NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

AY 2024/25 SEMESTER 1
SC3020 Database System Principles
Project 2 - Report

By Group 21

Name	Matric No.	Contributions
Wong Zi Lun	U2240220D	1. Preprocessing.py 2. Report writing
Javen Ong Jing Hian	U2240610E	1. whatif.py algorithm 2. Report writing
Thejesvi Ramesh	U2240554H	1. Preprocessing.py 2. Report writing
Jared Chong Geng Zhong	U2240283J	1. GUI 2. Report Writing
Kenny Seah Yong Jie	U2121539K	1. GUI 2. Report Writing

1. Introduction

This report outlines our design and implementation of an application that generates specific "what if" scenarios based on inputted Query Execution Plans. Our application will provide the user with the original costs of their inputted SQL Query, the ability to alter certain parameters based on their inputted QEP, visual representations of their original and altered QEP, and cost comparisons of both execution plans. This report aims to illustrate our program's functionality in helping users better understand how certain changes to their QEP can affect their output costs.

2. Dataset

The TPC-H Database was primarily used throughout our project lifetime. Its schema consists of 8 tables of data in total: region, nation, customer, supplier, part, partsupp, orders, lineitem. Each of the tables has between 3 to 16 attributes. Alternative Databases and their schemas may be inputted into the software in the Login Page section of the application, or be chosen after Login.

3. Python Modules Used

1) **customtkinter (ctk)**
   This is a third-party package that provides custom-styled widgets and themes for the tkinter module, making it easier to create modern-looking GUIs in Python. It is not part of the Python Standard Library, so it must be installed separately

2) **tkinter (tk)**
   This is the GUI package customtkinter is based on, which creates UI elements based on widgets. However, due to the basic appearance and lack of customizability of tkinter widgets, customtkinter is used to create the majority of UI elements in our GUI. This is with the exception of the canvas to embed the graph plots of the QEP and AQP trees, as customtkinter does not both support vertical and horizontal scroll simultaneously for a canvas.

3) **<u>tkinter.messagebox</u>**

The module tkinter is generally used for creating python based GUIs, while messagebox is imported specifically for displaying message boxes.

4) **<u>psycopg2</u>**

This module is a third-party package used for interacting with PostgreSQL databases and establishing connections between python and the database itself. It is not part of the Python standard library, so it must be installed separately

5) **<u>networkx (nx)</u>**

This module is a third-party library for the creation, manipulation, and study of the structure and dynamics of complex networks and graphs. This package is not part of the standard library and needs to be installed separately

6) **<u>matplotlib.pyplot (plt)</u>**

A library for creating static, animated, and interactive visualisations for our graphs.

7) **<u>matplotlib.backends.backend_tkagg</u>**

A submodule of matplotlib used to embed figures into a tkinter GUI application.

8) **<u>matplotlib.patches.FancyArrowPatch</u>**

A submodule for creating custom arrow shapes in plots.

9) **<u>typing.TypedDict</u>**

This module is a standard library module that provides support for type hints in Python. TypeDict is used to define dictionary-like structures with a fixed set of keys, where the types of the values are known.

10) **<u>ast</u>**

A standard Python module that helps in parsing, processing, and analysing Python abstract syntax trees (AST).

11) itertools

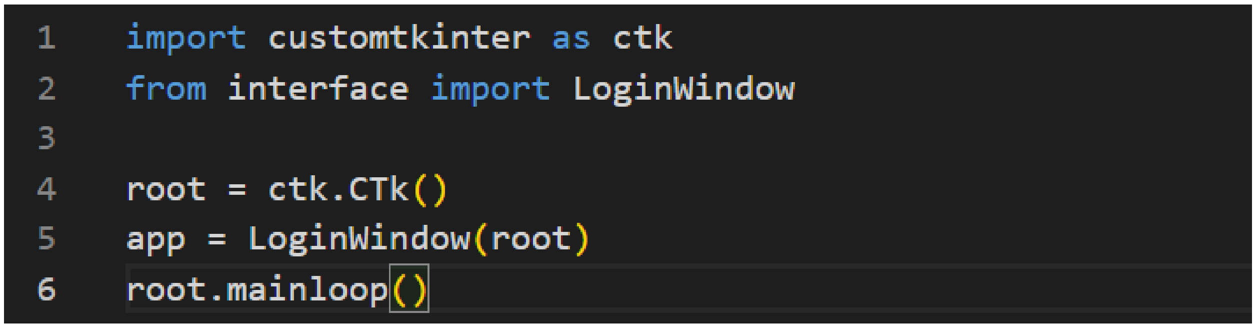
This module is a standard Python library that provides functions that create iterators for efficient looping. It includes tools for working with iterators, such as combinations and permutations.

12) re

This module is a standard Python library for working with regular expressions. It allows you to perform pattern matching and search or replace operations on strings.

4. project.py

This is our main file from which our main application, LoginWindow from our interface.py file, is called and ran.



```python
import customtkinter as ctk
from interface import LoginWindow

root = ctk.CTk()
app = LoginWindow(root)
root.mainloop()
```

Figure 1. Our main file program

5. interface.py

5.1 Designing User Interface (UI)

Rudimentary design of our GUI was first coded up to get a general sense of where all the necessary widget placements would be. This initial designing of the GUI allowed us to have a good starting point to build off of and improve. Below shows a figure of our initial draft of our GUI.

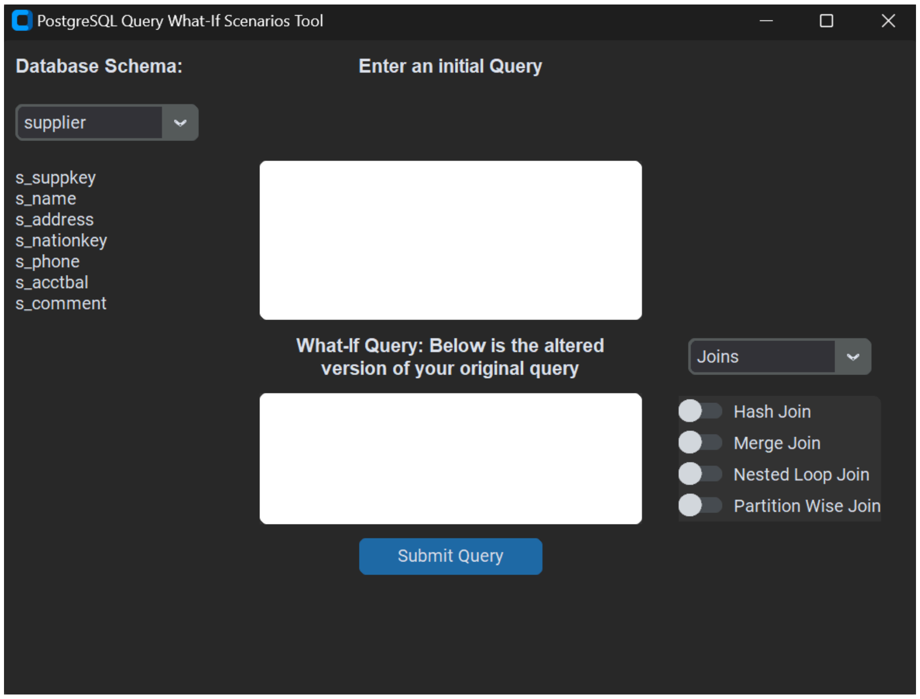


Figure 2. Initial code up of GUI design

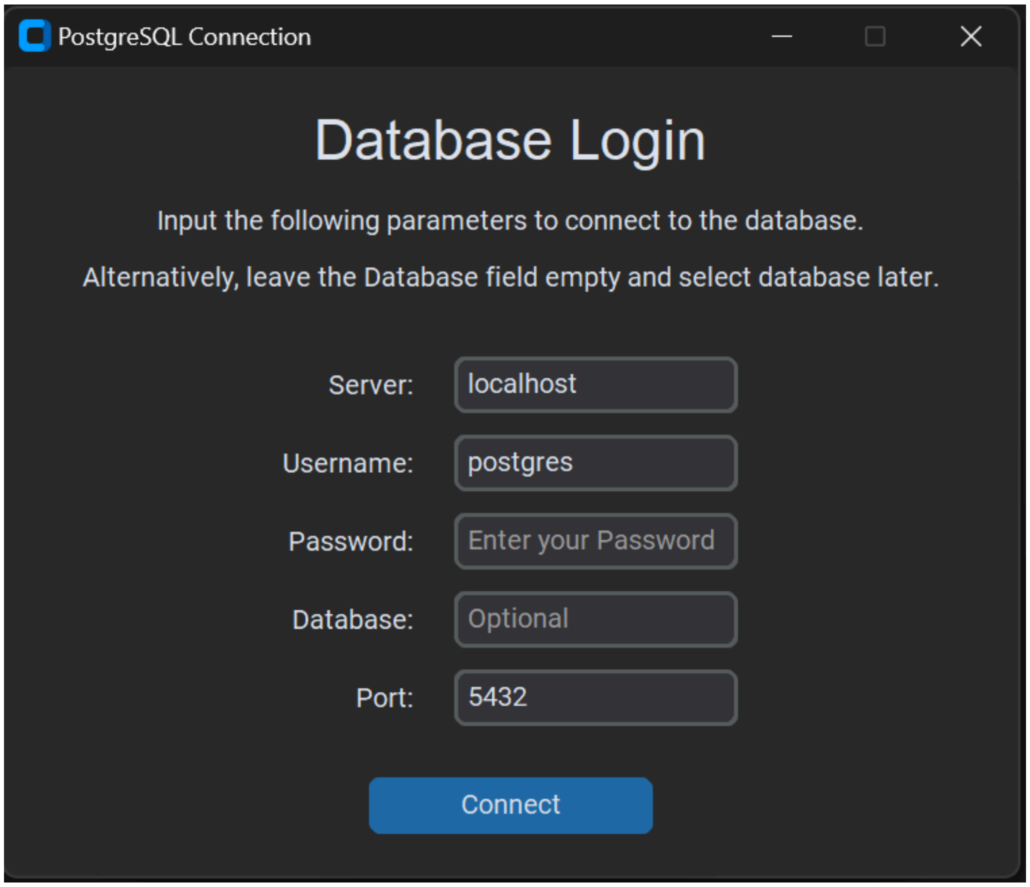
After a bit more tweaking, we finally settled on a design layout that incorporates all the above elements of the initial draft, and also implements more functionalities, while looking more cohesive and clean aesthetically. Our end product also contains a scrollbar in order to fit more information within the window.

5.2 Creating the GUI Components

Our GUI was built using customtkinter. **customtkinter** provides a more modern and visually appealing look to widgets. It allows us to customise the appearance of standard tkinter elements, such as buttons, frames, and sliders, to create a more user-friendly and aesthetically pleasing interface.

5.2.1 Login Window

In order to establish a connection with an external PostgreSQL server and database, we implemented a Login Window as shown below. Upon running the main file, users will be greeted with the aforementioned window, where they will be prompted to fill in the following fields to login to their postgreSQL account.



PostgreSQL Connection

Database Login

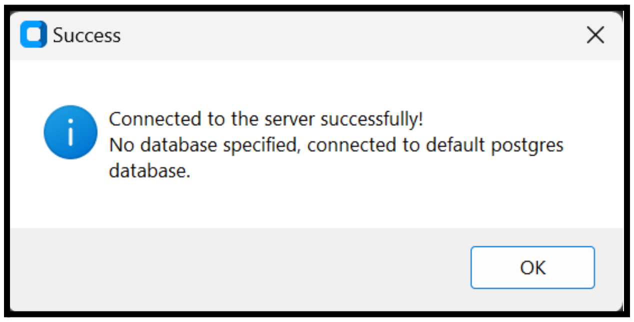
Input the following parameters to connect to the database.

Alternatively, leave the Database field empty and select database later.

Server:	localhost
Username:	postgres
Password:	Enter your Password
Database:	Optional
Port:	5432

Figure 3. Login Window Page

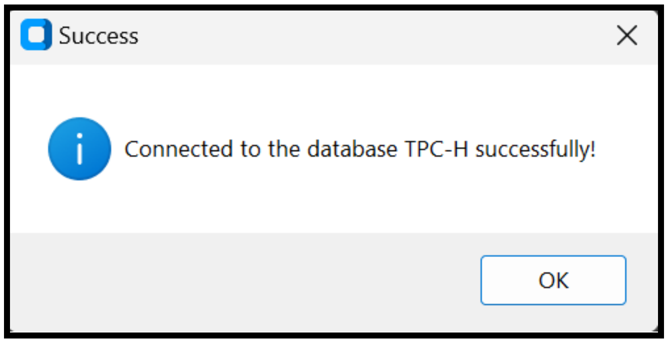
Within our Login Page, we gave users the option of inputting their desired Database that they want to work with or not, giving users more optionality. Should they decide to leave the Database field blank or not, they will be prompted with either one of the respective Pop-Ups shown below, signifying the successful connection to their postgreSQL server.



Success

Connected to the server successfully!
No database specified, connected to default postgres database.

Figure 4. Successful Connection Pop-up without Database specification

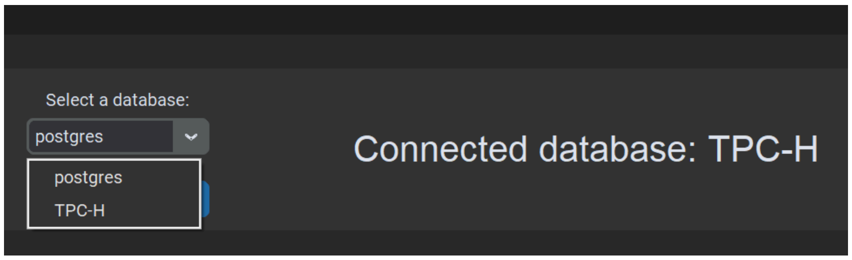


Connected to the database TPC-H successfully!

Figure 5. Successful Connection Pop-Up with Database specification

5.2.2 Connected Database

Upon clicking “OK” on the Pop-up, the user will then be taken to our main Application page, where they will be able to perform SQL Queries and compare the costs of various Query Plans. At the top of our application window, we have a Connected Database section, which specifies the database which the user is currently connected to. Should they wish to change it, they may do so by using the drop down menu on the left, selecting their desired database, and clicking the “Connect Button”, as shown below. (The Connect button is behind the dropdown menu in the Figure below)

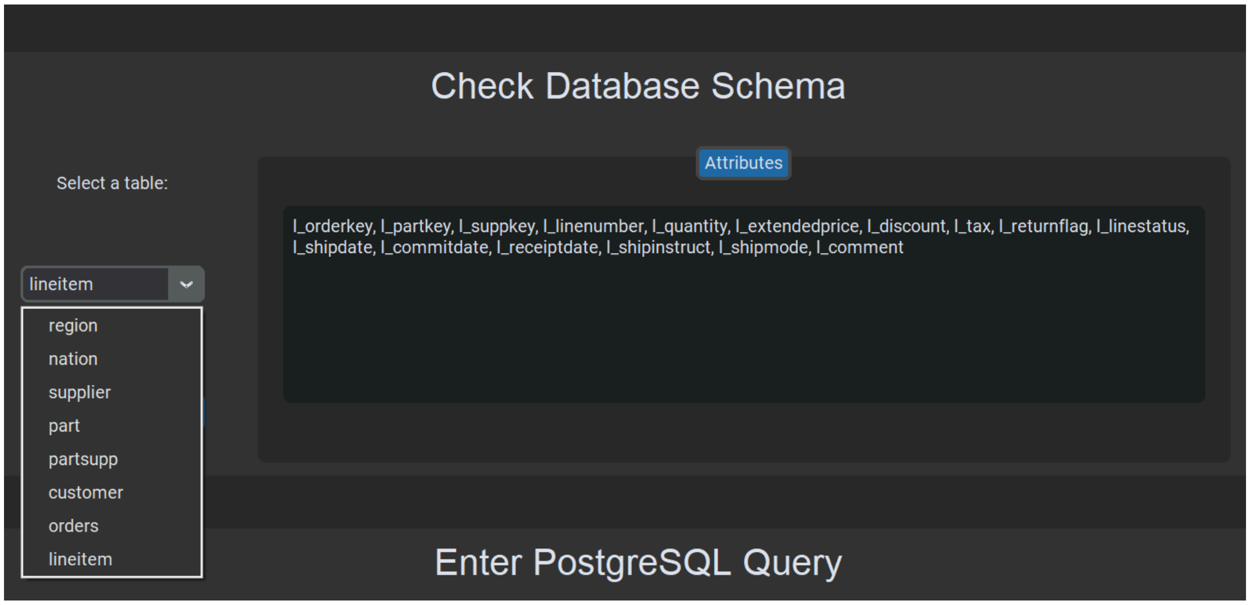


Connected database: TPC-H

Figure 6. Connected Database Menu

5.2.3 Database Schema Window

Next, we have a Check Database Schema section, where users can ensure that their desired database is correct, and check various table attribute names through selecting a specific table and clicking the “Select” button. (The Select Button is behind the drop down menu in the figure below)

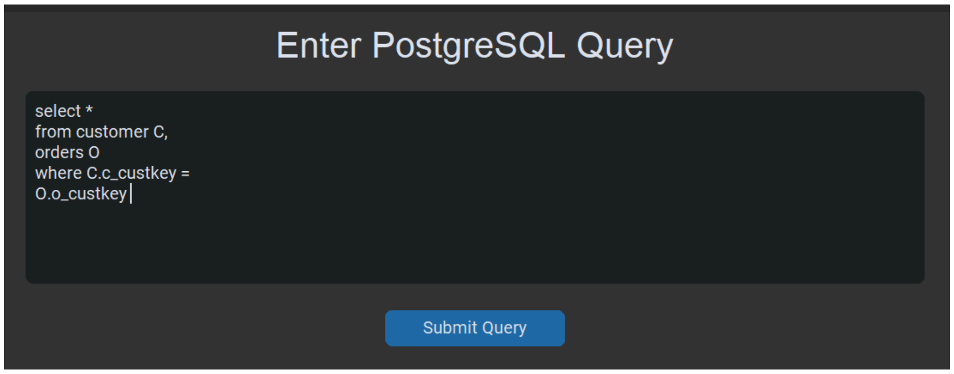


l_orderkey, l_partkey, l_suppkey, l_linenumber, l_quantity, l_extendedprice, l_discount, l_tax, l_returnflag, l_linestatus, l_shipdate, l_commitdate, l_receiptdate, l_shipinstruct, l_shipmode, l_comment

Figure 7. Check Database Schema Window

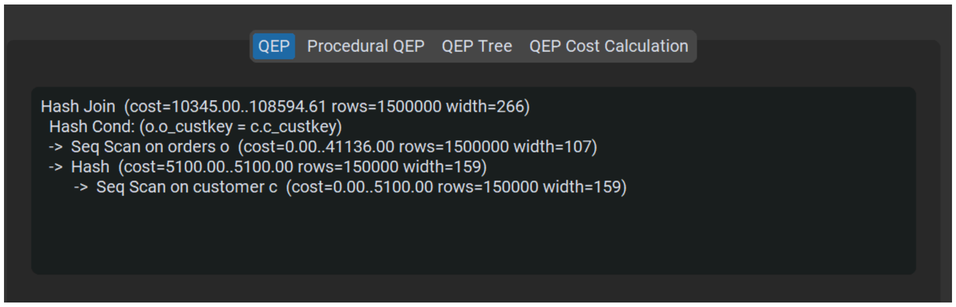
5.2.4 Query Execution Plan Panel

Our Query Execution Panel is where users may enter their SQL Queries, one at a time, and press the “Submit Query” button. Upon submitting their query, the application will return the corresponding Query Execution Plan (QEP), a procedural QEP, a visualisation of the tree structure of the QEP, and the QEP Cost calculation. For our example query in this section, we will be using the simple query provided in the Project Description.



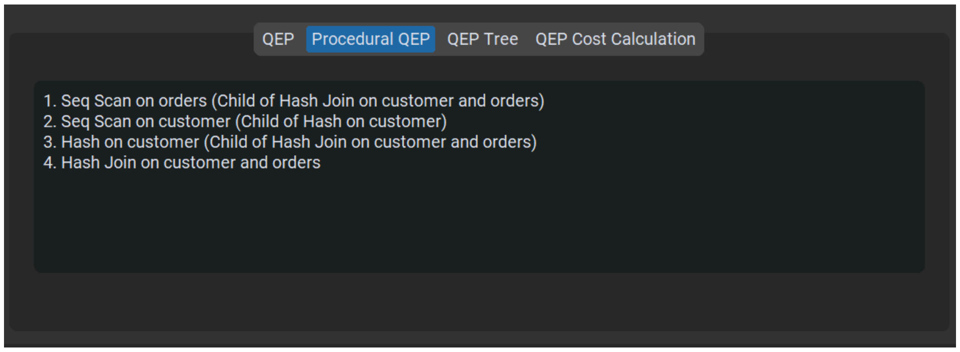
```
select *
from customer C,
orders O
where C.c_custkey =
O.o_custkey
```

Figure 8. Query Execution Panel



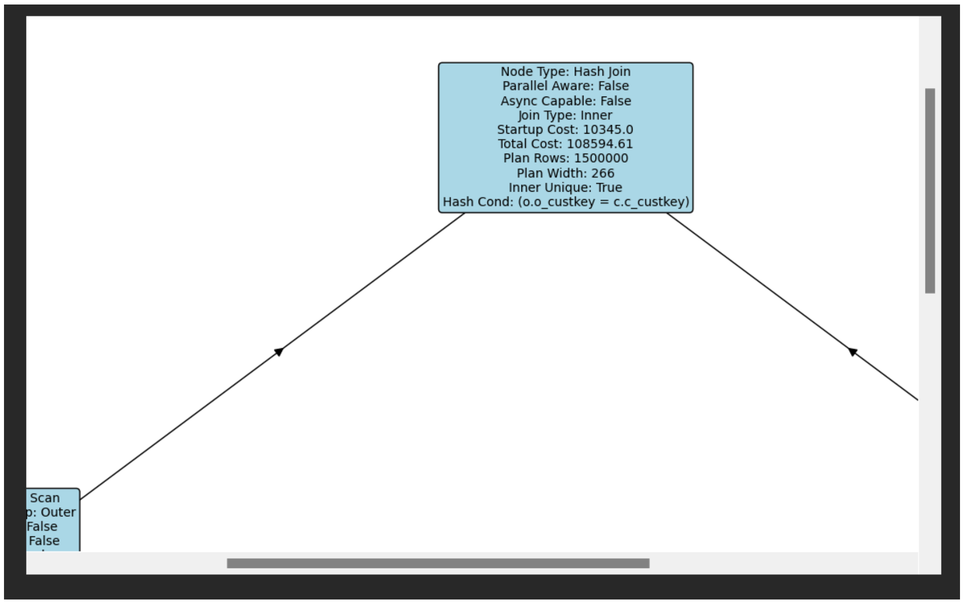
```
Hash Join  (cost=10345.00..108594.61 rows=1500000 width=266)
  Hash Cond: (o.o_custkey = c.c_custkey)
  ->  Seq Scan on orders o  (cost=0.00..41136.00 rows=1500000 width=107)
  ->  Hash  (cost=5100.00..5100.00 rows=150000 width=159)
        ->  Seq Scan on customer c  (cost=0.00..5100.00 rows=150000 width=159)
```

Figure 9. Outputted QEP



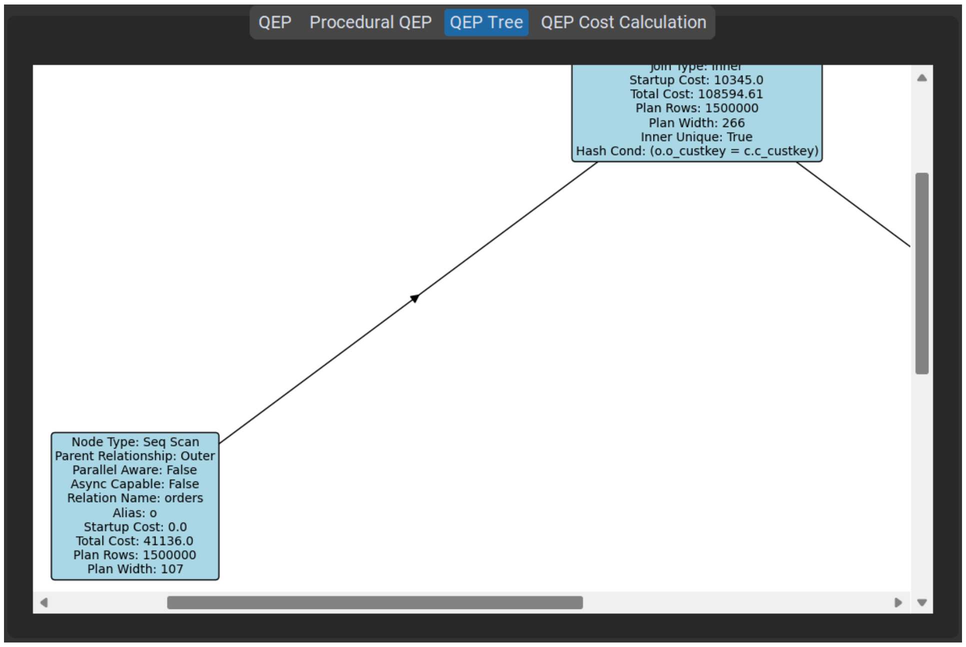
```
1. Seq Scan on orders (Child of Hash Join on customer and orders)
2. Seq Scan on customer (Child of Hash on customer)
3. Hash on customer (Child of Hash Join on customer and orders)
4. Hash Join on customer and orders
```

Figure 10. Query Execution Panel and outputted Procedural QEP



```
Node Type: Hash Join
Parallel Aware: False
Async Capable: False
Join Type: Inner
Startup Cost: 10345.0
Total Cost: 108594.61
Plan Rows: 1500000
Plan Width: 266
Inner Unique: True
Hash Cond: (o.o_custkey = c.c_custkey)
```

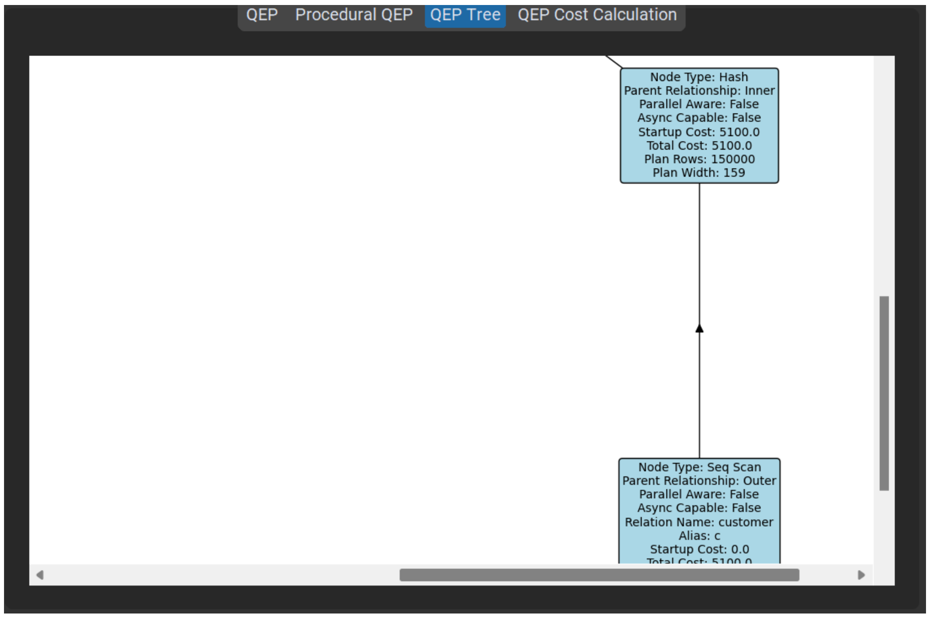
Figure 11. Root of the example QEP graph visualisation



Join Type: Inner
Startup Cost: 10345.0
Total Cost: 108594.61
Plan Rows: 1500000
Plan Width: 266
Inner Unique: True
Hash Cond: (o.o_custkey = c.c_custkey)

Node Type: Seq Scan
Parent Relationship: Outer
Parallel Aware: False
Async Capable: False
Relation Name: orders
Alias: o
Startup Cost: 0.0
Total Cost: 41136.0
Plan Rows: 1500000
Plan Width: 107

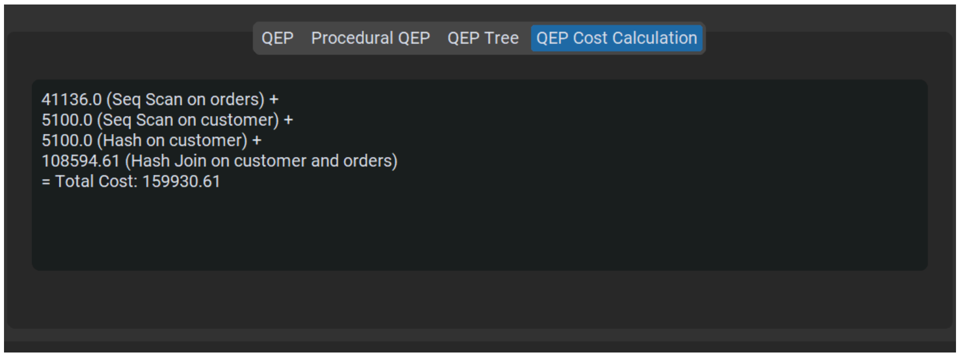
Figure 12. Left side of the example QEP graph visualisation



Node Type: Hash
Parent Relationship: Inner
Parallel Aware: False
Async Capable: False
Startup Cost: 5100.0
Total Cost: 5100.0
Plan Rows: 150000
Plan Width: 159

Node Type: Seq Scan
Parent Relationship: Outer
Parallel Aware: False
Async Capable: False
Relation Name: customer
Alias: c
Startup Cost: 0.0
Total Cost: 5100.0

Figure 13. Right side of the example QEP graph visualisation

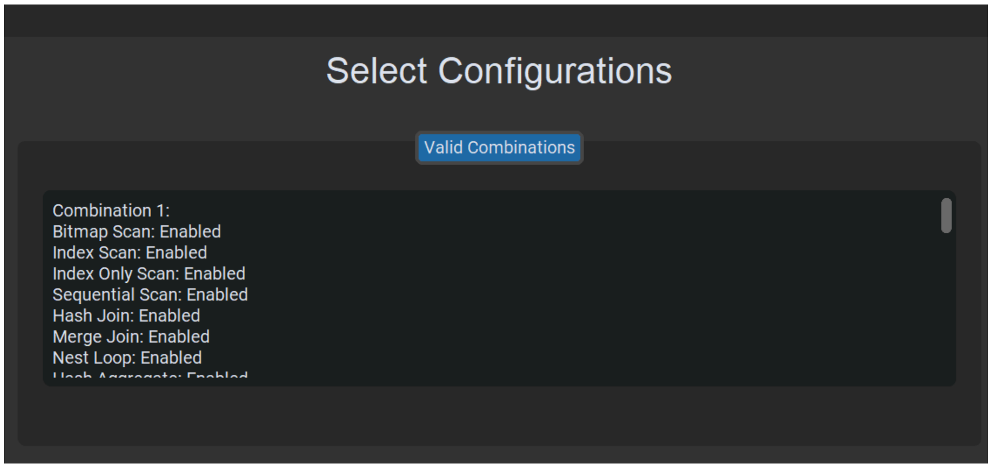


```
41136.0 (Seq Scan on orders) +
5100.0 (Seq Scan on customer) +
5100.0 (Hash on customer) +
108594.61 (Hash Join on customer and orders)
= Total Cost: 159930.61
```

Figure 14. Query Execution Panel and outputted QEP Cost Calculations

5.2.5 Configure settings Combinations

As for queries in postgreSQL, only certain combinations of scans, joins, aggregates, and sorts are allowed. In the panel below, users can check which combinations are permissible, and thus choose from the below combination list. (The query above has 9 total combinations)

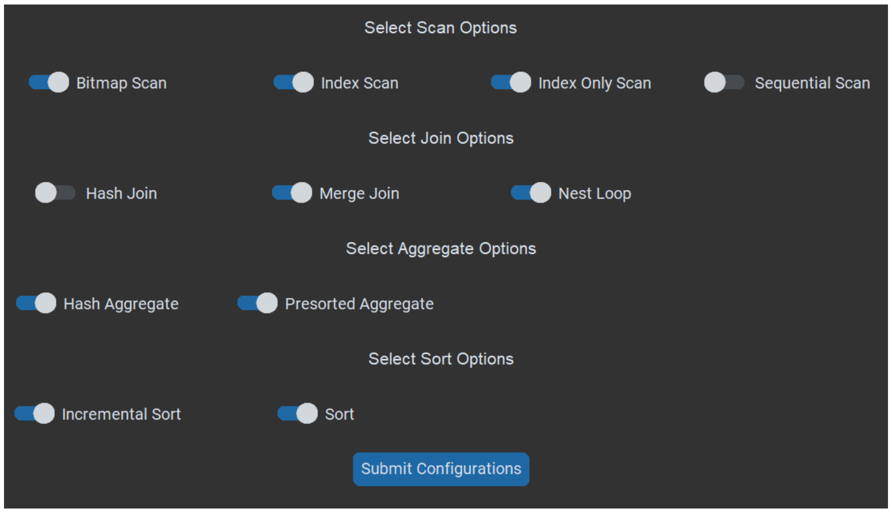


```
Combination 1:
Bitmap Scan: Enabled
Index Scan: Enabled
Index Only Scan: Enabled
Sequential Scan: Enabled
Hash Join: Enabled
Merge Join: Enabled
Nest Loop: Enabled
```

Figure 15. Section Configurations Combinations panel

5.2.6 Configure settings

These settings allow the user to ask potential “What-If” questions. These toggles give the user the freedom to choose from a variety of Scans, Joins, Aggregates, and Sorts. The toggles for each option turn on/off the following variable. In the particular case below, every option is turned on, except the Sequential scan and the Hash join.


Select Scan Options

[✓ Bitmap Scan] [✓ Index Scan] [✓ Index Only Scan] Sequential Scan

Select Join Options

Hash Join [✓ Merge Join] [✓ Nest Loop]

Select Aggregate Options

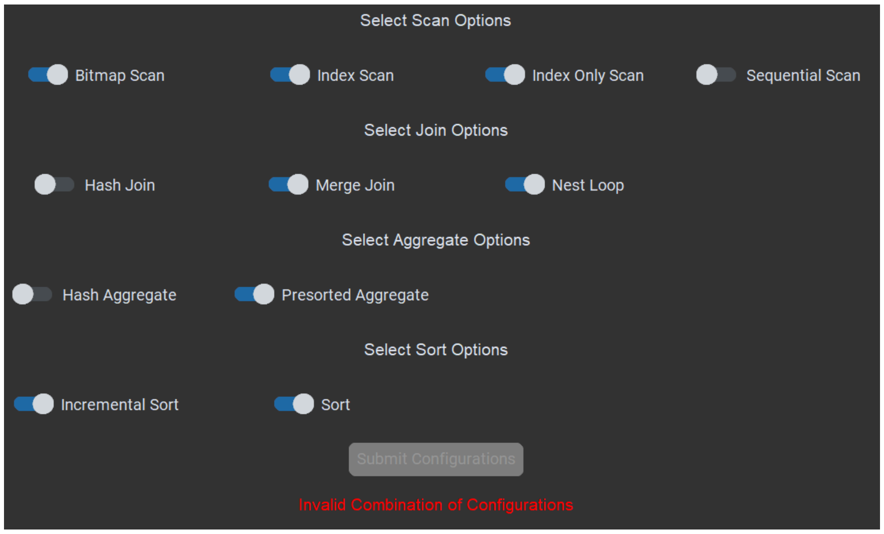
[✓ Hash Aggregate] [✓ Presorted Aggregate]

Select Sort Options

[✓ Incremental Sort] [✓ Sort]

Figure 16. Select Configurations Panel of Toggles

For invalid combinations, the application will not allow the user to submit them, as shown below.


Select Scan Options

[✓ Bitmap Scan] [✓ Index Scan] [✓ Index Only Scan] Sequential Scan

Select Join Options

Hash Join [✓ Merge Join] [✓ Nest Loop]

Select Aggregate Options

Hash Aggregate [✓ Presorted Aggregate]

Select Sort Options

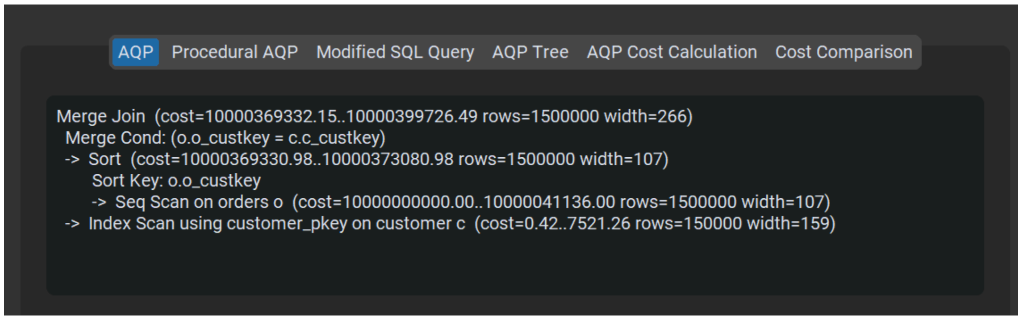
[✓ Incremental Sort] [✓ Sort]

Invalid Combination of Configurations

Figure 17. Invalid Configurations Panel of Toggles

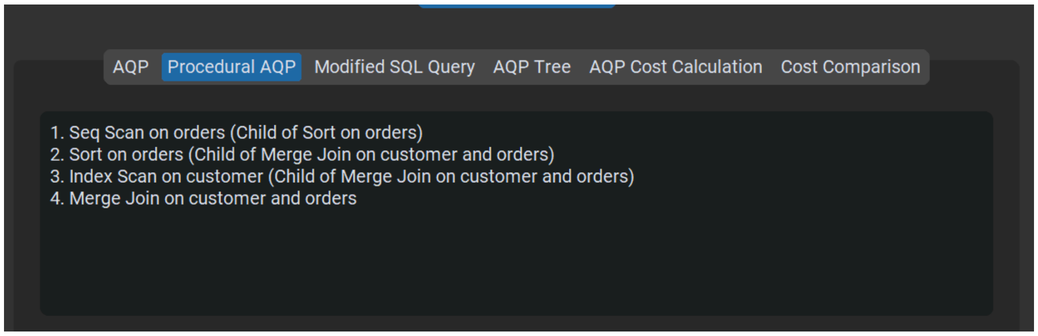
5.2.7 Alternate Execution Plan Panel

Upon clicking the “Submit Configurations” button, the application will then proceed to generate the Alternative Query Plan (AQP). The application will return the following Alternative Query Plan, the Procedural AQP, the Modified SQL Query to generate the AQP, the AQP Cost Calculation, the Cost Comparison between the original QEP and this AQP, and finally the visualisation of the AQP tree. The below AQP is a direct result of the example QEP and the above valid configured settings.



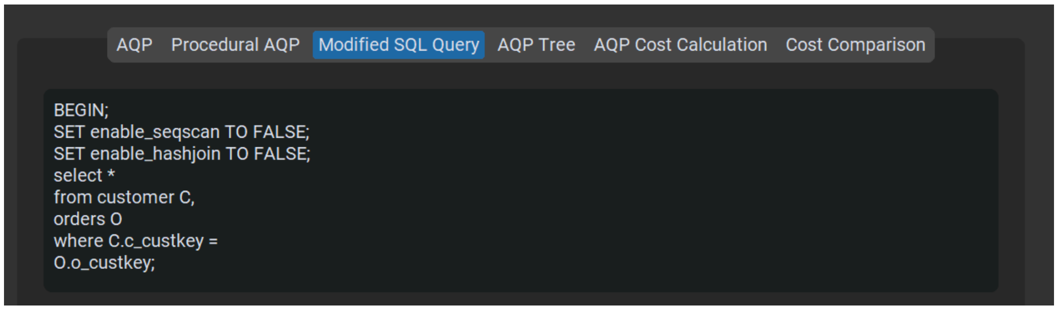
```
Merge Join  (cost=10000369332.15..10000399726.49 rows=1500000 width=266)
  Merge Cond: (o.o_custkey = c.c_custkey)
  ->  Sort  (cost=10000369330.98..10000373080.98 rows=1500000 width=107)
        Sort Key: o.o_custkey
        ->  Seq Scan on orders o  (cost=10000000000.00..10000041136.00 rows=1500000 width=107)
  ->  Index Scan using customer_pkey on customer c  (cost=0.42..7521.26 rows=150000 width=159)
```

Figure 18. AQP from the AQP Panel



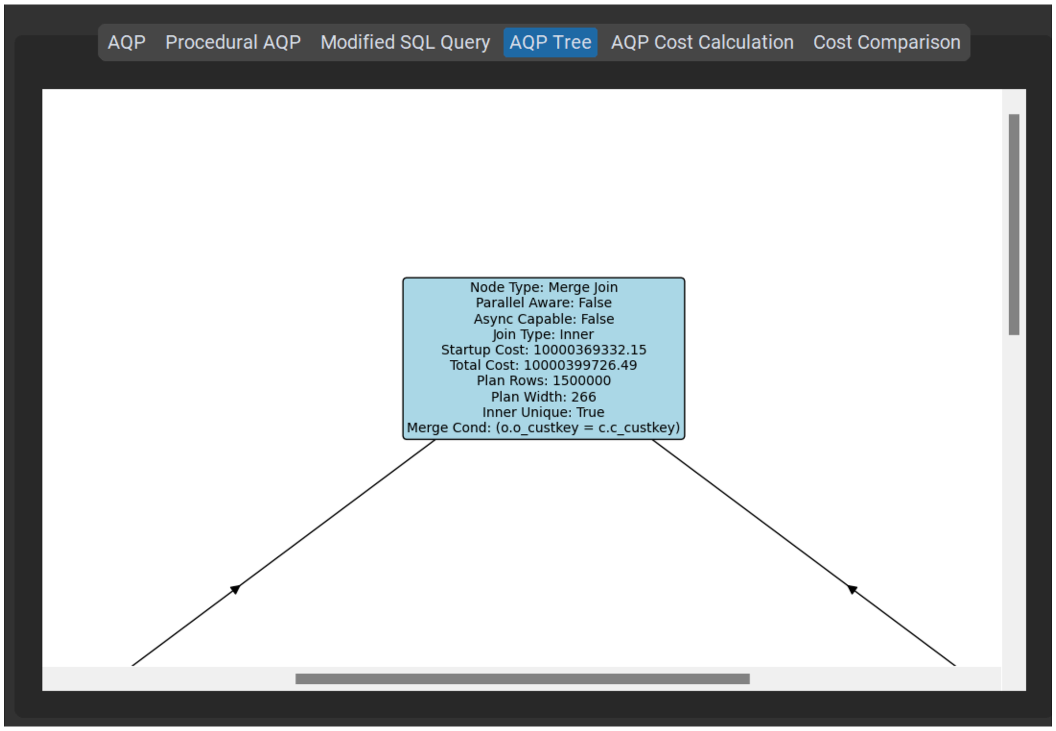
1. Seq Scan on orders (Child of Sort on orders)
2. Sort on orders (Child of Merge Join on customer and orders)
3. Index Scan on customer (Child of Merge Join on customer and orders)
4. Merge Join on customer and orders

Figure 19. Procedural AQP



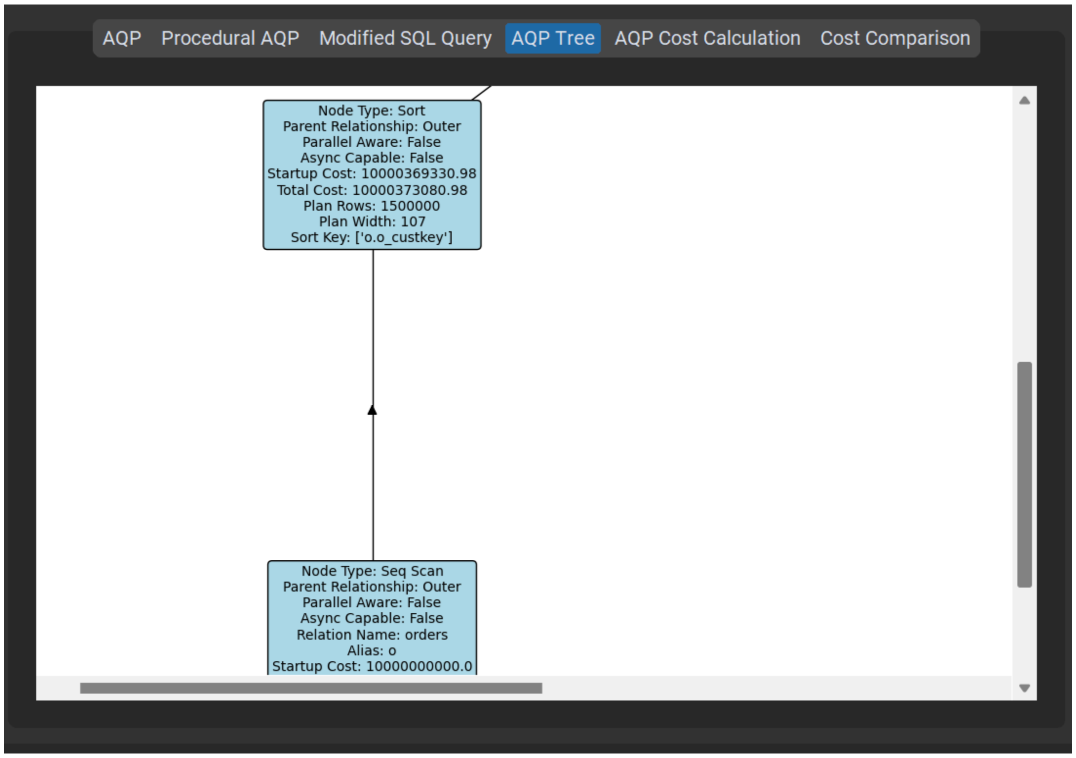
```
BEGIN;
SET enable_seqscan TO FALSE;
SET enable_hashjoin TO FALSE;
select *
from customer C,
orders O
where C.c_custkey =
O.o_custkey;
```

Figure 20. Modified SQL Query



Node Type: Merge Join
Parallel Aware: False
Async Capable: False
Join Type: Inner
Startup Cost: 10000369332.15
Total Cost: 10000399726.49
Plan Rows: 1500000
Plan Width: 266
Inner Unique: True
Merge Cond: (o.o_custkey = c.c_custkey)

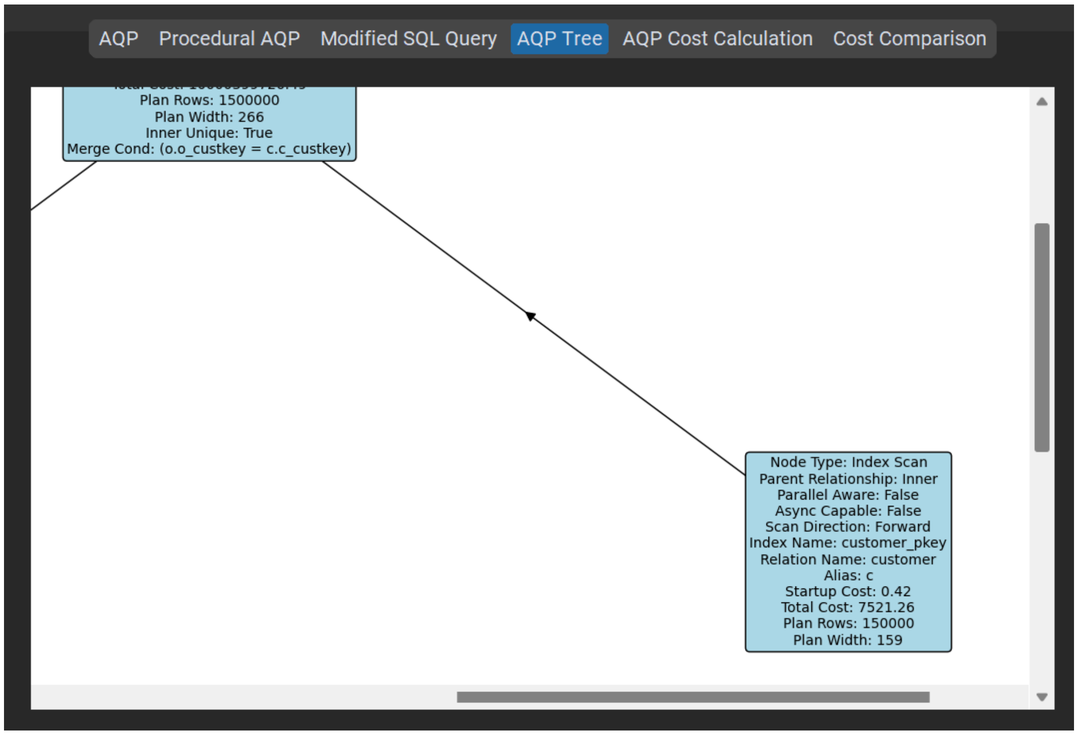
Figure 21. Root of the example AQP graph visualisation



Node Type: Sort
Parent Relationship: Outer
Parallel Aware: False
Async Capable: False
Startup Cost: 10000369330.98
Total Cost: 10000373080.98
Plan Rows: 1500000
Plan Width: 107
Sort Key: ['o.o_custkey']

Node Type: Seq Scan
Parent Relationship: Outer
Parallel Aware: False
Async Capable: False
Relation Name: orders
Alias: o
Startup Cost: 10000000000.0

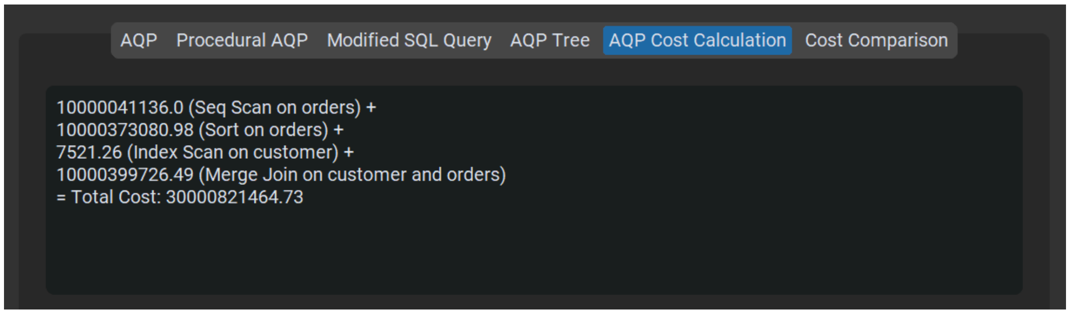
Figure 22. Left side of the example AQP graph visualisation



```
Plan Rows: 1500000
Plan Width: 266
Inner Unique: True
Merge Cond: (o.o_custkey = c.c_custkey)
```

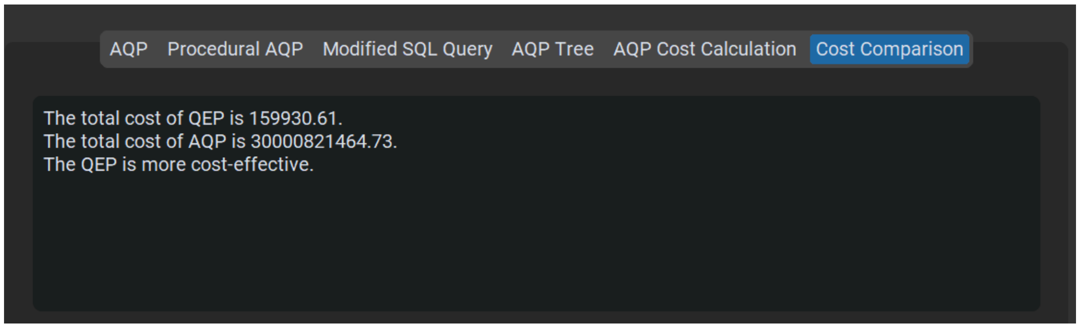
```
Node Type: Index Scan
Parent Relationship: Inner
Parallel Aware: False
Async Capable: False
Scan Direction: Forward
Index Name: customer_pkey
Relation Name: customer
Alias: c
Startup Cost: 0.42
Total Cost: 7521.26
Plan Rows: 150000
Plan Width: 159
```

Figure 22. Right side of the example AQP graph visualisation



```
10000041136.0 (Seq Scan on orders) +
10000373080.98 (Sort on orders) +
7521.26 (Index Scan on customer) +
10000399726.49 (Merge Join on customer and orders)
= Total Cost: 30000821464.73
```

Figure 23. AQP Cost Calculations



```
The total cost of QEP is 159930.61.
The total cost of AQP is 30000821464.73.
The QEP is more cost-effective.
```

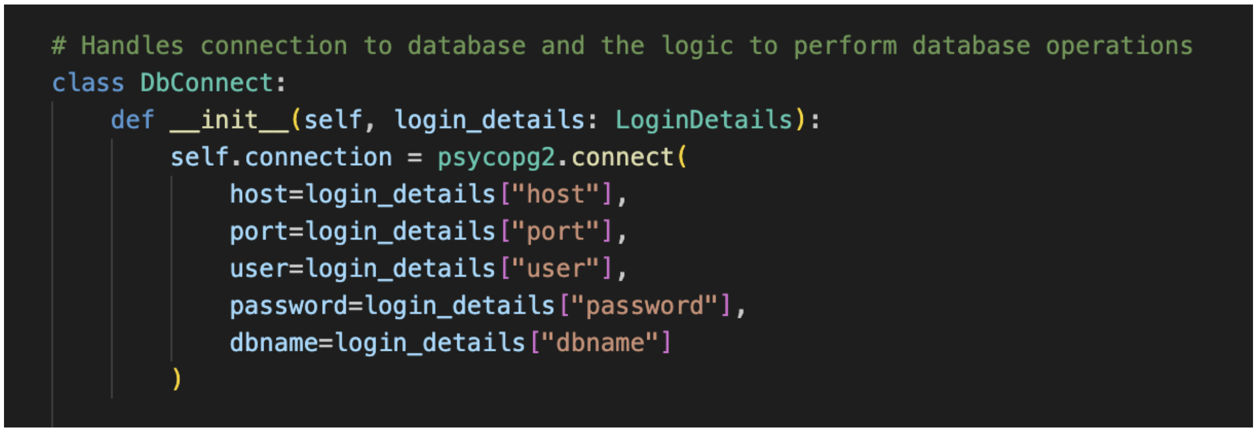
Figure 24. Cost Comparison between QEP and AQP

6. preprocessing.py

The preprocessing.py file contains a comprehensive suite of functions and classes required to prepare and manage data for the application's core functionalities. This section discusses the key components provided in this file.

6.1 Connecting to PostgreSQL Database

To initialise the application, we implemented a DBConnect class that establishes and manages a secure connection to the PostgreSQL database using the psycopg2 library. The class is instantiated with the login credentials encapsulated in a TypedDict (LoginDetails) that enforces type safety for parameters like host, user, password, database name, and port.

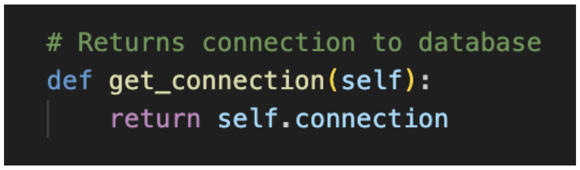


```
# Handles connection to database and the logic to perform database operations
class DbConnect:
    def __init__(self, login_details: LoginDetails):
        self.connection = psycopg2.connect(
            host=login_details["host"],
            port=login_details["port"],
            user=login_details["user"],
            password=login_details["password"],
            dbname=login_details["dbname"]
        )
```

Figure 25. Implementation of DBConnect class for establishing secure connection to the PostgreSQL database using provided credentials

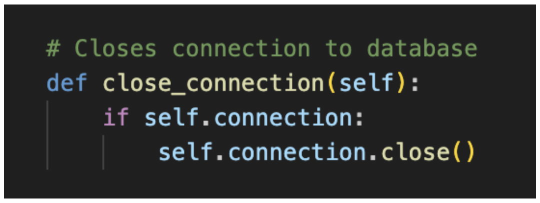
It maintains an active connection and provides methods to:

- Establish and return the connection
- Close the connection cleanly
- Retrieve table structure and related metadata, supporting schema exploration.



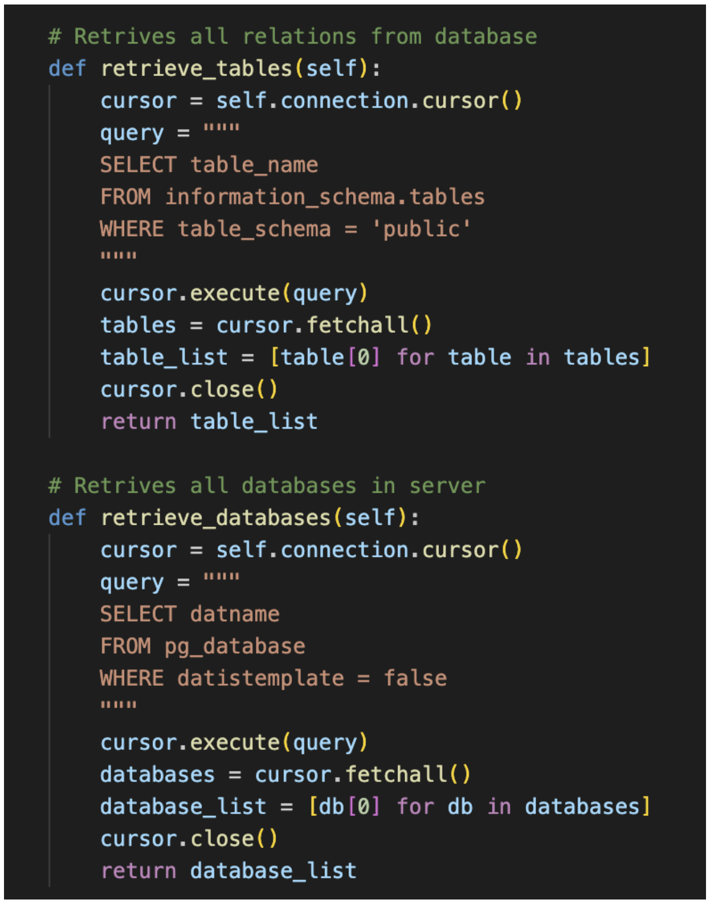
```
# Returns connection to database
def get_connection(self):
    return self.connection
```

Figure 26: Method get_connection() to establish and return the connection



```python
# Closes connection to database
def close_connection(self):
    if self.connection:
        self.connection.close()
```

Figure 27: Method close_connection() to close a connection if it exists.

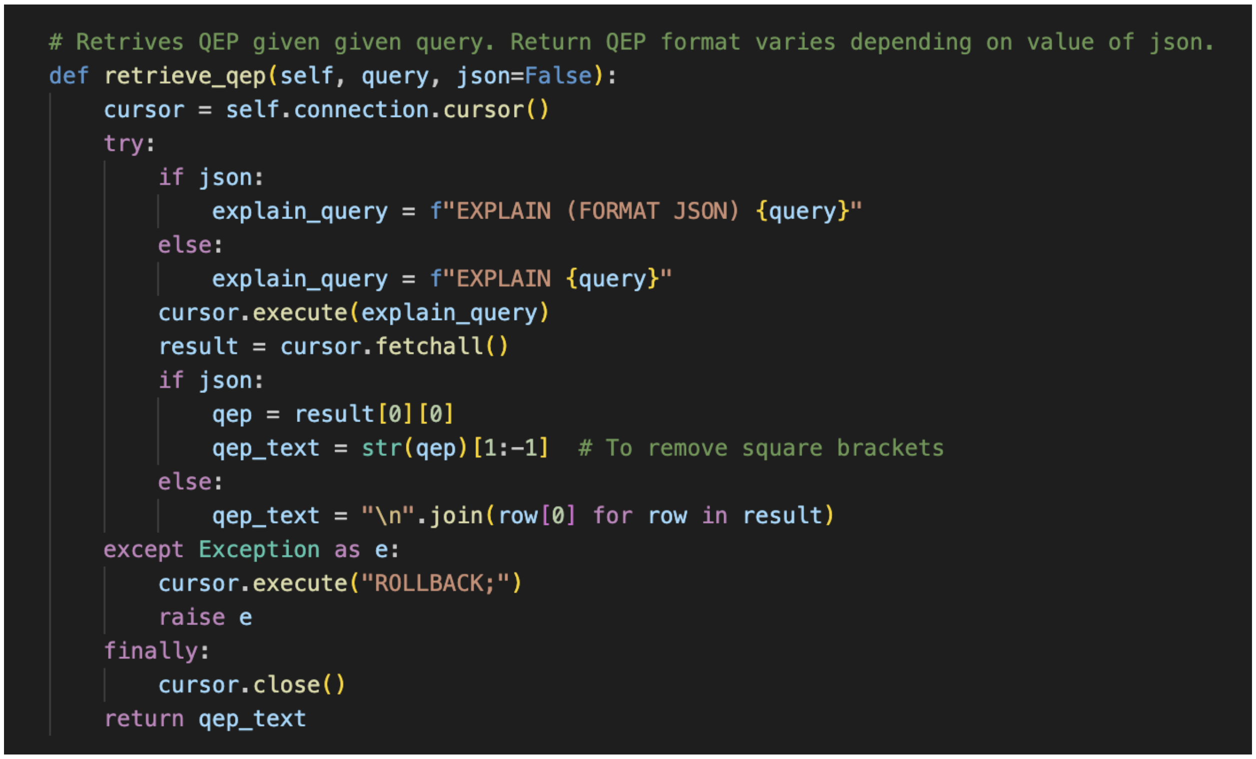


```python
# Retrives all relations from database
def retrieve_tables(self):
    cursor = self.connection.cursor()
    query = """
    SELECT table_name
    FROM information_schema.tables
    WHERE table_schema = 'public'
    """
    cursor.execute(query)
    tables = cursor.fetchall()
    table_list = [table[0] for table in tables]
    cursor.close()
    return table_list

# Retrives all databases in server
def retrieve_databases(self):
    cursor = self.connection.cursor()
    query = """
    SELECT datname
    FROM pg_database
    WHERE datistemplate = false
    """
    cursor.execute(query)
    databases = cursor.fetchall()
    database_list = [db[0] for db in databases]
    cursor.close()
    return database_list
```

Figure 28: Some of the methods used to gather and return metadata related to the database.

6.2 Query Execution Plan (QEP) Parsing and Analysis

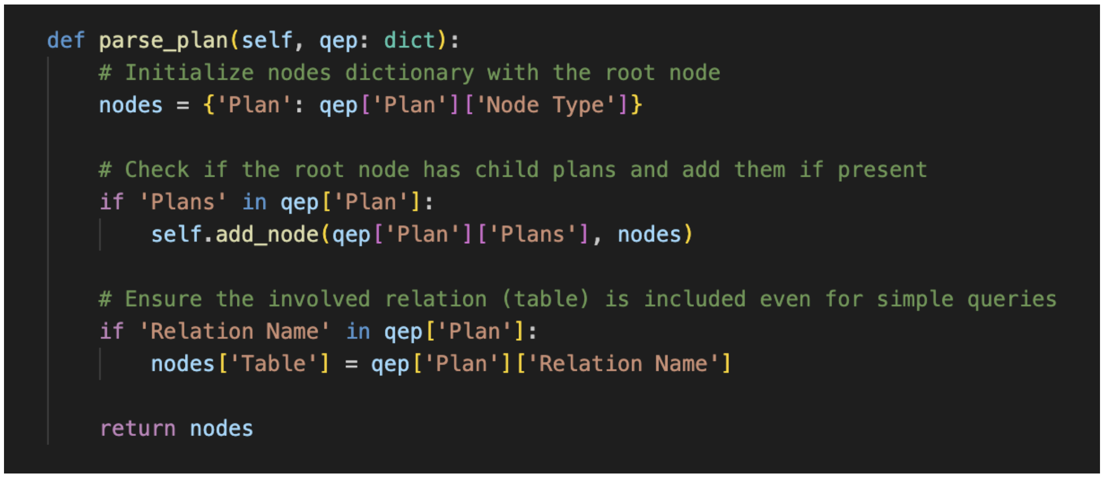
For each SQL query, our application retrieves a Query Execution Plan (QEP) in JSON format from PostgreSQL, which provides a detailed tree structure of the query's execution path and associated costs, including node types and dependencies.



```python
# Retrives QEP given given query. Return QEP format varies depending on value of json.
def retrieve_qep(self, query, json=False):
    cursor = self.connection.cursor()
    try:
        if json:
            explain_query = f"EXPLAIN (FORMAT JSON) {query}"
        else:
            explain_query = f"EXPLAIN {query}"
        cursor.execute(explain_query)
        result = cursor.fetchall()
        if json:
            qep = result[0][0]
            qep_text = str(qep)[1:-1]  # To remove square brackets
        else:
            qep_text = "\n".join(row[0] for row in result)
    except Exception as e:
        cursor.execute("ROLLBACK;")
        raise e
    finally:
        cursor.close()
    return qep_text
```

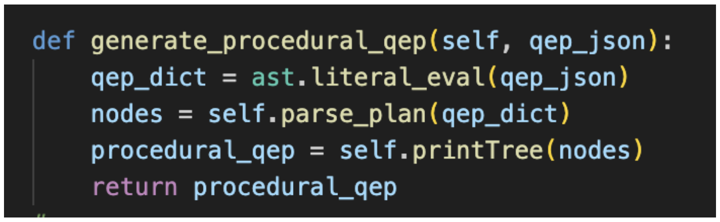
Figure 29: Retrieval of QEP in JSON format

The preprocessing.py module parses this JSON response and extracts relevant data points – such as scan types, join types, and their respective costs – structuring them for visualisation and comparison. Parsing is crucial to transforming the QEP data into a format that can be easily manipulated, displayed, and analysed by the application's other components.



```python
def parse_plan(self, qep: dict):
    # Initialize nodes dictionary with the root node
    nodes = {'Plan': qep['Plan']['Node Type']}

    # Check if the root node has child plans and add them if present
    if 'Plans' in qep['Plan']:
        self.add_node(qep['Plan']['Plans'], nodes)

    # Ensure the involved relation (table) is included even for simple queries
    if 'Relation Name' in qep['Plan']:
        nodes['Table'] = qep['Plan']['Relation Name']

    return nodes
```

Figure 30: Parsing of QEP JSON to extract relevant data



```python
def generate_procedural_qep(self, qep_json):
    qep_dict = ast.literal_eval(qep_json)
    nodes = self.parse_plan(qep_dict)
    procedural_qep = self.printTree(nodes)
    return procedural_qep
```

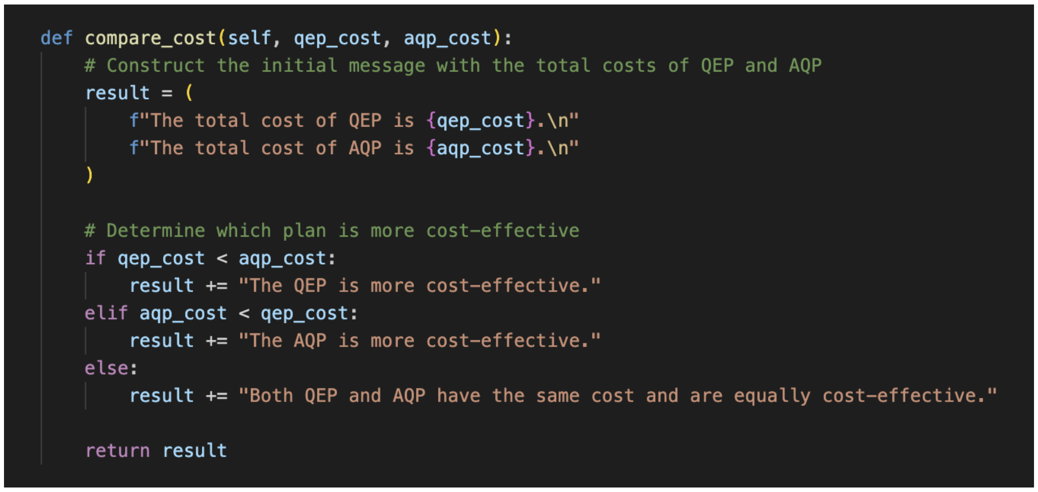
Figure 31: Generation of procedural QEP for visualisation and comparison

6.3 What-If Analysis and Configuration Management

The preprocessing module also manages what-if configurations, which allow users to simulate different query execution scenarios. When users choose to enable or disable specific settings such as index scans or hash joins, these preferences are parsed and applied dynamically by the module to create Alternative Query Execution Plans (AQPs). The preprocessing module ensures these configurations are communicated efficiently to PostgreSQL's query planner, allowing users to observe the cost and performance implications of hypothetical changes.

6.4 AQP Generation and Comparision Preparation

When generating diverse AQPs based on user configurations, the preprocessing module applies a systematic approach to toggle query planner settings in PostgreSQL. It tests various combinations of scan and join types, ensuring the generation of diverse execution plans while minimising redundancy. Each resulting AQP is stored and made available for comparison with the original QEP, enabling users to better understand cost trade-offs and execution dynamics.

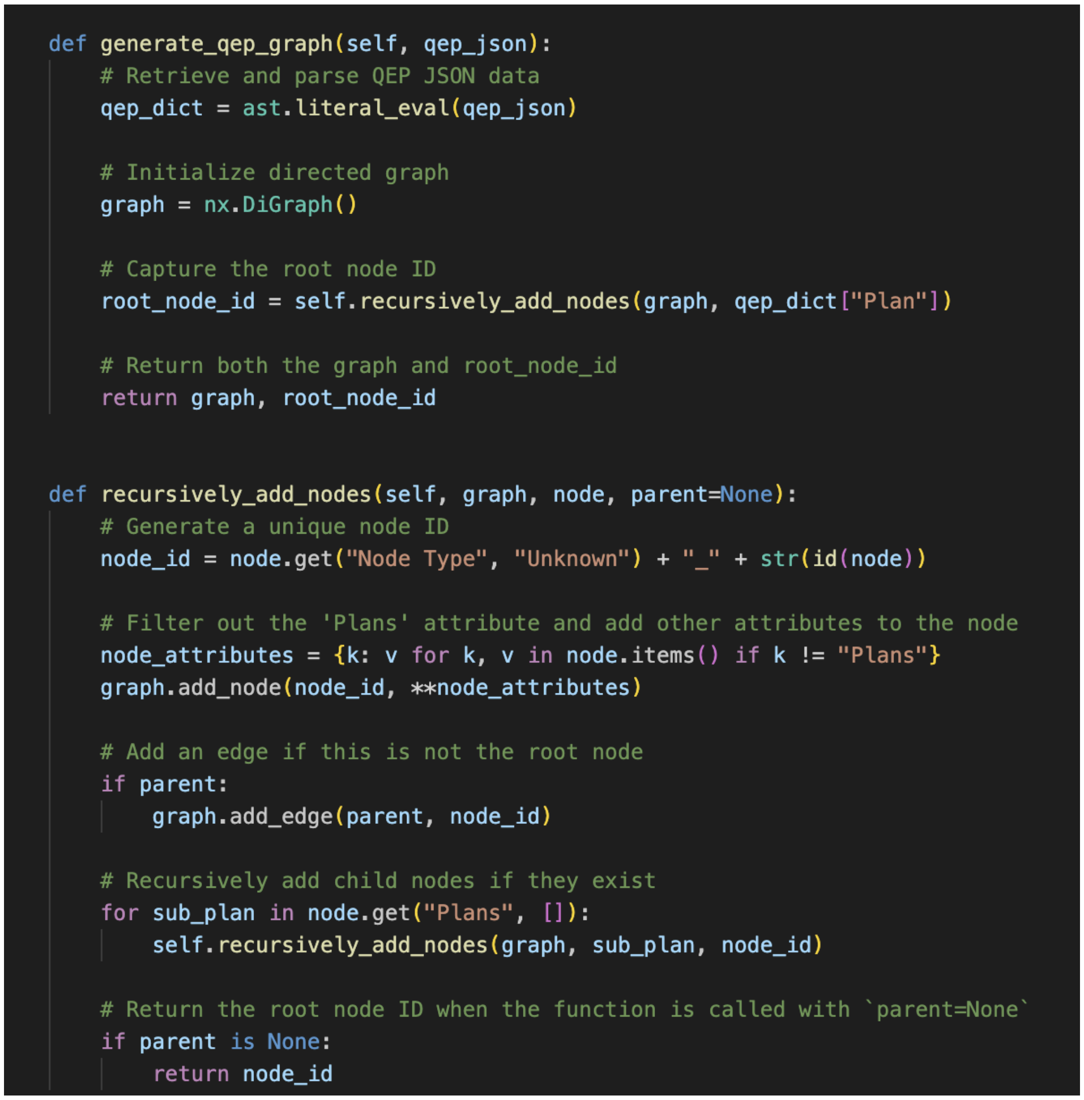


```python
def compare_cost(self, qep_cost, aqp_cost):
    # Construct the initial message with the total costs of QEP and AQP
    result = (
        f"The total cost of QEP is {qep_cost}.\n"
        f"The total cost of AQP is {aqp_cost}.\n"
    )

    # Determine which plan is more cost-effective
    if qep_cost < aqp_cost:
        result += "The QEP is more cost-effective."
    elif aqp_cost < qep_cost:
        result += "The AQP is more cost-effective."
    else:
        result += "Both QEP and AQP have the same cost and are equally cost-effective."

    return result
```

Figure 32: Method compare_cost() to compare cost of original QEP with AQP

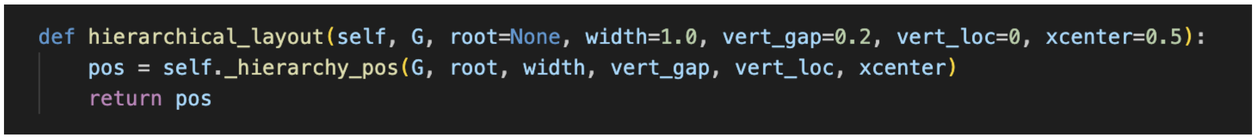
6.5 Data Preparation for Visualisation

To create effective visualisations of the QEP and AQP structures, the preprocessing module processes the parsed QEP data, translating it into a format compatible with the NetworkX library. This formatted data allows NetworkX to create tree structures of QEPs and AQPs, highlighting differences between the original and modified query plans. These visual aids help users to compare original and alternative execution paths, as well as visualise the impact of hypothetical changes on query execution.



```python
def generate_qep_graph(self, qep_json):
    # Retrieve and parse QEP JSON data
    qep_dict = ast.literal_eval(qep_json)

    # Initialize directed graph
    graph = nx.DiGraph()

    # Capture the root node ID
    root_node_id = self.recursively_add_nodes(graph, qep_dict["Plan"])

    # Return both the graph and root_node_id
    return graph, root_node_id


def recursively_add_nodes(self, graph, node, parent=None):
    # Generate a unique node ID
    node_id = node.get("Node Type", "Unknown") + "_" + str(id(node))

    # Filter out the 'Plans' attribute and add other attributes to the node
    node_attributes = {k: v for k, v in node.items() if k != "Plans"}
    graph.add_node(node_id, **node_attributes)

    # Add an edge if this is not the root node
    if parent:
        graph.add_edge(parent, node_id)

    # Recursively add child nodes if they exist
    for sub_plan in node.get("Plans", []):
        self.recursively_add_nodes(graph, sub_plan, node_id)

    # Return the root node ID when the function is called with `parent=None`
    if parent is None:
        return node_id
```

Figure 33: Method generate_qep_graph() to create the tree structure by calling recursively_add_nodes() to add nodes and edges



```python
def hierarchical_layout(self, G, root=None, width=1.0, vert_gap=0.2, vert_loc=0, xcenter=0.5):
    pos = self._hierarchy_pos(G, root, width, vert_gap, vert_loc, xcenter)
    return pos
```

Figure 34: Method hierarchical_layout() to determine placement of nodes in graph

6.6 Additional Features

1) The preprocessing.py script is designed with features that ensure reliability, maintainability, and clarity.

2) Error handling mechanisms have been implemented to manage potential issues during database operations and JSON parsing, ensuring the application operates smoothly under various conditions.

3) The code follows a modular design, with well-segmented functions that promote reusability and simplify maintenance.

4) Detailed inline comments provide comprehensive documentation for each method, aiding understanding and collaboration.

These form a solid foundation for the application's data preprocessing requirements.

7. whatif.py

This is the file containing the code of the following procedures relating to what-ifs:

- Generating all the possible AQPs for the inputted query to help generate the appropriate what-if questions for user
- Retrieving and parsing valid configuration combinations
- Generating query and AQP from user's what-ifs

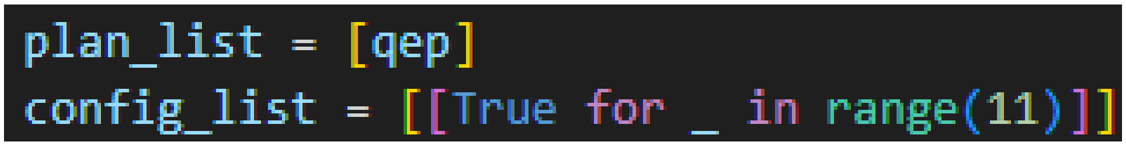
7.1 Retrieving all AQPs

To even generate what-if queries, we need to know what kind of configurations are possible for a certain query and so we set to do this by manipulating the following settings of planner method configuration in PostgreSQL: enable_bitmapscan, enable_indexscan, enable_indexonlyscan, enable_seqscan, enable_hashjoin, enable_mergejoin, enable_nestloop, enable_hashagg, enable_presorted_aggregate, enable_incremental_sort, enable_sort.

Other possible configurations are not considered because they are either optimisations for a sort, join, scan step in the execution plan or they are not useful in the context of this application running on a standard configuration of Postgres via Python.

The algorithm to get all AQPs is optimised in a logical way to have relatively quick computation by trying to cut out unnecessary configuration combinations and at the same time in most cases still should not miss out any possible AQP with different scans, sorts and join combinations.

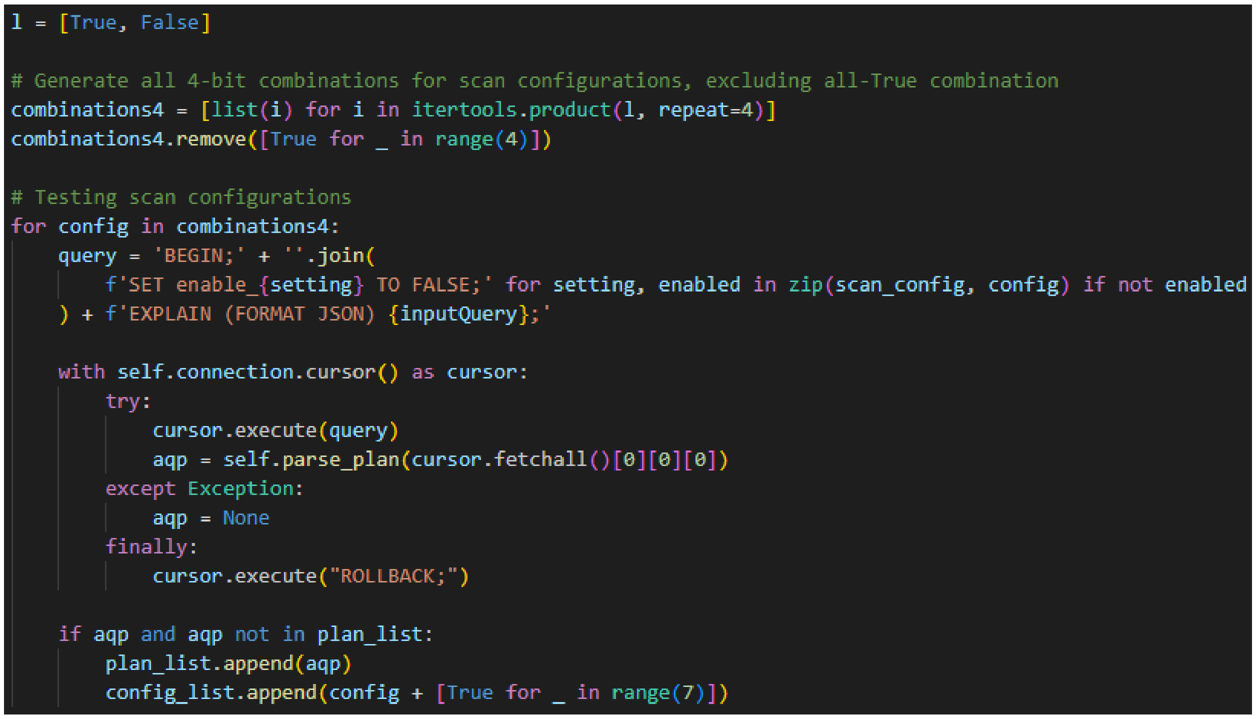
The algorithm initialises with plan_list with the default given QEP from the EXPLAIN query in PostgreSQL in the list and config_list with all 11 configurations of planner method configuration being True.



```
plan_list = [qep]
config_list = [[True for _ in range(11)]]
```

Figure 35. Initialisation of algorithm

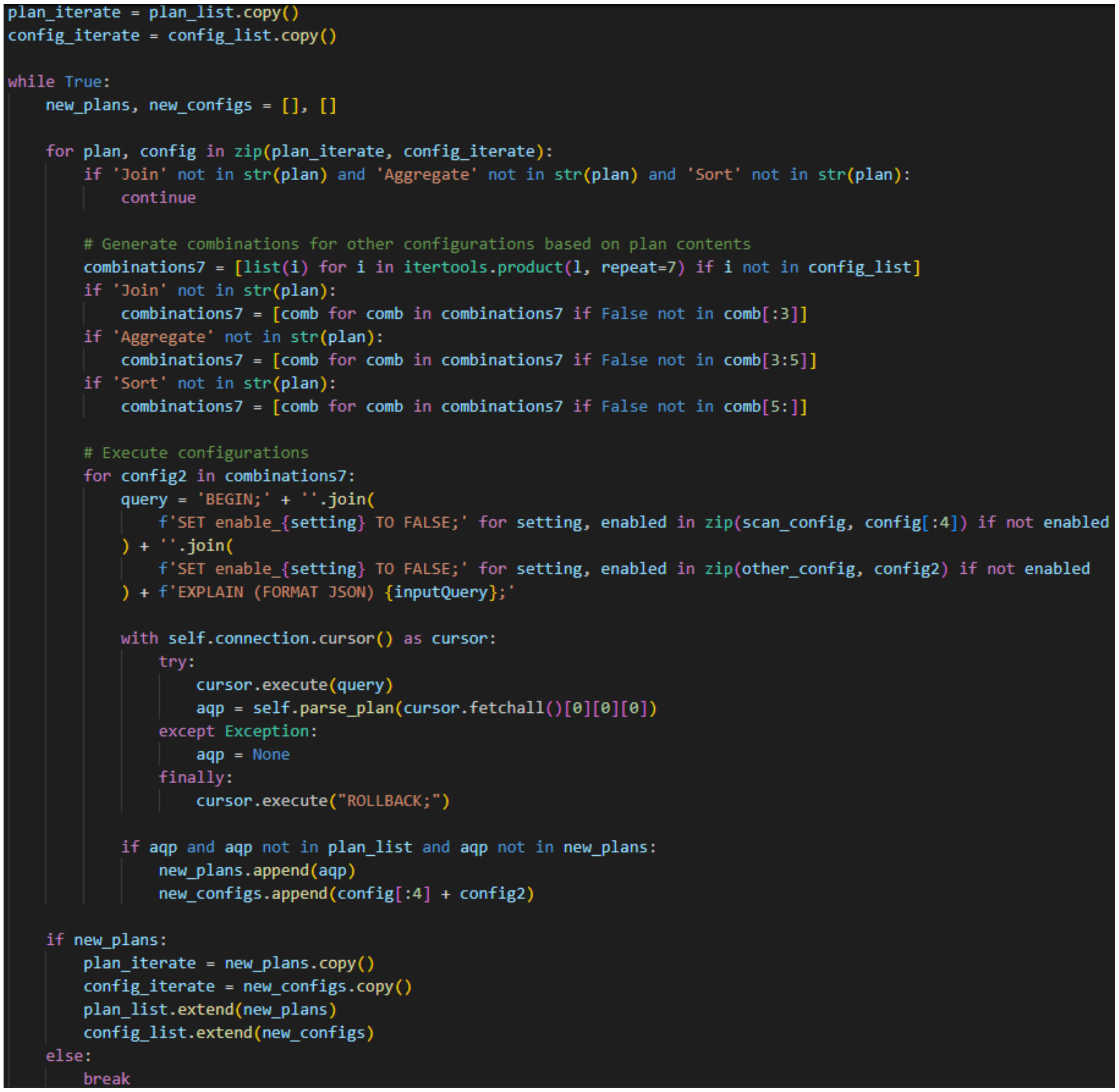
Then, it will gather all the possible AQPs from changing the 4 scan configurations since scans will always be at the root of the execution plan tree. This is done by generating every possible 4-bit combination of True/False excluding the all-True combination where each True/False element corresponds to the enabling/disabling of a configuration and then testing for each combination for a new plan. If a new plan is discovered, it will then be added to plan_list and the corresponding configuration will be added to config_list.



```python
l = [True, False]

# Generate all 4-bit combinations for scan configurations, excluding all-True combination
combinations4 = [list(i) for i in itertools.product(l, repeat=4)]
combinations4.remove([True for _ in range(4)])

# Testing scan configurations
for config in combinations4:
    query = 'BEGIN;' + ''.join(
        f'SET enable_{setting} TO FALSE;' for setting, enabled in zip(scan_config, config) if not enabled
    ) + f'EXPLAIN (FORMAT JSON) {inputQuery};'

    with self.connection.cursor() as cursor:
        try:
            cursor.execute(query)
            aqp = self.parse_plan(cursor.fetchall()[0][0][0])
        except Exception:
            aqp = None
        finally:
            cursor.execute("ROLLBACK;")

    if aqp and aqp not in plan_list:
        plan_list.append(aqp)
        config_list.append(config + [True for _ in range(7)])
```

Figure 36. Code for obtaining plans from different combinations of scan configurations

Finally, it will proceed to iterate through combinations of the other configurations by iterating through the plans that were initially obtained from scan configurations and then continuously iterating through new plans to check for any more new plans that can be produced. During each iteration of a plan-configuration pair, the combinations of configurations to check will be reduced according to types of operators missing from the iterated plan for efficiency purposes.



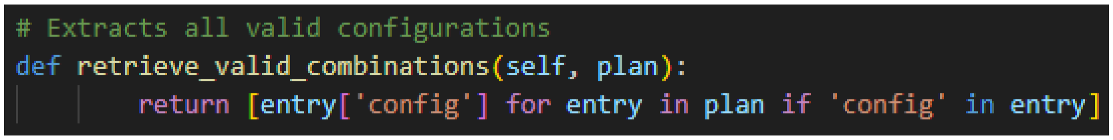
```python
plan_iterate = plan_list.copy()
config_iterate = config_list.copy()

while True:
    new_plans, new_configs = [], []

    for plan, config in zip(plan_iterate, config_iterate):
        if 'Join' not in str(plan) and 'Aggregate' not in str(plan) and 'Sort' not in str(plan):
            continue

        # Generate combinations for other configurations based on plan contents
        combinations7 = [list(i) for i in itertools.product(l, repeat=7) if i not in config_list]
        if 'Join' not in str(plan):
            combinations7 = [comb for comb in combinations7 if False not in comb[:3]]
        if 'Aggregate' not in str(plan):
            combinations7 = [comb for comb in combinations7 if False not in comb[3:5]]
        if 'Sort' not in str(plan):
            combinations7 = [comb for comb in combinations7 if False not in comb[5:]]

        # Execute configurations
        for config2 in combinations7:
            query = 'BEGIN;' + ''.join(
                f'SET enable_{setting} TO FALSE;' for setting, enabled in zip(scan_config, config[:4]) if not enabled
            ) + ''.join(
                f'SET enable_{setting} TO FALSE;' for setting, enabled in zip(other_config, config2) if not enabled
            ) + f'EXPLAIN (FORMAT JSON) {inputQuery};'

            with self.connection.cursor() as cursor:
                try:
                    cursor.execute(query)
                    aqp = self.parse_plan(cursor.fetchall()[0][0][0])
                except Exception:
                    aqp = None
                finally:
                    cursor.execute("ROLLBACK;")

            if aqp and aqp not in plan_list and aqp not in new_plans:
                new_plans.append(aqp)
                new_configs.append(config[:4] + config2)

    if new_plans:
        plan_iterate = new_plans.copy()
        config_iterate = new_configs.copy()
        plan_list.extend(new_plans)
        config_list.extend(new_configs)
    else:
        break
```

Figure 37. Code which tests for the other 7 configurations for new plans

7.2 Retrieving and parsing valid configurations

After getting all the distinct plans that can occur with a certain query, the application will then call upon these functions to then retrieve these valid combinations of configurations and then parses them for the AQP frame

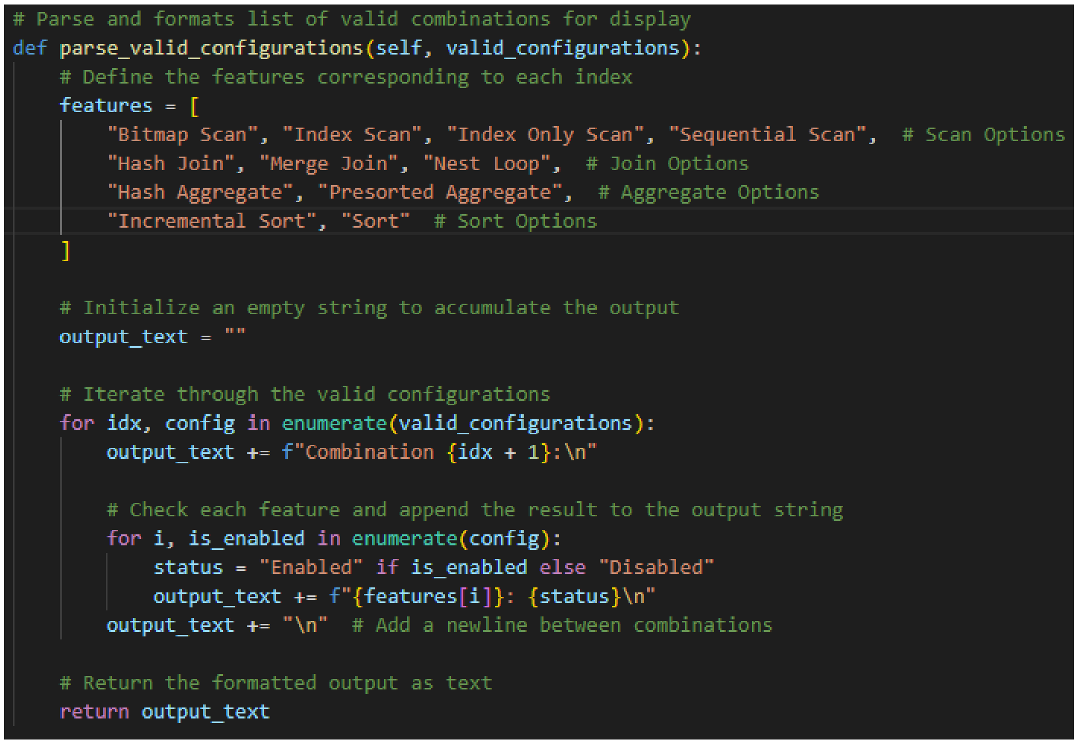
Retrieval of configurations just requires the list of plans to be passed into the function and then collect the list of configurations



```python
# Extracts all valid configurations
def retrieve_valid_combinations(self, plan):
        return [entry['config'] for entry in plan if 'config' in entry]
```

Figure 38. Code to retrieve valid configurations after getting list of all plans

Then the application proceeds to have these valid configurations parsed into the AQP frame of the interface. It takes in all the valid configurations of the specific query and parses the plans into an output text in a list format.

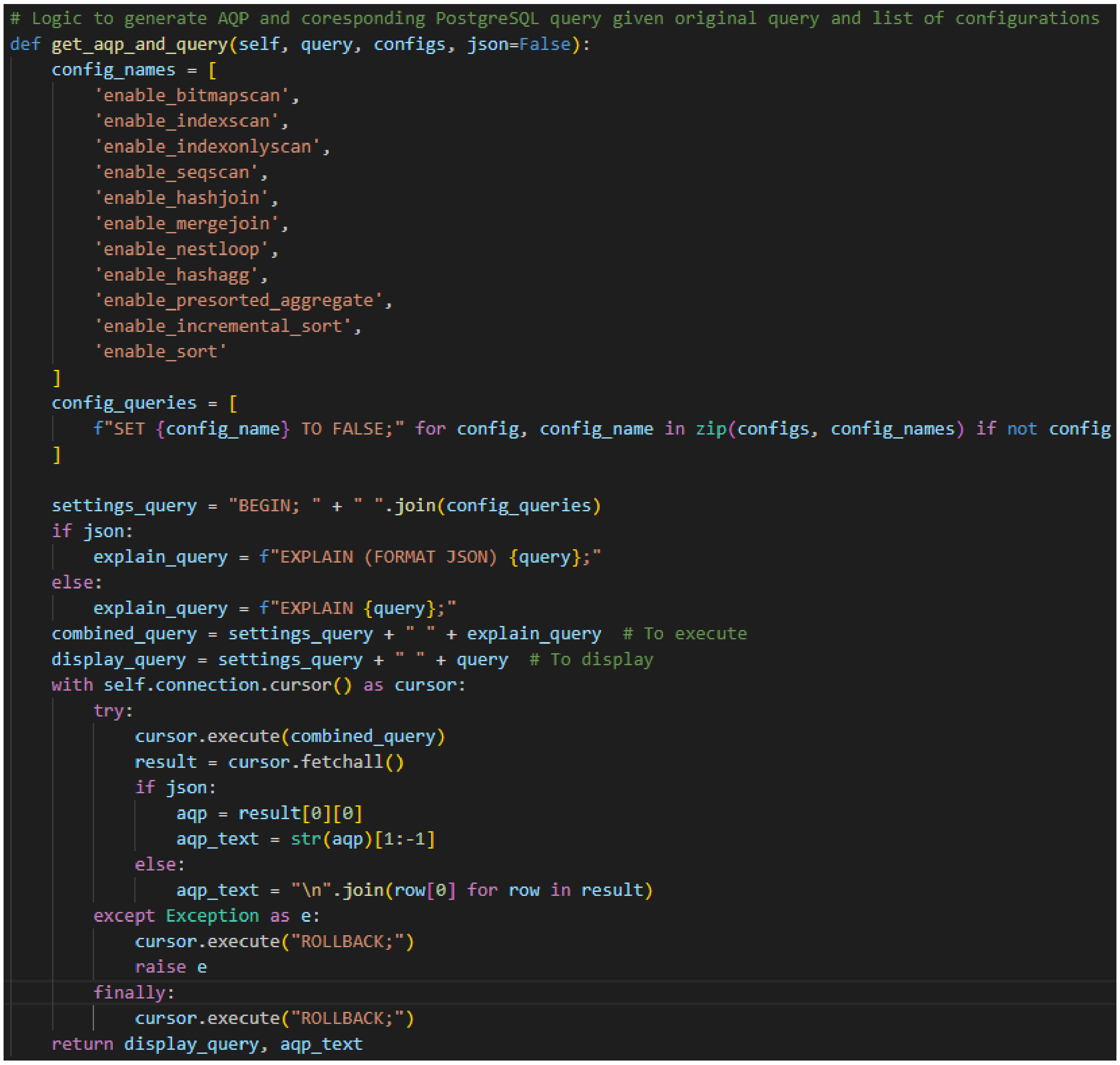


```python
# Parse and formats list of valid combinations for display
def parse_valid_configurations(self, valid_configurations):
    # Define the features corresponding to each index
    features = [
        "Bitmap Scan", "Index Scan", "Index Only Scan", "Sequential Scan",  # Scan Options
        "Hash Join", "Merge Join", "Nest Loop",  # Join Options
        "Hash Aggregate", "Presorted Aggregate",  # Aggregate Options
        "Incremental Sort", "Sort"  # Sort Options
    ]

    # Initialize an empty string to accumulate the output
    output_text = ""

    # Iterate through the valid configurations
    for idx, config in enumerate(valid_configurations):
        output_text += f"Combination {idx + 1}:\n"

        # Check each feature and append the result to the output string
        for i, is_enabled in enumerate(config):
            status = "Enabled" if is_enabled else "Disabled"
            output_text += f"{features[i]}: {status}\n"
        output_text += "\n"  # Add a newline between combinations

    # Return the formatted output as text
    return output_text
```

Figure 39. Code to parse valid configurations for AQP frame

7.3 Getting query and AQP from a set of what-ifs

After Users has enabled/disabled the desired configurations (the what-ifs), it is then passed into the below function where it will produce the appropriate PostgreSQL query for the configuration and its AQP



```python
# Logic to generate AQP and coresponding PostgreSQL query given original query and list of configurations
def get_aqp_and_query(self, query, configs, json=False):
    config_names = [
        'enable_bitmapscan',
        'enable_indexscan',
        'enable_indexonlyscan',
        'enable_seqscan',
        'enable_hashjoin',
        'enable_mergejoin',
        'enable_nestloop',
        'enable_hashagg',
        'enable_presorted_aggregate',
        'enable_incremental_sort',
        'enable_sort'
    ]
    config_queries = [
        f"SET {config_name} TO FALSE;" for config, config_name in zip(configs, config_names) if not config
    ]

    settings_query = "BEGIN; " + " ".join(config_queries)
    if json:
        explain_query = f"EXPLAIN (FORMAT JSON) {query};"
    else:
        explain_query = f"EXPLAIN {query};"
    combined_query = settings_query + " " + explain_query  # To execute
    display_query = settings_query + " " + query  # To display
    with self.connection.cursor() as cursor:
        try:
            cursor.execute(combined_query)
            result = cursor.fetchall()
            if json:
                aqp = result[0][0]
                aqp_text = str(aqp)[1:-1]
            else:
                aqp_text = "\n".join(row[0] for row in result)
        except Exception as e:
            cursor.execute("ROLLBACK;")
            raise e
        finally:
            cursor.execute("ROLLBACK;")
    return display_query, aqp_text
```

Figure 40. Code to produce the appropriate PostgreSQL query (using Planner Method Configuration) and the AQP associated with it

8. Limitations of Software

In PostgreSQL , the configuration settings related to query execution such as the enabling and disabling specific scan, join, aggregation and sort operations are treated as suggestions rather than strict directives. This means that the PostgreSQL planner can override these settings if it determines that other approaches are necessary or more optimal for query execution. The impact this has on our application is that sometimes even when applying the settings for different combinations of configurations, the resulting AQP will be the same as the original QEP.

Another limitation of our GUI is the requirement for the figure size of the matplotlib plot used to embed the QEP and AQP graph plots to be declared at the time of instantiation. To ensure the graphs generated fit in the matplotlib figure, we have implemented an algorithm to calculate the size of the plot required before creating the plot. This approach, however, may not work well for unusually large graphs that correspond to complex QEPs and AQPs.

Finally, unlike when using psql, psycopg2 does not allow for connection to a server without the specification of a database to connect to. To handle this, when a database is not specified by the user when logging in, we have configured psycopg2 to automatically connect to the default postgres database. This approach however only works if the user has not modified the name of the postgres database.

9. Example Query Inputs experimented

9.1 No input query at all

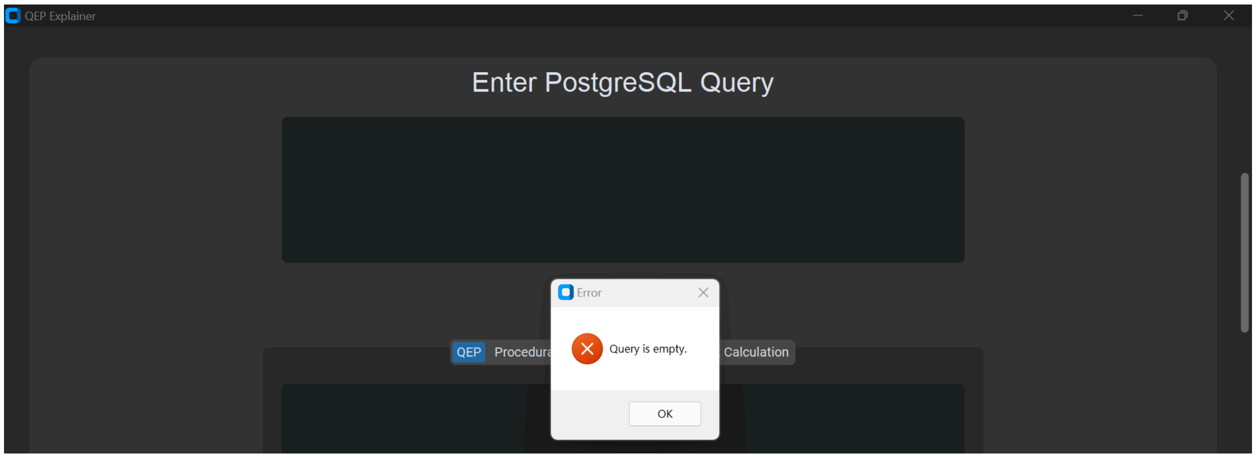


Figure. 41 “Query is empty” Pop-up

As you can see from the figure above, the above pop up will display if no input query is typed when the “Submit Query” button is pressed.

9.2 Query total quantity ordered for each part

The query below was used to find the total quantity ordered for each part:

```
SELECT p.p_partkey,
    SUM(l.l_quantity) AS total_quantity
FROM part p
JOIN lineitem l ON p.p_partkey = l.l_partkey
GROUP BY p.p_partkey;
```

Below are the results of inputting our query into our application:

Procedural QEP:

1. Seq Scan on lineitem (Child of Hash Join on part and lineitem)
2. Index Only Scan on part (Child of Hash on part)
3. Hash on part (Child of Hash Join on part and lineitem)
4. Hash Join on part and lineitem (Child of Aggregate on part and lineitem)
5. Aggregate on part and lineitem (Child of Sort on part and lineitem)
6. Sort on part and lineitem (Child of Gather Merge on part and lineitem)
7. Gather Merge on part and lineitem (Child of Aggregate on part and lineitem)
8. Aggregate on part and lineitem

Total Cost of QEP Calculation: **1810995.42**

Below are the results of the total cost of the AQP when only certain parameters are changed:

1) Only Index-only scan and Sequential scan are **turned off**

    The total cost of AQP is **15001567813.81.**

2) Only Hash Join and Merge Join are **turned off**

    The total cost of AQP is **5374893.57.**

3) Only Hash Aggregate is **turned off**

    The total cost of AQP is **2466477.52.**

4) Only Sort is **turned off**

The total cost of AQP is **1430816.38.** (The AQP actually beats the QEP here)

9.3 Query the total revenue for each customer

SELECT c.c_custkey,
    c.c_name **AS** customer_name,
    SUM(l.l_extendedprice * (1 - l.l_discount)) **AS** total_revenue
FROM customer c
JOIN orders o **ON** c.c_custkey = o.o_custkey
JOIN lineitem l **ON** o.o_orderkey = l.l_orderkey
GROUP BY c.c_custkey, c.c_name;

Below are the results of inputting our query into our application:

Procedural QEP:

1. Seq Scan on lineitem (Child of Hash Join on lineitem and orders)
2. Seq Scan on orders (Child of Hash on orders)
3. Hash on orders (Child of Hash Join on lineitem and orders)
4. Hash Join on lineitem and orders (Child of Hash Join on customer and lineitem and orders)
5. Seq Scan on customer (Child of Hash on customer)
6. Hash on customer (Child of Hash Join on customer and lineitem and orders)
7. Hash Join on customer and lineitem and orders (Child of Aggregate on customer and lineitem and orders)
8. Aggregate on customer and lineitem and orders (Child of Sort on customer and lineitem and orders)
9. Sort on customer and lineitem and orders (Child of Gather Merge on customer and lineitem and orders)
10. Gather Merge on customer and lineitem and orders (Child of Aggregate on customer and lineitem and orders)
11. Aggregate on customer and lineitem and orders

Total cost of QEP calculation: **2769972.4499999997**

Below are the results of the total cost of the AQP when only certain parameters are changed:

1) Only Bitmap scan, Index scan and Sequential scan are **turned off:**
   The total cost of AQP is **130005776323.05.** (QEP beats AQP)

2) Only Hash Join and Merge Join are **turned off:**
   The total cost of AQP is **5789070.29.** (QEP beats AQP)

3) Only Hash Aggregate is **turned off:**
   The total cost of AQP is **3666068.99.** (QEP beats AQP)

4) Only Sort is **turned off:**
   The total cost of AQP is **2243398.24.** (The AQP actually beats the QEP here)

9.4 Query the **top 5 suppliers** (based on their average supply cost) for each region, sorted by region name and average supply cost (in descending order).

```
SELECT
   r.r_name AS region,
   s.s_name AS supplier,
   AVG(ps.ps_supplycost) AS avg_supply_cost
FROM
   public.region r
JOIN public.nation n ON r.r_regionkey = n.n_regionkey
JOIN  public.supplier s ON n.n_nationkey = s.s_nationkey
JOIN  public.partsupp ps ON s.s_suppkey = ps.ps_suppkey
GROUP BY  r.r_name, s.s_name
ORDER BY  r.r_name, avg_supply_cost DESC
```

FETCH FIRST 5 ROWS ONLY;

Procedural QEP:

1. Seq Scan on partsupp (Child of Hash Join on partsupp and supplier)
2. Seq Scan on supplier (Child of Hash on supplier)
3. Hash on supplier (Child of Hash Join on partsupp and supplier)
4. Hash Join on partsupp and supplier (Child of Hash Join on nation and partsupp and supplier)
5. Seq Scan on nation (Child of Hash on nation)
6. Hash on nation (Child of Hash Join on nation and partsupp and supplier)
7. Hash Join on nation and partsupp and supplier (Child of Hash Join on nation and partsupp and supplier and region)
8. Seq Scan on region (Child of Hash on region)
9. Hash on region (Child of Hash Join on nation and partsupp and supplier and region)
10. Hash Join on nation and partsupp and supplier and region (Child of Sort on nation and partsupp and supplier and region)
11. Sort on nation and partsupp and supplier and region (Child of Aggregate on nation and partsupp and supplier and region)
12. Aggregate on nation and partsupp and supplier and region (Child of Gather Merge on nation and partsupp and supplier and region)
13. Gather Merge on nation and partsupp and supplier and region (Child of Aggregate on nation and partsupp and supplier and region)
14. Aggregate on nation and partsupp and supplier and region (Child of Incremental Sort on nation and partsupp and supplier and region)
15. Incremental Sort on nation and partsupp and supplier and region (Child of Limit on nation and partsupp and supplier and region)
16. Limit on nation and partsupp and supplier and region

Total cost of QEP calculation: **878373.8699999995**

Below are the results of the total cost of the AQP when only certain parameters are changed:

1) Only Index scan and Sequential scan are **turned off:**

   The total cost of AQP is **80001681916.91998.** (QEP beats AQP)

2) Only Hash Join is **turned off:**

   The total cost of AQP is **1079565.01.** (QEP beats AQP)

3) Only Incremental Sort is **turned off:**

   The total cost of AQP is **984100.8999999996.** (QEP beats AQP)