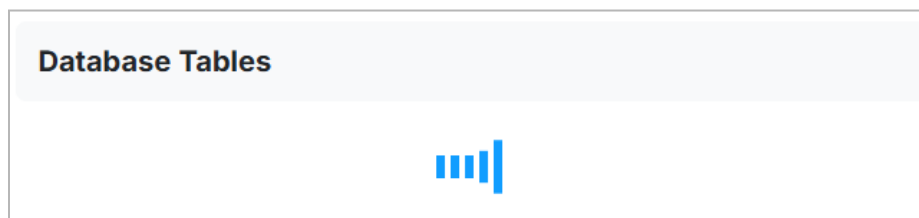


Figure 3. Loading feedback to show the loading state of the execution.

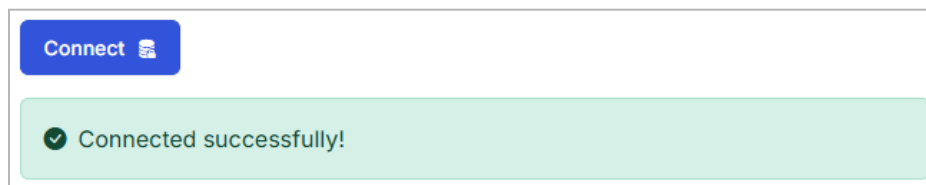


Figure 4. Success feedback to show the successful outcome of the operation.

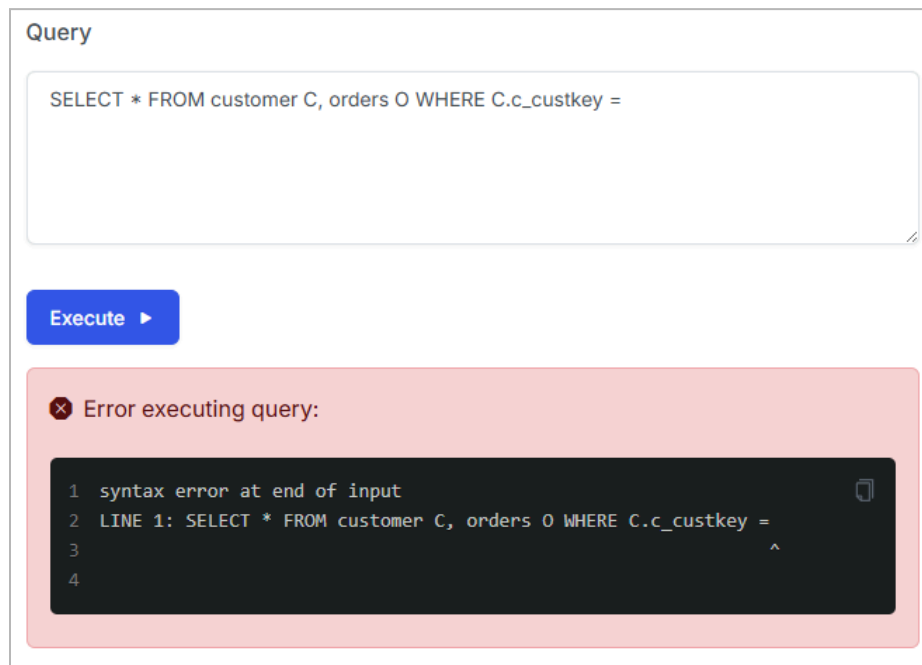


Figure 5. Error feedback with proper code formatting.

- 2. Graph Visualisation:** This component was implemented using the Plotly library to visualize the tree graphs of the QEP and AQP. This allows the user to view and compare the structure of the graphs easily.

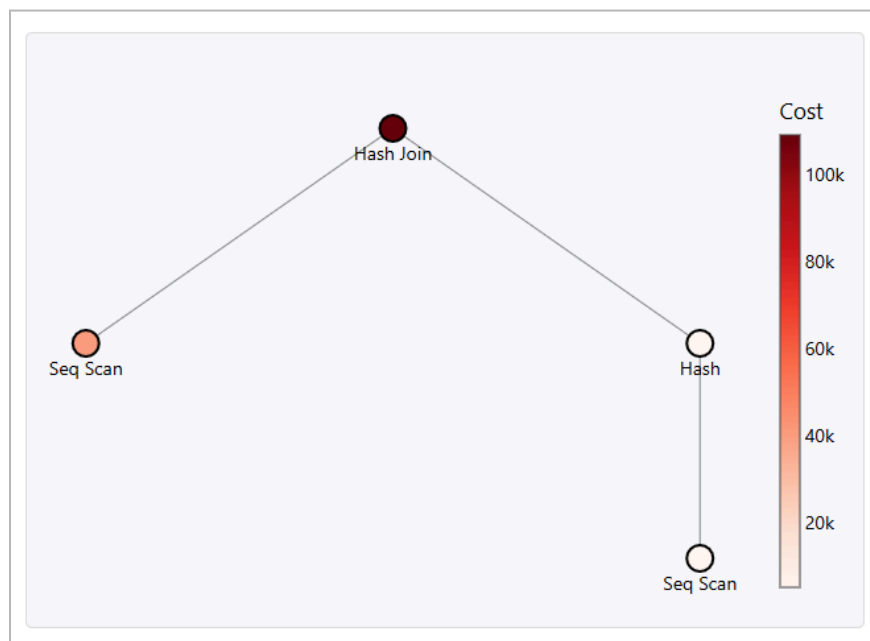


Figure 6. Graph Visualization using Plotly and Networkx library.

3. **Interactive Components:** The interactive component includes graph interaction and elements like dropdown menus that users can interact with to modify the node types and visualise the output of the AQP instantly.

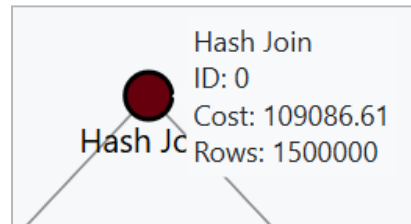


Figure 7. Mouse hover to show the details of the node.

Figure 8. Dropdown select menu to manipulate the node types of QEP.

To reorder the join sequence in the QEP, users can review the default join order and modify it by specifying the desired order of relations as a comma-separated list.

Figure 9. Re-ordering the join orders for QEP.

### 3.3 Backend Logic

In this project, we connect to the TPC-H database in PostgreSQL using the `psycopg2` database adapter in Python to facilitate the interaction and connection. It uses a connection object in `psycopg2` to establish a connection to the database, and then gets a `Cursor` object to execute queries on the database.

```
db.py

import psycopg2
import time

class Database:

    def execute_query(self, query, params=None):
        """
        Execute a query and return the result
        """
        try:
            start_time = time.time()
            self.cursor.execute(query, params)

            if self.cursor.description:
                result = self.cursor.fetchall()
                row_count = self.cursor.rowcount
            else:
                self.conn.commit()
                result = None
                row_count = 0

            execution_time = time.time() - start_time
            return result, execution_time, None, row_count
        except psycopg2.Error as e:
            print(f"Error executing query: {e}")
            return None, 0, str(e), 0
```

Figure 10. Code snippet for the `execute_query` method implementation.

In the following section, we will detail the implementation of the two primary objectives of this project:

1. Retrieval of QEP from PostgreSQL and visualising the Query Tree
2. Dynamically modifying the QEP based on operator or join order input from the user to retrieve the AQP (What-If Queries)



### 3.3.1 Retrieval of Query Execution Plan & Generation of Query Tree

PostgreSQL offers the EXPLAIN command which is used to show the execution plan of a SQL query. In order to retrieve the Query Execution Plan, the command 'EXPLAIN (FORMAT JSON)' is appended to the inputted SQL query. This query is then executed and the query plan is retrieved in JSON format. The estimated total cost of the QEP is also retrieved by extracting the estimated cost of the root node. This is because the estimated costs in the QEP are cumulative from bottom-up. This means that the root node is the summation of costs of executing all nodes in the QEP as estimated costs of child nodes are recursively added onto the cost of the parent node.

```
db.py

def get_qep(self, query):
    """
    Get the query execution plan (QEP) for a query
    """
    query = f"EXPLAIN (FORMAT JSON) {query}"
    result, execution_time, error, _ = self.execute_query(query)
    # Extract the total cost of the top-level plan
    qep_cost = None
    qep_rows = None
    if result is not None:
        qep_cost = result[0][0][0]["Plan"]["Total Cost"]
        qep_rows = result[0][0][0]["Plan"]["Plan Rows"]
    return json.dumps(result[0][0], indent=2), qep_cost, qep_rows, execution_time, error
```

Figure 11. Code snippet for get\_qep method implementation.

The QEP is then processed into a directed-graph data structure in order to build the edges and nodes of the graph. In this part, the function parse\_qep() traverses each node in the QEP retrieved and recursively adds nodes and edges (between parent and child nodes) into respective dictionaries. If a node has a sub-plan, the function recursively traverses these sub-plans by calling itself with the current node as the parent.

For instance, if we get the QEP of the sample query, the QEP will be preprocessed into the following graph with nodes and edges:

sql

```
SELECT * FROM customer, orders, nation
WHERE   c_custkey = o_custkey
AND     c_nationkey = n_nation_key;
```

Figure 12. Sample SQL query.

nodes (simplified)

```
[
  {
    "id": 0,
    "label": "Hash Join",
    "cost": 109086.61,
    "rows": 1500000,
  },
  {
    "id": 1,
    "label": "Seq Scan",
    "cost": 41136.0,
    "rows": 1500000,
  },
  {
    "id": 2,
    "label": "Hash",
    "cost": 5300.0,
    "rows": 150000,
  },
  {
    "id": 3,
    "label": "Seq Scan",
    "cost": 5300.0,
    "rows": 150000,
  },
]
```

edges

```
[(0, 1), (0, 2), (2, 3)]
```

Figure 13. Structure of the graph in nodes (simplified) and edges format.

The graph structure will then be constructed using the **NetworkX** library and visualised with **Plotly** to generate the graph plot, as outlined in [Section 3.2](#), Figure 6.

### 3.3.2 Implementation of What-If Queries

Under What-If Queries, a user can modify operators and/or change the join order of tables. Both of these are achieved by changing the configuration parameters of PostgreSQL to enforce the desired query plan.

#### 3.3.2.1 Join Order Modification

The user can change the join order of the relevant tables by inputting their own order. The back-end code extracts all possible tables that a user can change the join order for. This is done with the use of a regex pattern which looks for comma-separated table names in the SQL query.

The logic and the pattern used are as follows:

```
whatif.py

import re

# Function to get the list of tables that can be modified
def get_modifiable_list(query_string: str):
    regex = r"\bFROM\b\s+([a-zA-Z_][a-zA-Z0-9_]*(?:\s*,\s*[a-zA-Z_][a-zA-Z0-9_]*))*"

    matches = re.findall(regex, query_string, re.IGNORECASE)
    print(matches)
    matches = [match for match in matches if len(re.findall(',', match)) > 0]
    return matches
```

Figure 14. Method to update the join orders.

- `\bFROM\b` matches the entire word FROM in the SQL query
- `\s+` looks for one or more spaces
- `[a-zA-Z_][a-zA-Z0-9_]*` finds table names starting with a letter or underscore, followed by 0 or more alphanumeric characters or underscores.
- `(?:\s*,\s*[a-zA-Z_][a-zA-Z0-9_]*)*` matches multiple table names separated by commas.

After the user enters a comma separated preferred join order, the SQL query is modified for generating QEPs. This modified SQL query is also displayed as shown in [4.4 QEP Join Order Modification](#).

To modify the SQL query, The commas in the preferred join order are replaced with 'NATURAL JOIN'. Then using the same Regex pattern as above we find and Replace the old FROM clause with the updated clause. Finally, the PostgreSQL configuration parameter *join\_collapse\_limit* is set to 1. *join\_collapse\_limit* defines the number of tables upto which PostgreSql will perform join order optimisation.

Therefore, by setting the parameter to 1 and by enforcing Natural Join in the FROM clause, we force PostgreSQL to join the tables in the order provided by the user. For instance, a modified query after switching the join order could look as follows:

```
sql
/* Original Query */
SELECT * FROM customer, orders, nation
WHERE   c_custkey = o_custkey
        AND   c_nationkey = n_nation_key;

/* Modified Query with join order - [customer, nation, order] */
SET      join_collapse_limit = 1;
SELECT * FROM customer NATURAL JOIN nation NATURAL JOIN orders
WHERE   c_custkey = o_custkey
        AND   c_nationkey = n_nation_key;
```

Figure 15. Modified query after reordering the join orders.

The `join_collapse_limit` is reset to the default value after the query execution.

```
whatif.py

# Commands for default postgresql settings
reset_commands = [
    "SET enable_bitmapscan = on;\n",
    "SET enable_gathermerge = on;\n",
    "SET enable_hashagg = on;\n",
    "SET enable_hashjoin = on;\n",
    "SET enable_indexscan = on;\n",
    "SET enable_indexonlyscan = on;\n",
    "SET enable_material = on;\n",
    "SET enable_mergejoin = on;\n",
    "SET enable_nestloop = on;\n",
    "SET enable_seqscan = on;\n",
    "SET enable_sort = on;\n",
    "SET enable_tidscan = on;\n",
    'RESET join_collapse_limit;\n'
]
```

Figure 16. SQL commands to restore the default settings for the database.

**Note:** If the Query has multiple subqueries with their individual FROM Clause then the system will provide the user the option to change the join order of the tables within each of the FROM clauses. This is shown in [Section 4.4 QEP Join Order Modification](#) of the report.

### 3.3.2.2 Operator Modification

What-If queries can be modified by choosing amongst the following operators:

#### 1. Join

- a. Hash Join
- b. Merge Join
- c. Nested Loop Join

#### 2. Scan

- a. Sequential Scan
- b. Bitmap Scan
- c. Index Scan

### 3. Aggregate

#### a. Hash

PostgreSQL has the following configuration parameters for each operator:

**Join:** [enable\_mergejoin, enable\_hashjoin, enable\_nestloop]

**Scan:** [enable\_bitmapscan, enable\_seqscan, enable\_indexscan, enable\_indexonlyscan, enable\_tidscan]

**Aggregate:** [enable\_hashagg]

By default, these parameters are set to ON so that PostgreSQL can choose between all possible operators to find the most effective QEP amongst the AQP's. Therefore, when a particular type of operator is specified through a What-If query, the other operators are set to OFF.

For example, if a user wants to enforce Hash Join, the other two possible joins will be set to OFF.

whatif.py

```
# Disable all other joins apart from hash join
if join == "hash":
    commands.extend(["SET enable_mergejoin = off;\n", "SET enable_nestloop = off;\n"])
```

Figure 17. SQL query to enforce Hash Join.

**Note:** Tidscan was not put as a What-If query option because it is less commonly used and may or may not result in the correct output. Moreover, Index only Scan is also not a separate option as it follows under the category of Index Scan.

These commands are then appended to the beginning of the original SQL query and executed. After each execution, all these parameters are reset to their default values using the reset\_commands..

## 4. Application Demo

This section outlines the demonstration of the application usage.

### 4.0 Assumptions

We assume the user has the TPC-H database set up and is ready to connect with the necessary credentials to execute SQL queries.

### 4.1 Database Connection

The first step requires users to connect to the TPC-H database using their credentials. Upon successful connection, the application displays a list of database tables and their respective row counts and attributes.

### Connection

Host
localhost

Port
5433

Database
TPC-H

User
postgres

Password
.....

Connect

✓ Connected successfully!

### Database Tables

Time taken: 13,199 ms

TABLE	ATTRIBUTES
customer	c_custkey, c_name, c_address, c_nationkey, c_phone, c...
lineitem	l_orderkey, l_partkey, l_suppkey, l_linenum, l_quantity
nation	n_nationkey, n_name, n_regionkey, n_comment
orders	o_orderkey, o_custkey, o_orderstatus, o_totalprice, o_or...
part	p_partkey, p_name, p_mfgr, p_brand, p_type, p_size, p_c...
partsupp	ps_partkey, ps_suppkey, ps_availqty, ps_supplycost, ps...
region	r_regionkey, r_name, r_comment
supplier	s_suppkey, s_name, s_address, s_nationkey, s_phone, s...

Figure 18: Database connection interface.

## 4.2 SQL Query Input

The application allows users to input SQL queries in two ways:

1. By selecting predefined query templates from the dropdown menu; or
2. By manually entering their custom SQL statements in the text input area.

Once a query is input, users can execute it, and the query output and execution time will be displayed in the output panel.

### SQL Query Input

Query Template

Default (2 Joins and 1 Nested SELECT with 2 Joins) ▼

Query

```
SELECT s_name, r_name, ps_supplycost * ps_availqty AS
total_cost FROM supplier, nation, region, ( SELECT
ps_partkey, ps_suppkey, ps_availqty, ps_supplycost FROM
partsupp, part, lineitem WHERE ps_partkey = p_partkey AND
ps_partkey = l_partkey AND l_shipdate = '1997-01-01' ) AS ps
WHERE s_suppkey = ps_suppkey AND s_nationkey =
n_nationkey AND n_regionkey = r_regionkey AND r_name =
```

Execute ▶

✓ Query executed successfully!

### SQL Query Output

Time taken: 277 ms

Rows returned: 2014

S_NAME	R_NAME	TOTAL_COST
Supplier#000007816	ASIA	9719305.02
Supplier#000004141	ASIA	9605219.18
Supplier#000002761	ASIA	9514740.8
Supplier#000005308	ASIA	9477077.98
Supplier#000001479	ASIA	9420356.88
Supplier#000005233	ASIA	9394502.96
Supplier#000001870	ASIA	9322249.86
Supplier#000003340	ASIA	9220394.56

Figure 19: SQL query execution interface.



### 4.3 QEP Graph

After inputting the query statement, the application can now generate the QEP. By clicking “Get QEP,” the application runs the EXPLAIN statement in the backend and returns the QEP in JSON format. This JSON data is then used to create a network graph using the **NetworkX** library, with the graph plotted using **Plotly**. To enhance user experience, features such as colour gradients and tooltips on hover are implemented, improving visual clarity and interactivity.

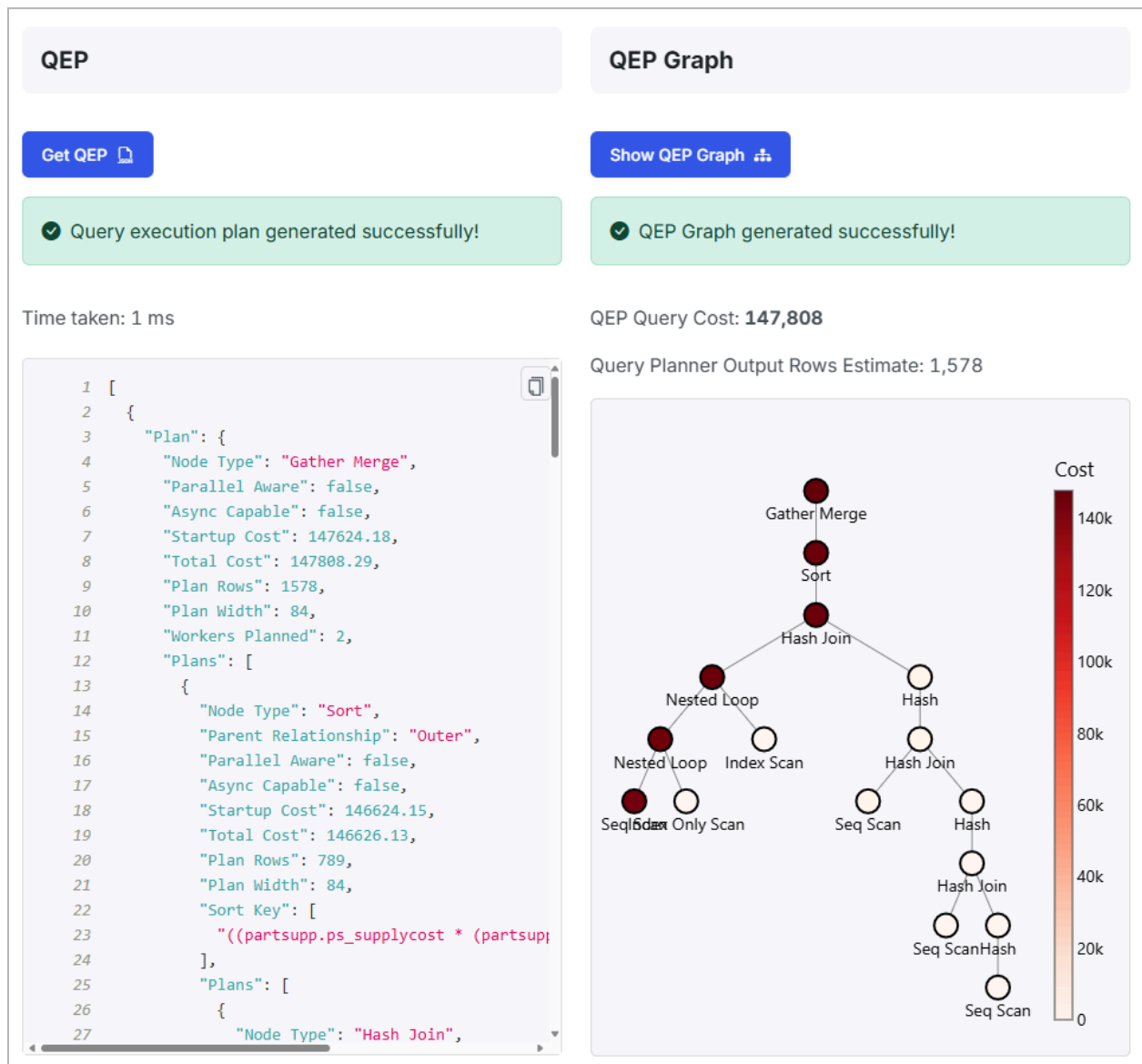


Figure 20: QEP output interface.

## 4.4 QEP Join Order Modification

The application allows users to reorder and rank the join order of the “WHERE” clause by inputting the relations separated by commas. For instance, the first relation will be joined first, and the result will then be joined with the third relation iteratively. This allows users to explore how different join orders impact query performance regarding execution cost and time.

The user can preview the modified SQL query after reordering the relations.

### Modify Join Order

View Join Orders 🔍

Default Join Order: supplier nation region

Modify the join order separated by commas  
supplier, nation, region

Default Join Order: partsupp part lineitem

Modify the join order separated by commas  
partsupp, part, lineitem

Preview Modified Join Orders 🔍

Reset Join Orders ↺

Modified SQL Query:

```

1 SELECT s_name,
2        r_name,
3        ps_supplycost * ps_availqty as total_cost
4 FROM   supplier natural join nation natural join region, (SELECT ps_partkey,
5                                                                    ps_supkey,
6                                                                    ps_availqty,
7                                                                    ps_supplycost
8                                                                    FROM   partsupp natural join part natural join lineitem
9                                                                    WHERE  ps_partkey = p_partkey
10                                                                    and ps_partkey = l_partkey
11                                                                    and l_shipdate = '1997-01-01') as ps
12 WHERE  s_supkey = ps_supkey
13        and s_nationkey = n_nationkey
14        and n_regionkey = r_regionkey
15        and r_name = 'ASIA'
16 ORDER BY total_cost desc;

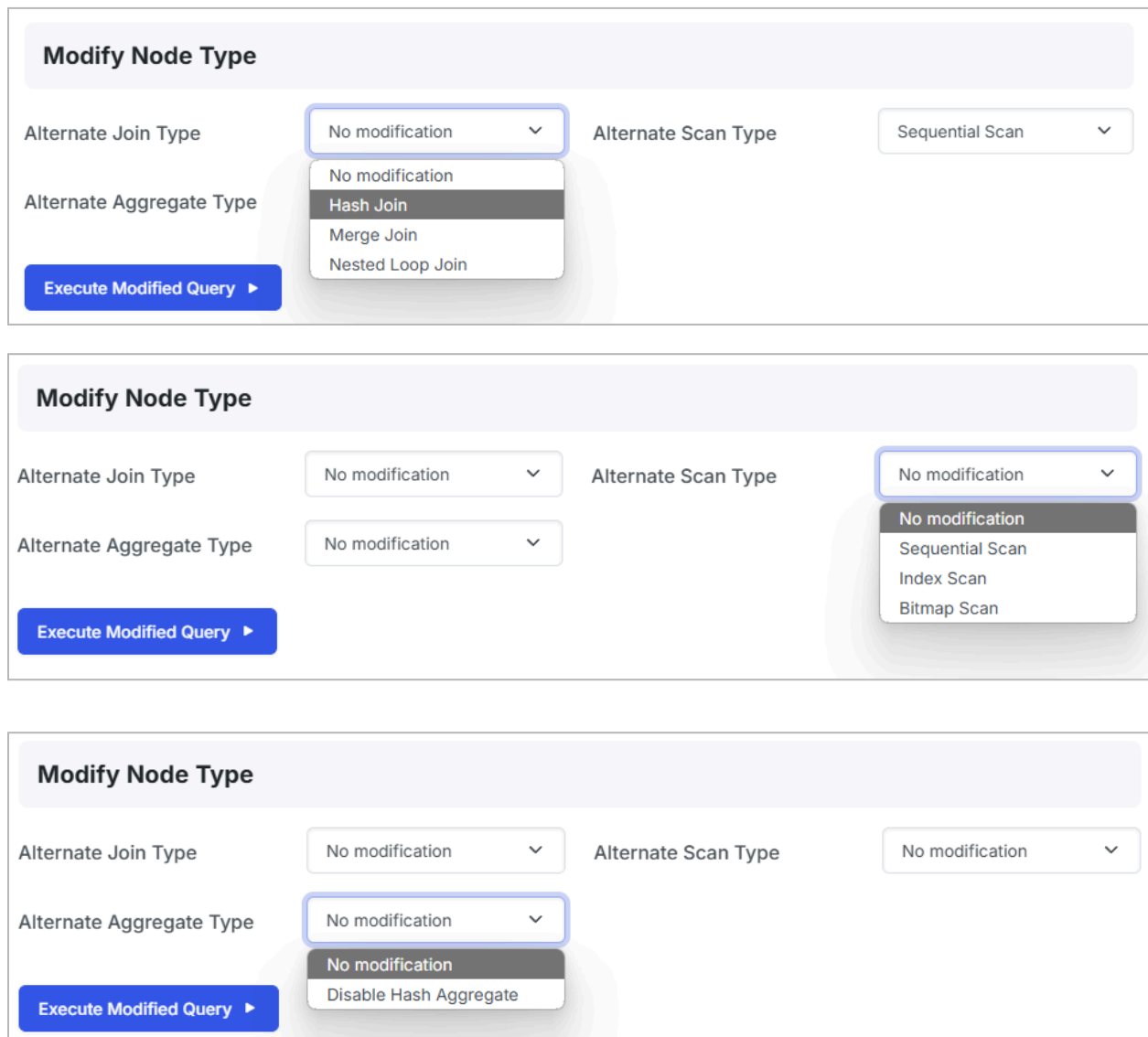
```

Figure 21: QEP modification by reordering the join orders.

## 4.5 QEP Node Type Modification

**Node Type Selection:** Users can change node types (e.g., Nested Loop, Hash Join) through a dropdown menu for the Join, Aggregate, and Scan operations.

After all the modifications made to the QEP, users can now execute the modified query to obtain the Alternate Query Plan (AQP).



The figure displays three sequential screenshots of the 'Modify Node Type' interface, illustrating how different node types can be selected for Join, Aggregate, and Scan operations.

**Screenshot 1 (Top):** The 'Alternate Join Type' dropdown is open, showing options: 'No modification', 'Hash Join' (highlighted), 'Merge Join', and 'Nested Loop Join'. The 'Alternate Scan Type' is set to 'Sequential Scan'. The 'Alternate Aggregate Type' is set to 'No modification'. The 'Execute Modified Query' button is visible.

**Screenshot 2 (Middle):** The 'Alternate Scan Type' dropdown is open, showing options: 'No modification' (highlighted), 'Sequential Scan', 'Index Scan', and 'Bitmap Scan'. The 'Alternate Join Type' is set to 'No modification'. The 'Alternate Aggregate Type' is set to 'No modification'. The 'Execute Modified Query' button is visible.

**Screenshot 3 (Bottom):** The 'Alternate Aggregate Type' dropdown is open, showing options: 'No modification' (highlighted) and 'Disable Hash Aggregate'. The 'Alternate Join Type' is set to 'No modification'. The 'Alternate Scan Type' is set to 'No modification'. The 'Execute Modified Query' button is visible.

Figure 22: QEP modification by changing the node types.

## 4.6 Alternate Query Plan (AQP)

When users execute the modified query, the application will display the modified query statement, which includes the settings for the node types and the join orders.

Execute Modified Query ▶

✓ What-If Query executed successfully!

**Final Modified SQL Query:**

```

1  set enable_indexscan = off; set enable_indexonlyscan = off; set enable_bitmapscan = off; set enable_tidscan = off;
2  SELECT s_name,
3         r_name,
4         ps_supplycost * ps_availqty as total_cost
5  FROM   supplier, nation, region, (SELECT ps_partkey,
6                                         ps_suppkey,
7                                         ps_availqty,
8                                         ps_supplycost
9                                         FROM   partsupp, part, lineitem
10                                        WHERE  ps_partkey = p_partkey
11                                             and ps_partkey = l_partkey
12                                             and l_shipdate = '1997-01-01') as ps
13  WHERE  s_suppkey = ps_suppkey
14         and s_nationkey = n_nationkey
15         and n_regionkey = r_regionkey
16         and r_name = 'ASIA'
17  ORDER BY total_cost desc;
```

Figure 23: SQL query of the AQP.

The application will then display the query execution plan in JSON format and the graph of the AQP. In the following figure, we can observe that setting the join nodes to hash join increases the query cost and, therefore, results in worse performance when changing the default query execution plan determined by PostgreSQL.

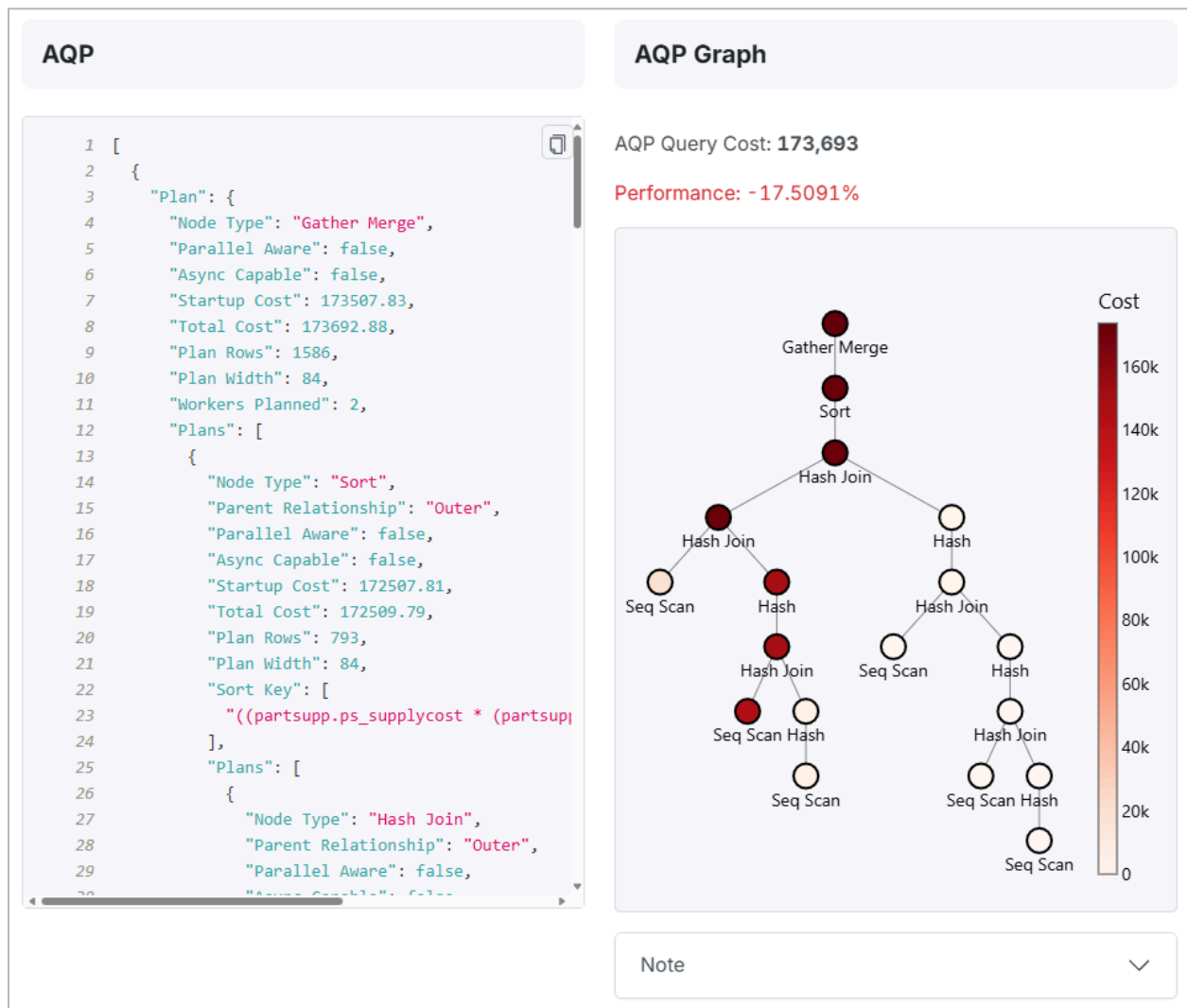


Figure 24: AQP in JSON and Graph format.

## 5. Limitations and Evaluation

Through the What-If questions we allow the user to visualize how the QEP of the query would look like if we used different operators / join ordering. This is primarily achieved by changing the configuration parameters of PostgreSQL to enforce a certain Query Plan. However, this method has its own limitations which are identified below:

### 1. Operator Enforcement in Query Plans

The operators chosen by the user for the What-If question is enforced throughout the query plan. It is not possible to enforce operators for individual nodes in the Query Tree. For instance, if there are 2 joins in the Query the user cannot specify a different join type for each of those joins.

The effect of the change of the configuration parameters (such as disabling other join types) is applied throughout the query and cannot be isolated for individual operators in PostgreSQL.

### 2. Limitations of Operator Enforcement

The Operator type chosen in the What-If Questions isn't guaranteed to be enforced always. For instance, an Index scan cannot be performed if an index on the particular attribute does not exist in the first place. By manipulating the configuration parameters the program can only guide PostgreSQL to choose our desired operators. However, if doing so is not possible for a particular node then PostgreSQL will decide the type of operator to use.

### 3. Restrictions on Join Reordering

The option to reorder the tables for the join clause is only available for FOR clauses with comma-separated table names. If the FOR clause in the original query specifies a certain type of Join (E.g. FROM customer LEFT OUTER JOIN order) then the option to reorder the joins for that query is not provided.

This is because we enforce the joins by replacing the commas with NATURAL JOIN to force PostgreSQL to use the specified join order. If the original query already specifies a certain type of join as seen in the example above, enforcing the join order using NATURAL JOIN could change the output of the query. Thus, for the purpose of this project join reordering has been limited to comma-separated relations in the FROM clause.

### 4. Regex Constraints for Table Name Matching

The regex for to match for table name finds table names starting with a letter or underscore, followed by 0 or more alphanumeric characters or underscores. It does not allow spaces to be present in the table names. Thus, if the User inputs table aliases in the FROM clause it will not be matched by the regex.

For instance, the system would not provide any join reordering option to the user for the below query:

sql

```
SELECT * FROM customer C, orders O WHERE C.custkey = O.custkey;
```

Figure 25. Invalid query for reordering join order.

Considering the above limitations, we propose an alternative approach for dealing with What-If queries—*pg\_hint\_plan*. This PostgreSQL extension allows users to add directives for PostgreSQL in the form of hints to manipulate the QEP. It allows the user to set the operator types for each operation and specify the join order for all relations through the hints, thus providing more flexibility.

*pg\_hint\_plan* first interprets the hints provided in the form of block comments at the beginning of the query, as seen below:

```
sql
SELECT /*+ SeqScan(customer) */ * FROM customer;
```

Figure 26. Custom join ordering using *pg\_hint\_plan* extension.

It then modifies the internal structure of the PostgreSQL Planner to adjust the cost estimate associated with various execution strategies. By making your suggested execution strategy seem to be the most cost-effective it forces PostgreSQL to choose that plan.

*pg\_hint\_plan* is specifically implemented for Linux-based operating systems. Given the constraint of the project to run it on a Windows operating system, we would need to either install a virtual machine or use docker containers with PostgreSQL, *pg\_hint\_plan* and the database preloaded. This would become too complex for the end user to set up and evaluate our implementation, hence it was beyond the scope of this project.