NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC3020: Database System Principles

Group 9

Project 2 Report

Name	Matriculation Number
Lee Hao Guang	U2122950A
Leong Hong Yi	U2120932C
Lim Jun Hern	U2120981B
Yap Xuan Ying	U2021297G

* Each group member contributed equally to the project

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

Table of Content

Introduction

Project Goal

This project is to design and implement a software that facilitates visual exploration of disk blocks accessed by a given input SQL query as well as various features of the corresponding QEP.

Installation Guide

Please refer to the **Installation Guide.pdf** found in the same folder.

File Structure

Our files are separated into 3 main python files:

1) **project.py**

   This serves as the central file for our application, initiating all the essential procedures from the other files.

2) **interface.py**

   This file is responsible for presenting the optimal query plan, along with annotations on the plan in QEP_tab and blocks content presentations as well as visualisation in disk_tab. It also encompasses the logic for user interactions within the program, including the handling of errors originating from user inputs.

3) **explore.py**

   The main responsibilities of this file are:

   1. To establish connection with the database tables

   2. Obtain Optimal QEP

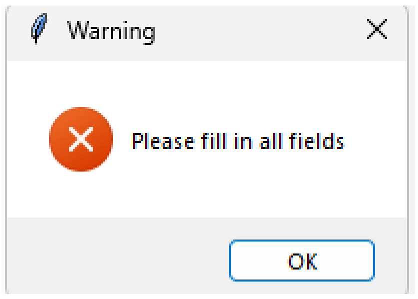
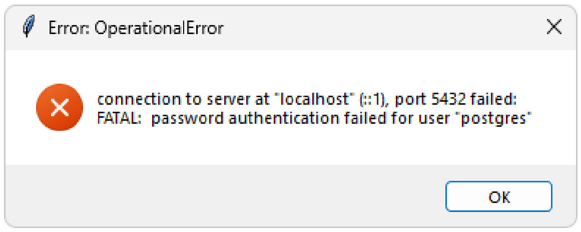
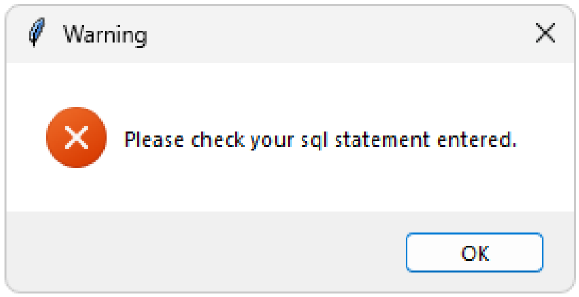
   3. Annotate the Optimal QEP

Assumptions

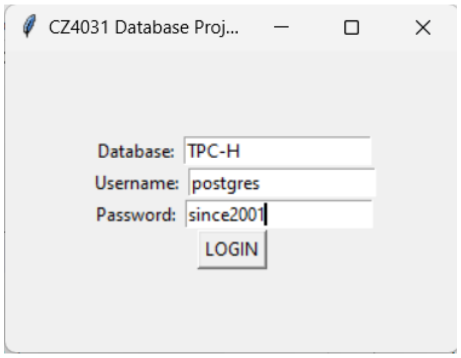
1) The data is hosted locally in the user’s PostgreSQL database
2) The program can only analyse SELECT queries, other queries that alter the database (INSERT, UPDATE, DELETE, CREATE etc.) cannot be analysed
3) Only support maximum of 1 query in the input
4) Disk blocks of tuples that remain at different intermediate nodes are ignored
5) Buffer hits (disk I/O saved by using cache) are ignored when retrieving block IDs
6) Special assumption are made for Index NL join and other joins, please refer to the Explore section for more details
7) Number of Relations involved is smaller than 10,000
8) Only Access Methods taught in the lecture such as Sequential Scan, Index Scan, Bitmap Heap Scan, Index Only Scan and Tid Scan are considered

Interface

Error Handling

	ERROR
 Warning Please fill in all fields OK	Logging in without filling required fields `(username, password, database name)`
 Error: OperationalError connection to server at "localhost" (::1), port 5432 failed: FATAL: password authentication failed for user "postgres" OK	`Connection error` while logging in to database
 Warning Please check your sql statement entered. OK	`Error` executing SQL query

Login


CZ4031 Database Proj...

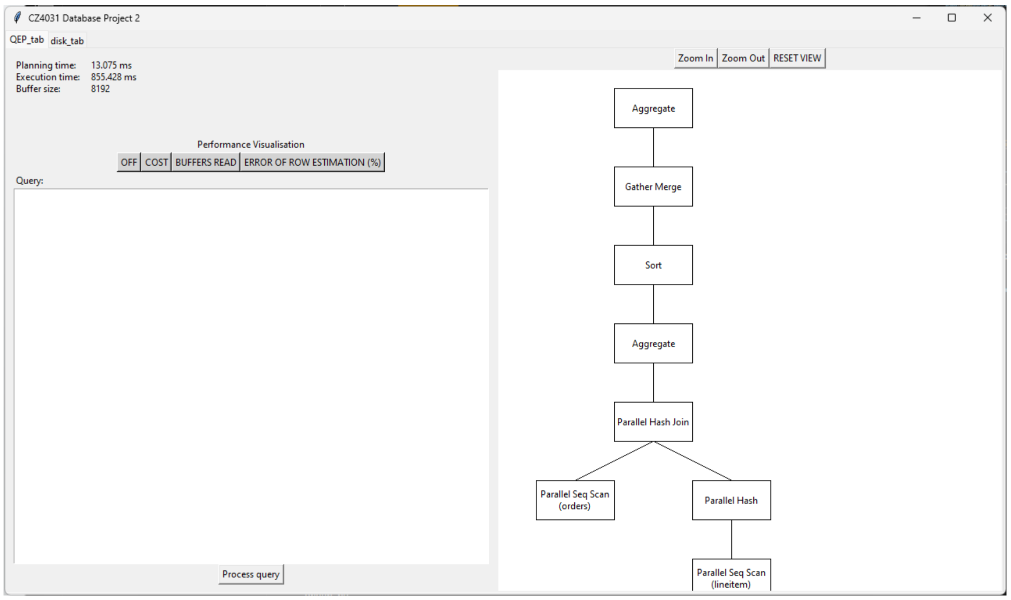
Database: TPC-H
Username: postgres
Password: since2001

Fig. 1 GUI for login

A window will popup (Fig. 1) where the user is required to enter details required to access the PostgreSQL database (`database name, username and password`). It is assumed that the PostgreSQL database is hosted locally. Once the details are entered and login button is clicked, the user is redirected to a window with multiple tabs, **QEP_tab** and **disk_tab** (Fig. 2a)

QEP_tab

To use our program, the user first enters a query at the query text box. By clicking the "Process Query" button, the Optimal Query Plan generated by the PostgreSQL will be generated in the form of a tree as shown in Fig. 2a. The **overall planning time**, **execution time** and **buffer size** is also displayed at the top left corner.

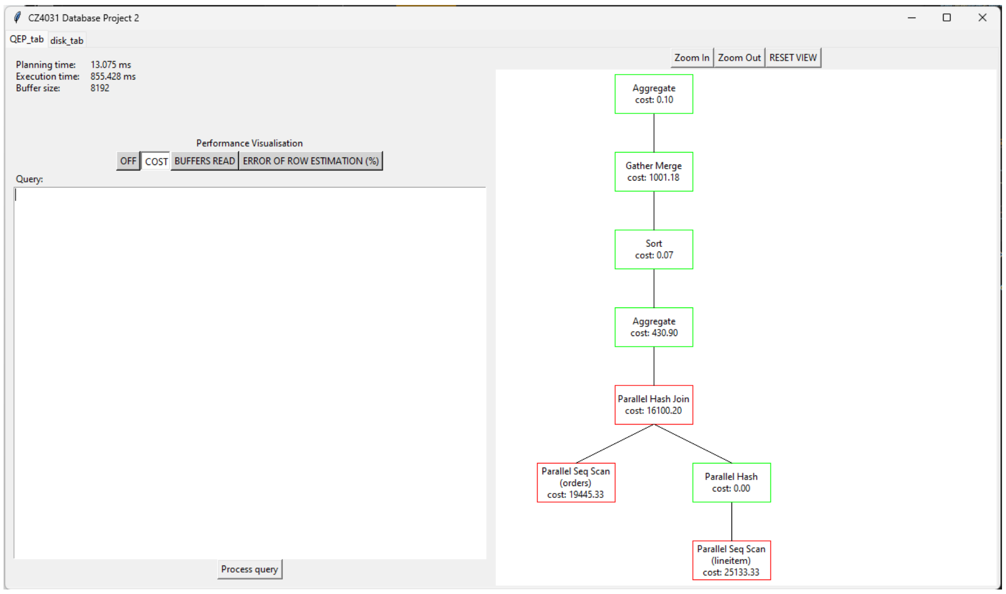

CZ4031 Database Project 2

Planning time: 13.075 ms
Execution time: 855.428 ms
Buffer size: 8192

Query:

Fig. 2a GUI of QEP_tab

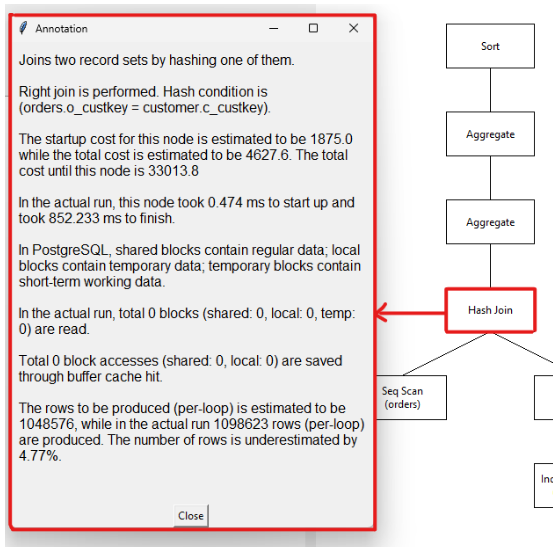
Users can toggle OFF(default), COST, BUFFER or ERROR OF ROW ESTIMATION where the respective node cost, buffer or error estimate is shown and highlighted in **RED (>60%), ORANGE (30-60%), YELLOW (10-30%), GREEN (0-10%)** according to percentage of total as shown in Fig. 2b. This helps the user better visualise each operation's cost in relation to the whole query.


CZ4031 Database Project 2
QEP_tab
disk_tab
Zoom In
Zoom Out
RESET VIEW
Performance Visualisation
OFF
COST
BUFFERS READ
ERROR OF ROW ESTIMATION (%)
Query:
Process query
Aggregate cost: 0.10
Gather Merge cost: 1001.18
Sort cost: 0.07
Aggregate cost: 430.90
Parallel Hash Join cost: 16100.20
Parallel Seq Scan (orders) cost: 19445.33
Parallel Hash cost: 0.00
Parallel Seq Scan (lineitem) cost: 25133.33

```
Planning time:    13.075 ms
Execution time:   855.428 ms
Buffer size:      8192
```

Fig. 2b COST filter

When users click on a node, they can access the relevant Query Execution Plan (QEP) aspects through annotations. Upon clicking, a pop-up window will appear, displaying the specific text corresponding to the selected tree node (as illustrated in Fig. 3).



Joins two record sets by hashing one of them.

Right join is performed. Hash condition is (orders.o_custkey = customer.c_custkey).

The startup cost for this node is estimated to be 1875.0 while the total cost is estimated to be 4627.6. The total cost until this node is 33013.8

In the actual run, this node took 0.474 ms to start up and took 852.233 ms to finish.

In PostgreSQL, shared blocks contain regular data; local blocks contain temporary data; temporary blocks contain short-term working data.

In the actual run, total 0 blocks (shared: 0, local: 0, temp: 0) are read.

Total 0 block accesses (shared: 0, local: 0) are saved through buffer cache hit.

The rows to be produced (per-loop) is estimated to be 1048576, while in the actual run 1098623 rows (per-loop) are produced. The number of rows is underestimated by 4.77%.

Fig. 3 annotation popup for operation node

disk_tab

By clicking the disk_tab above, the user can see how each relation is being accessed by the **access methods**. The user first selects the relation that he/she is interested in the Relation Options section (Fig. 4).

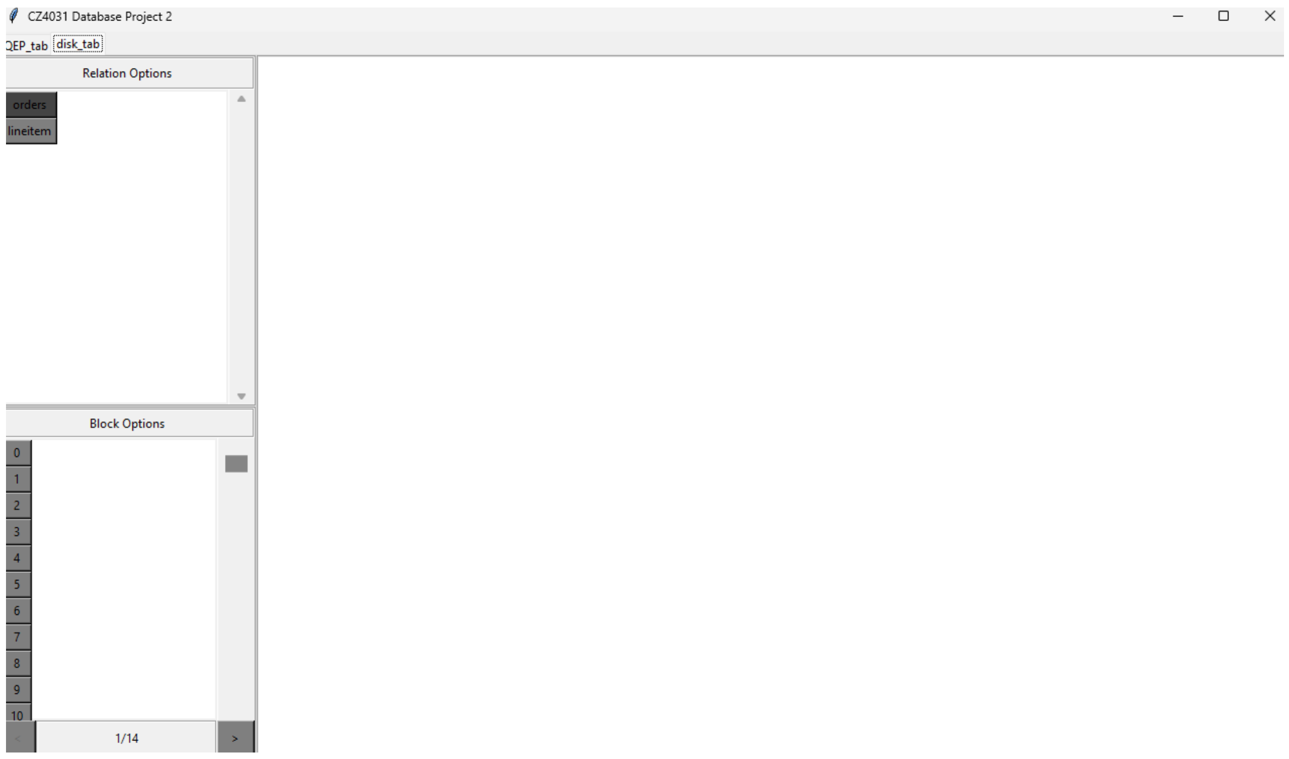


Fig. 4 GUI for disk_tab after selecting a relation

On the Block Options section displayed below, users can observe the dynamically generated block IDs in the canvas. To **enhance user-friendliness** in our application, we have chosen to group these blocks into **pages of 1000** each, instead of displaying them all at once. This approach is particularly helpful because `tkinter library` is a single-threaded application. It means users won't experience extended waiting times while the system lays out all the blocks in one go (Fig. 5).

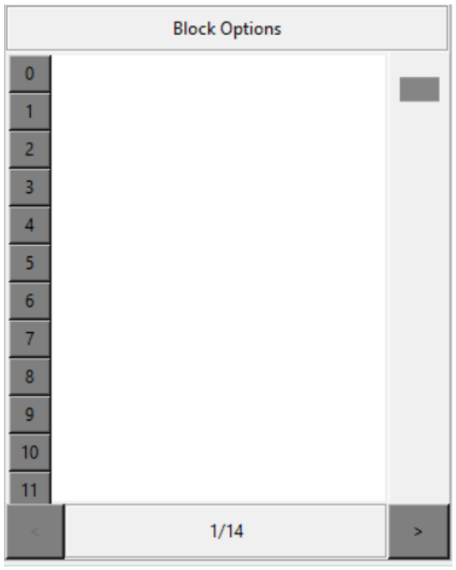


Fig. 5 Paginated Blocks

Users can easily switch between different pages by clicking the "<" symbol to go to the previous page or the ">" symbol to navigate to the next page (Fig. 6).

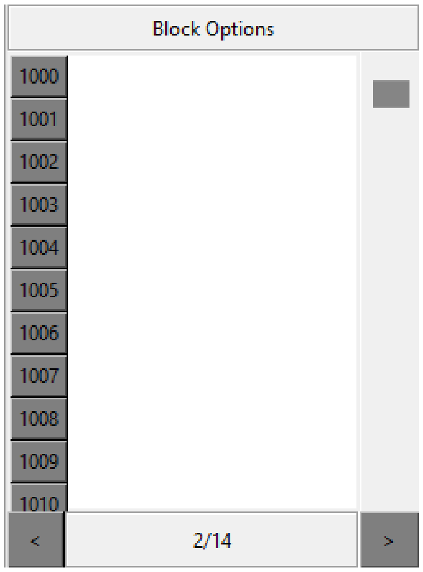


Fig. 6 Accessing different pages of blocks

To enable this functionality, our application **maintains information** about the **currently selected relation** and the corresponding index of the current sublists of block IDs being accessed. For instance, after the block IDs are divided into paginated sublists, such as [[1, 2, 3], [4, 5, 6], [7, 8, 9]], the current index being displayed (initially set at [0]) is tracked. When the user clicks the ">"

symbol, the current index is updated (current index +=1), and the new sublists are retrieved and displayed on the Block Page. “<” and “>” navigation buttons will be checked for configuration to be enabled or disabled to prevent index out of bound error.

For **block accessed visualisation**, we make the buttons corresponding to those block IDs that have not been accessed unclickable. This action allows users to clearly differentiate between assessed and non-accessed blocks, providing a visual representation (Fig. 7).

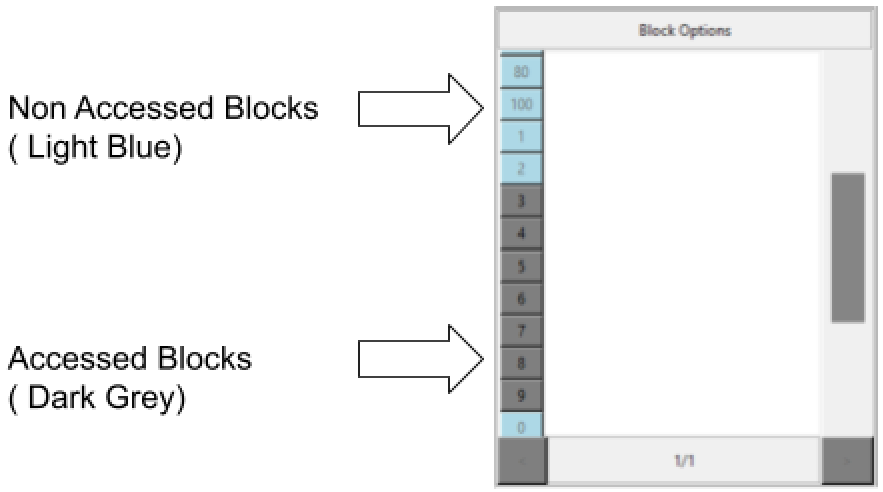
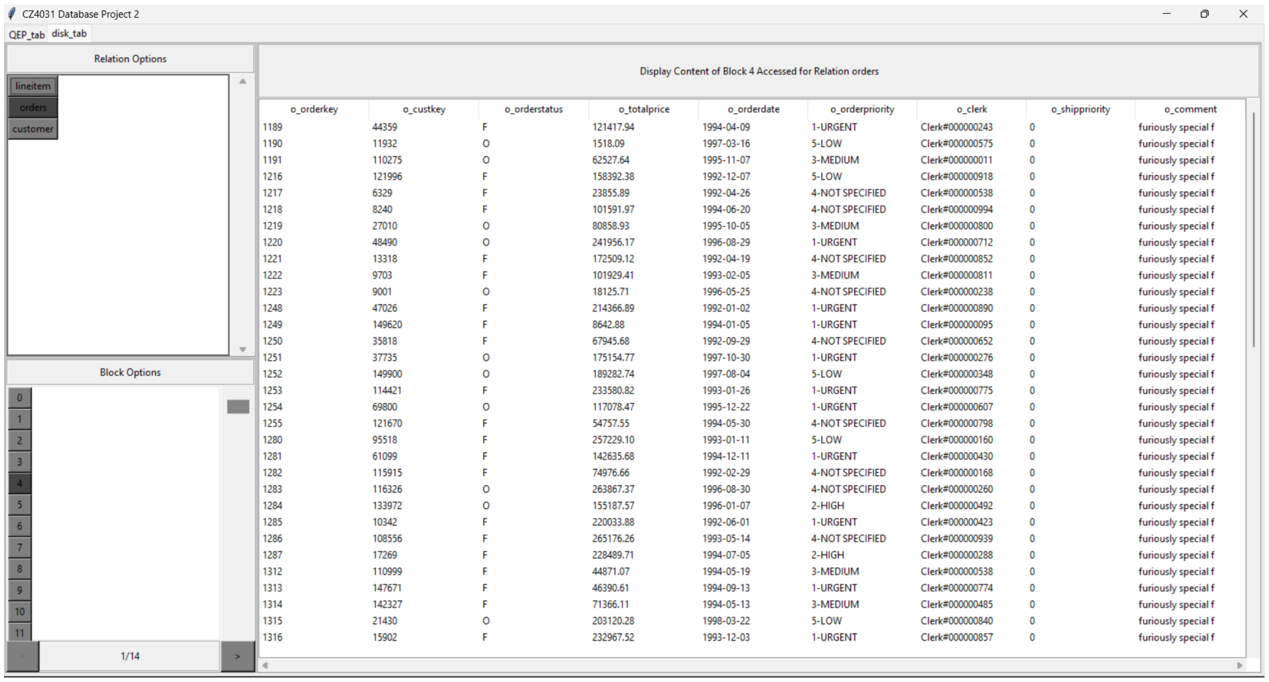


Fig. 7 Clear differentiation between assessed and non-accessed blocks

To view the complete blocks within a single page, users can scroll the **vertical scrollbar** to navigate up and down.

Click on the `BlockID` button, and the associated tuples will be fetched from execute.py and displayed in the right window in the form of TreeView (Fig. 8). Since the table's dimensions can be substantial, users can use the vertical and horizontal scrollbars to navigate and view the content of the blocks by either dragging the scrollbar or using mouse control. The scrollbars will be dynamically configured based on the table’s size.

Finally, to **enhance clarity**, each clicked button is highlighted with a dark grey colour. This highlights the current relation and block ID being accessed (Fig. 8).



Display Content of Block 4 Accessed for Relation orders

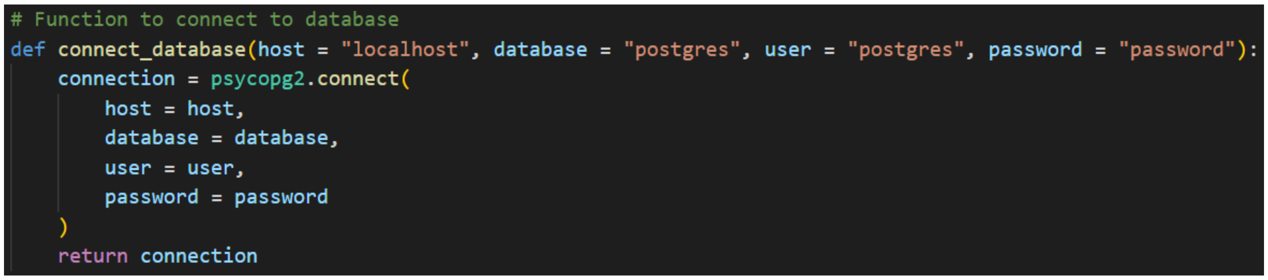
o_orderkey	o_custkey	o_orderstatus	o_totalprice	o_orderdate	o_orderpriority	o_clerk	o_shippriority	o_comment
1189	44359	F	121417.94	1994-04-09	1-URGENT	Clerk#000000243	0	furiously special f
1190	11932	O	1518.09	1997-03-16	5-LOW	Clerk#000000575	0	furiously special f
1191	110275	O	62527.64	1995-11-07	3-MEDIUM	Clerk#000000011	0	furiously special f
1216	121996	F	158392.38	1992-12-07	5-LOW	Clerk#000000918	0	furiously special f
1217	6329	F	23855.89	1992-04-26	4-NOT SPECIFIED	Clerk#000000538	0	furiously special f
1218	8240	F	101591.97	1994-06-20	4-NOT SPECIFIED	Clerk#000000994	0	furiously special f
1219	27010	O	80858.93	1995-10-05	3-MEDIUM	Clerk#000000800	0	furiously special f
1220	48490	O	241956.17	1996-08-29	1-URGENT	Clerk#000000712	0	furiously special f
1221	13318	F	172509.12	1992-04-19	4-NOT SPECIFIED	Clerk#000000852	0	furiously special f
1222	9703	F	101929.41	1993-02-05	3-MEDIUM	Clerk#000000811	0	furiously special f
1223	9001	O	18125.71	1996-05-25	4-NOT SPECIFIED	Clerk#000000238	0	furiously special f
1248	47026	F	214366.89	1992-01-02	1-URGENT	Clerk#000000890	0	furiously special f
1249	149620	F	8642.88	1994-01-05	1-URGENT	Clerk#000000095	0	furiously special f
1250	35818	F	67945.68	1992-09-29	4-NOT SPECIFIED	Clerk#000000652	0	furiously special f
1251	37735	O	175154.77	1997-10-30	1-URGENT	Clerk#000000276	0	furiously special f
1252	149900	O	189282.74	1997-08-04	5-LOW	Clerk#000000348	0	furiously special f
1253	114421	F	233580.82	1993-01-26	1-URGENT	Clerk#000000775	0	furiously special f
1254	69800	O	117078.47	1995-12-22	1-URGENT	Clerk#000000607	0	furiously special f
1255	121670	F	54757.55	1994-05-30	4-NOT SPECIFIED	Clerk#000000798	0	furiously special f
1280	95518	F	257229.10	1993-01-11	5-LOW	Clerk#000000160	0	furiously special f
1281	61099	F	142635.68	1994-12-11	1-URGENT	Clerk#000000430	0	furiously special f
1282	115915	F	74976.66	1992-02-29	4-NOT SPECIFIED	Clerk#000000168	0	furiously special f
1283	116326	O	263867.37	1996-08-30	4-NOT SPECIFIED	Clerk#000000260	0	furiously special f
1284	133972	O	155187.57	1996-01-07	2-HIGH	Clerk#000000492	0	furiously special f
1285	10342	F	220033.88	1992-06-01	1-URGENT	Clerk#000000423	0	furiously special f
1286	108556	F	265176.26	1993-05-14	4-NOT SPECIFIED	Clerk#000000939	0	furiously special f
1287	17269	F	228489.71	1994-07-05	2-HIGH	Clerk#000000288	0	furiously special f
1312	110999	F	44871.07	1994-05-19	3-MEDIUM	Clerk#000000538	0	furiously special f
1313	147671	F	46390.61	1994-09-13	1-URGENT	Clerk#000000774	0	furiously special f
1314	142327	F	71366.11	1994-05-13	3-MEDIUM	Clerk#000000485	0	furiously special f
1315	21430	O	203120.28	1998-03-22	5-LOW	Clerk#000000840	0	furiously special f
1316	15902	F	232967.52	1993-12-03	1-URGENT	Clerk#000000857	0	furiously special f

Fig. 8 Overall GUI in disk_tab

Explore

Establishing Connection

To connect to the database, the `connection()` function from `psycopg2` is used.



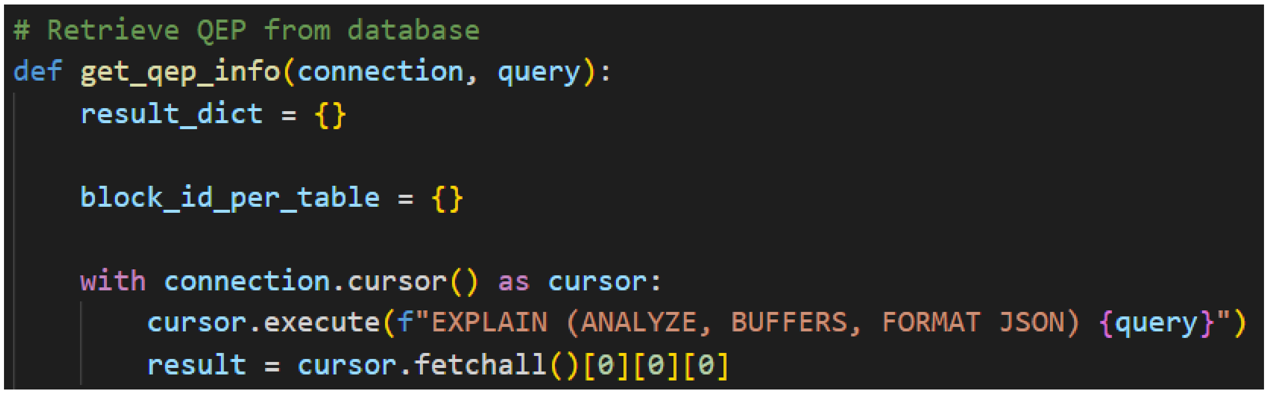
```python
# Function to connect to database
def connect_database(host = "localhost", database = "postgres", user = "postgres", password = "password"):
    connection = psycopg2.connect(
        host = host,
        database = database,
        user = user,
        password = password
    )
    return connection
```

To suit different database instances, the user can key in their respective IP Address (`host`), database name (`database`), username (`user`) and password (`password`) to connect to their own database.

The database connection will then be returned by this function, and utilised for subsequent processes.

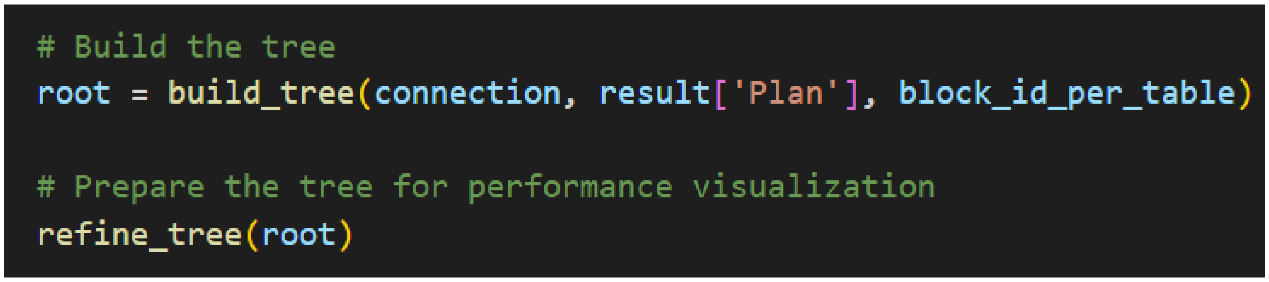
Obtain QEP

Once the query is obtained from the user input, `get_qep_info()` will be called to fetch the QEP for the corresponding query. This is made possible through the use of `EXPLAIN` statements in `PostgreSQL`. In the statement, we enabled the `ANALYZE` and `BUFFERS` options, and changed the result format to be `JSON` for easier processing. The `SUMMARY` option will be enabled together when the `ANALYZE` option is turned on. We have chosen not to enable the other options, since they are not very meaningful for the analysis of QEP.



```python
# Retrieve QEP from database
def get_qep_info(connection, query):
    result_dict = {}

    block_id_per_table = {}

    with connection.cursor() as cursor:
        cursor.execute(f"EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON) {query}")
        result = cursor.fetchall()[0][0][0]
```

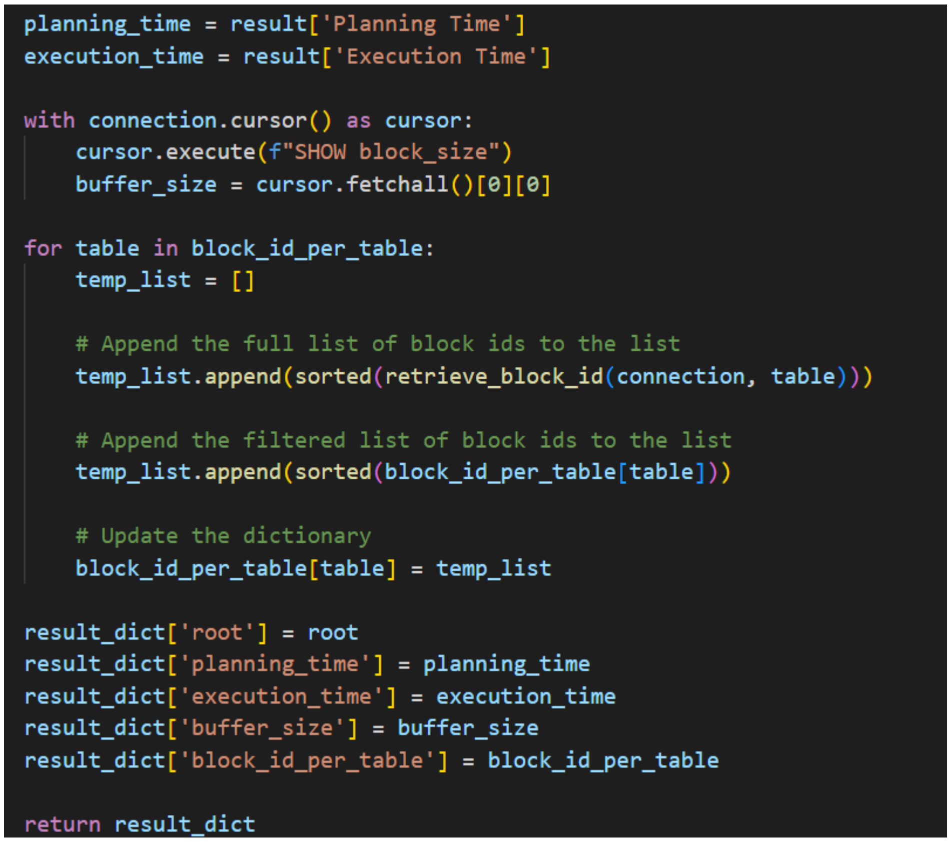
After obtaining the `JSON` format of the QEP, we will then proceed to build the QEP tree using the `build_tree()` function, which will be explained in the next section. `refine_tree()` function will be called as well to add relevant attributes to the tree nodes for performance visualisation.



```python
# Build the tree
root = build_tree(connection, result['Plan'], block_id_per_table)

# Prepare the tree for performance visualization
refine_tree(root)
```

The other aspects of QEP, including planning time, execution time, and buffer size, will also be extracted in this function. Since the `block_id_per_table` dictionary will be storing the block ID in the form of a set for each table after calling the `build_tree()` function, we will use the `sorted()` function implemented in Python, to convert them into sorted lists, making them easier to be processed by the GUI functions. We also added the list of block IDs before filtering for visualisation purposes.

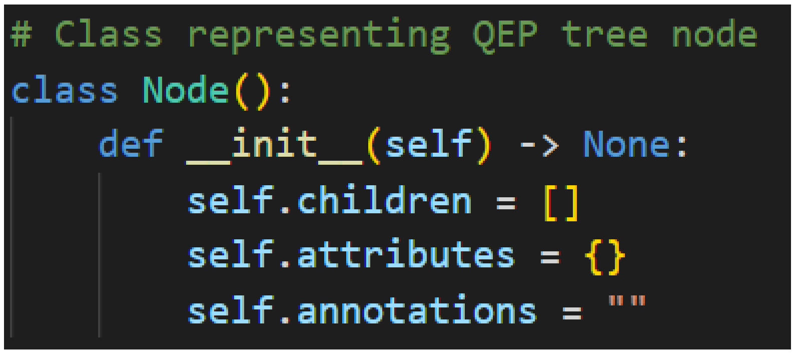
The information of the QEP will finally be packed into a dictionary, and passed to the functions in GUI for visualisation.



```python
planning_time = result['Planning Time']
execution_time = result['Execution Time']

with connection.cursor() as cursor:
    cursor.execute(f"SHOW block_size")
    buffer_size = cursor.fetchall()[0][0]

for table in block_id_per_table:
    temp_list = []

    # Append the full list of block ids to the list
    temp_list.append(sorted(retrieve_block_id(connection, table)))

    # Append the filtered list of block ids to the list
    temp_list.append(sorted(block_id_per_table[table]))

    # Update the dictionary
    block_id_per_table[table] = temp_list

result_dict['root'] = root
result_dict['planning_time'] = planning_time
result_dict['execution_time'] = execution_time
result_dict['buffer_size'] = buffer_size
result_dict['block_id_per_table'] = block_id_per_table

return result_dict
```

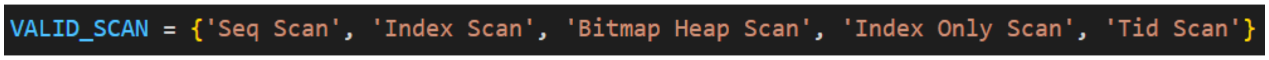
Build QEP Tree

The QEP tree is built using depth first search by calling the `build_tree()` function that we have implemented. We have used the `Node` class as shown in the figure below to represent the node in QEP Tree.



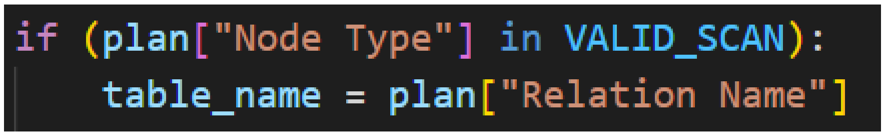
```python
# Class representing QEP tree node
class Node():
    def __init__(self) -> None:
        self.children = []
        self.attributes = {}
        self.annotations = ""
```

We first check whether the node is in charge of performing the scan operation. If so, we will then proceed to retrieve the disk blocks accessed in this node. In our algorithm, only 5 types of scan as shown in the `VALID_SCAN` set below are considered. Bitmap Index Scans are ignored as they are the children nodes of the Bitmap Heap Scan. The remaining scan types are either not executed on the actual database table or are way beyond the scope of this course.



```python
VALID_SCAN = {'Seq Scan', 'Index Scan', 'Bitmap Heap Scan', 'Index Only Scan', 'Tid Scan'}
```

We then retrieve the table name that this node is scanning, as shown in the figure below.



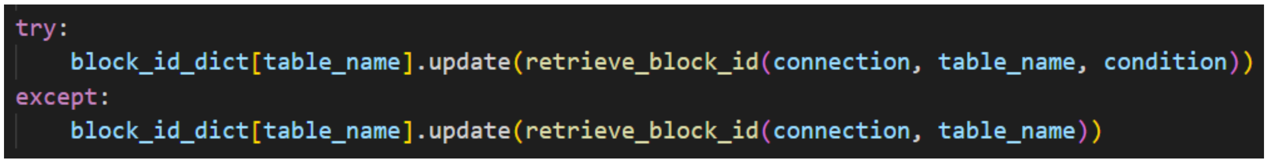
```python
if (plan["Node Type"] in VALID_SCAN):
    table_name = plan["Relation Name"]
```

As the filter operation is performed after retrieving the tuples from blocks, the `Filter` provided in the plan is not considered when determining the disk blocks accessed. Due to the length of the code, the screenshot of this part of the code is not provided, but the algorithm of determining the condition for retrieving the block IDs accessed can be explained as follows:

1. Don't retrieve block IDs for **Index Only Scan**
2. Use `Index Cond` provided in the plan for **Index Scan**
3. Use `Recheck Cond` provided in the plan for **Bitmap Heap Scan**
4. Use `TID Cond` provided in the plan for **Tid Scan**
5. No condition checking for **Seq Scan**

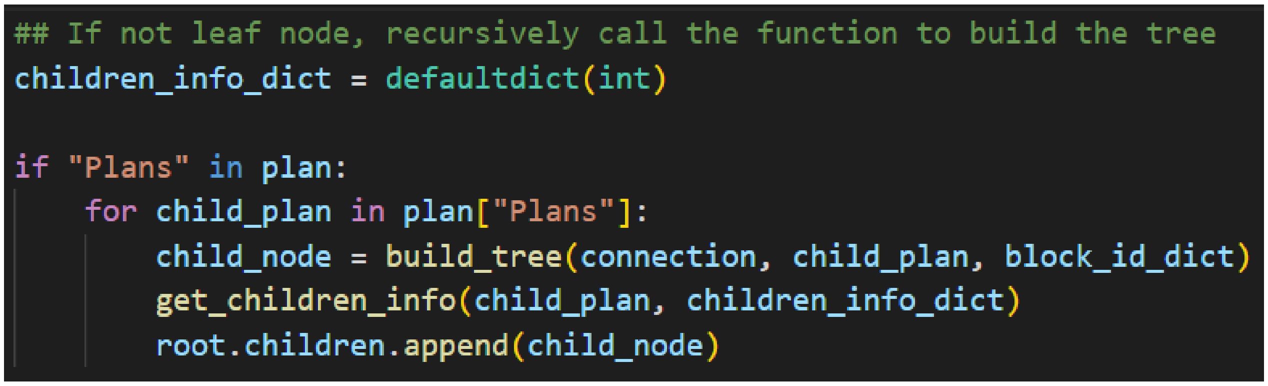
The condition given in the QEP might contain join conditions (such as Index Scan when performing index nested loop join), which hold references to other tables. This causes some issues because we don't have information about the other tables in a particular QEP node. The details of how we handle this issue and the reasons why we have chosen that approach are explained in the "Heuristics for Join Conditions" section.

Note that in the code, we use the built-in `update()` function to update the set. This is done to avoid overriding the previously retrieved block IDs in case any tables are scanned twice. If the heuristics fail due to unforeseen circumstances, the block IDs will still be fetched without regard to the conditions.



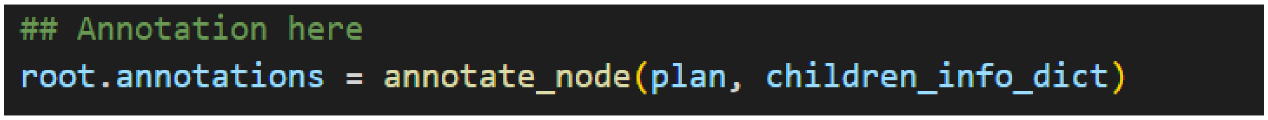
```
try:
    block_id_dict[table_name].update(retrieve_block_id(connection, table_name, condition))
except:
    block_id_dict[table_name].update(retrieve_block_id(connection, table_name))
```

Representing the children nodes, the algorithm will **iterate** through those nested plans to build the tree for them and **add** these built trees to the current node as the child nodes. The information of the children nodes will be **aggregated** as well for annotation purposes.



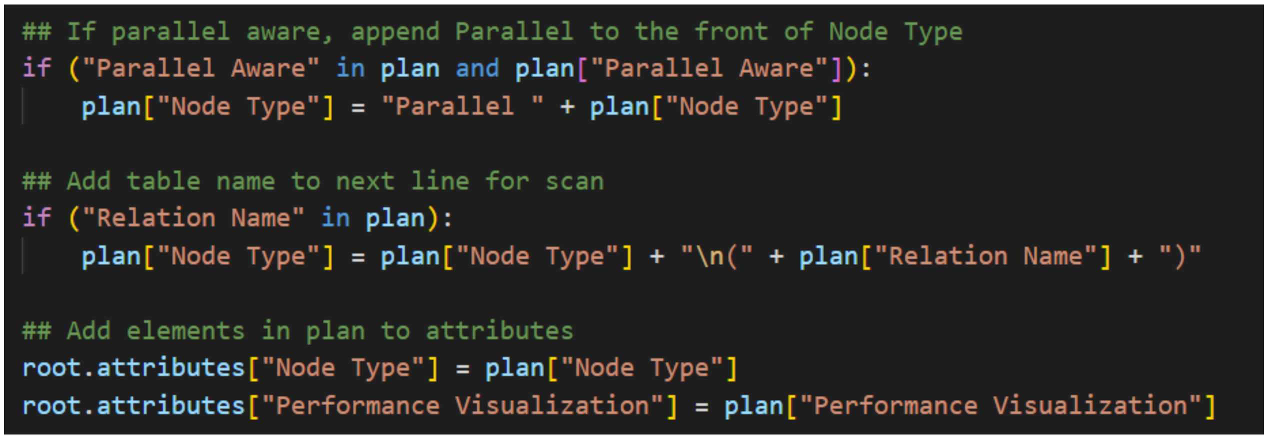
```
## If not leaf node, recursively call the function to build the tree
children_info_dict = defaultdict(int)

if "Plans" in plan:
    for child_plan in plan["Plans"]:
        child_node = build_tree(connection, child_plan, block_id_dict)
        get_children_info(child_plan, children_info_dict)
        root.children.append(child_node)
```

Next, the `annotate_node()` function will be called to provide natural language explanations for the node. The actual algorithm of the `annotate_node()` function is explained in detail in the later part of this report.



```
## Annotation here
root.annotations = annotate_node(plan, children_info_dict)
```

We further add more explanations for a node, including whether it is parallel-aware or the table name it is scanning for nodes that perform a scan operation, as shown in the figure below. `Node Type` and `Performance Visualization` in the plan will be added to the `attributes` of the node for potential utilisation by the functions in the GUI. Finally, this function returns the root node of the QEP tree.



```
## If parallel aware, append Parallel to the front of Node Type
if ("Parallel Aware" in plan and plan["Parallel Aware"]):
    plan["Node Type"] = "Parallel " + plan["Node Type"]

## Add table name to next line for scan
if ("Relation Name" in plan):
    plan["Node Type"] = plan["Node Type"] + "\n(" + plan["Relation Name"] + ")"

## Add elements in plan to attributes
root.attributes["Node Type"] = plan["Node Type"]
root.attributes["Performance Visualization"] = plan["Performance Visualization"]
```

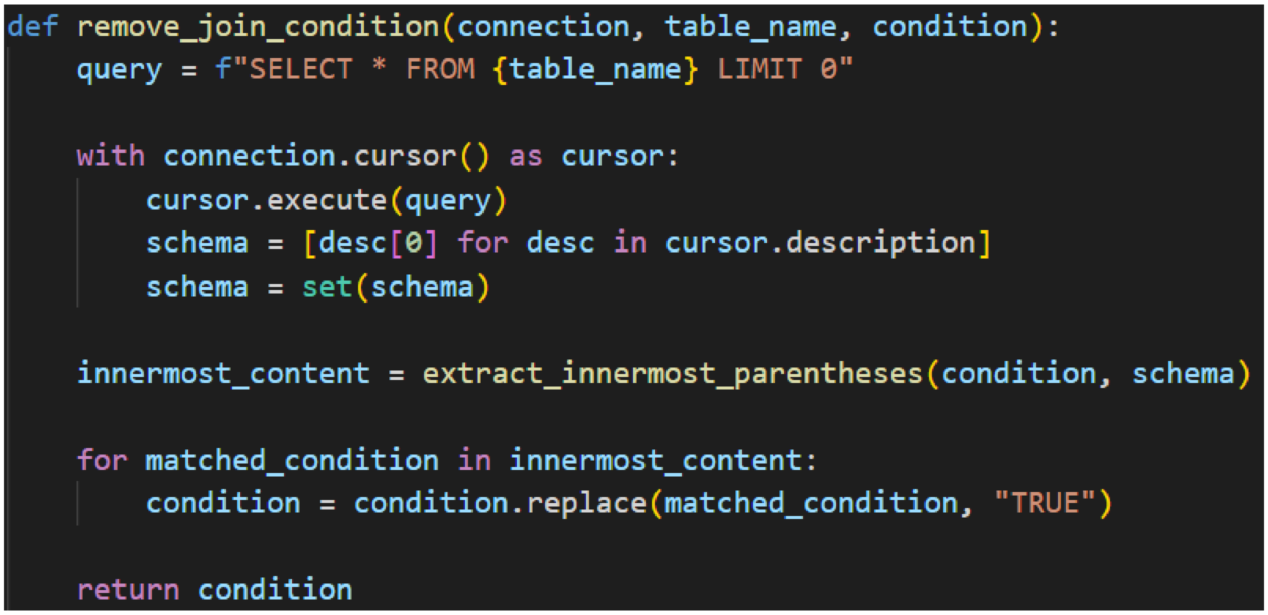
Heuristics for Join Condition

As mentioned in the previous section, the join condition provided in the QEP node has caused some issues when we try to retrieve the block IDs. To simulate the actual case, at each children node, we could perform the necessary actions in that particular node and pass the processed data to the respective parent nodes in order to retrieve the candidate tuples for joining. However, we have decided not to do so for the following reasons:

1. Passing such a **large volume of data** while traversing the QEP tree is not realistic and will cause a **huge burden** to the system
2. Achieving this goal involves considering **too many QEP nodes**, and some of the operations are even hidden from users
3. There is **no readily available off-the-shelf software** to serve this purpose. To achieve the objectives mentioned above, a large codebase is required, which will demand a significant amount of time for development and may result in a program slowdown

We've tried to explore some alternative solutions, explore alternative solutions, such as parsing QEP nodes into SQL statements to solve this issue. However, these solutions are still too complex. Thus, we have decided to implement our own ***heuristic functions*** to filter out the join conditions from the condition section in the QEP plan that we use for retrieving block IDs, aiming to maintain relatively high accuracy without causing the entire algorithm to become overly complex.

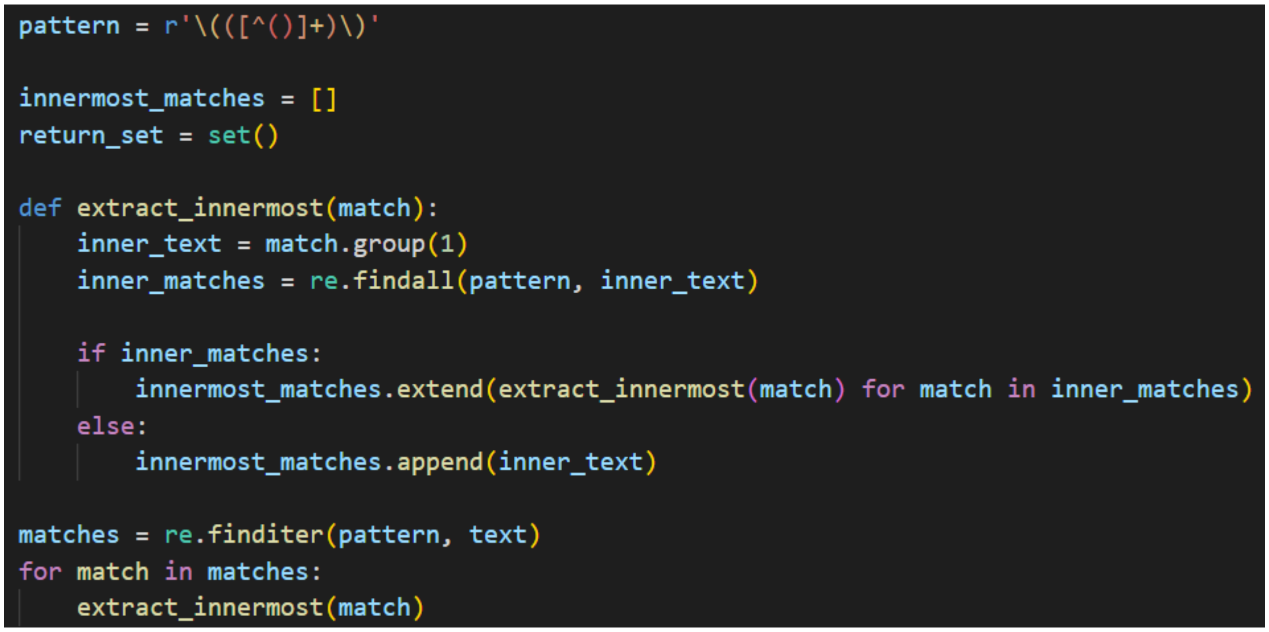
As shown in the following figure, we will begin by extracting the list of attributes for the specific table. Afterward, the heuristic functions for identifying the join conditions will be invoked. Those join conditions will then be replaced by `TRUE`. By doing so, we effectively ignore the join conditions while preserving the other conditions.



```python
def remove_join_condition(connection, table_name, condition):
    query = f"SELECT * FROM {table_name} LIMIT 0"

    with connection.cursor() as cursor:
        cursor.execute(query)
        schema = [desc[0] for desc in cursor.description]
        schema = set(schema)

    innermost_content = extract_innermost_parentheses(condition, schema)

    for matched_condition in innermost_content:
        condition = condition.replace(matched_condition, "TRUE")

    return condition
```

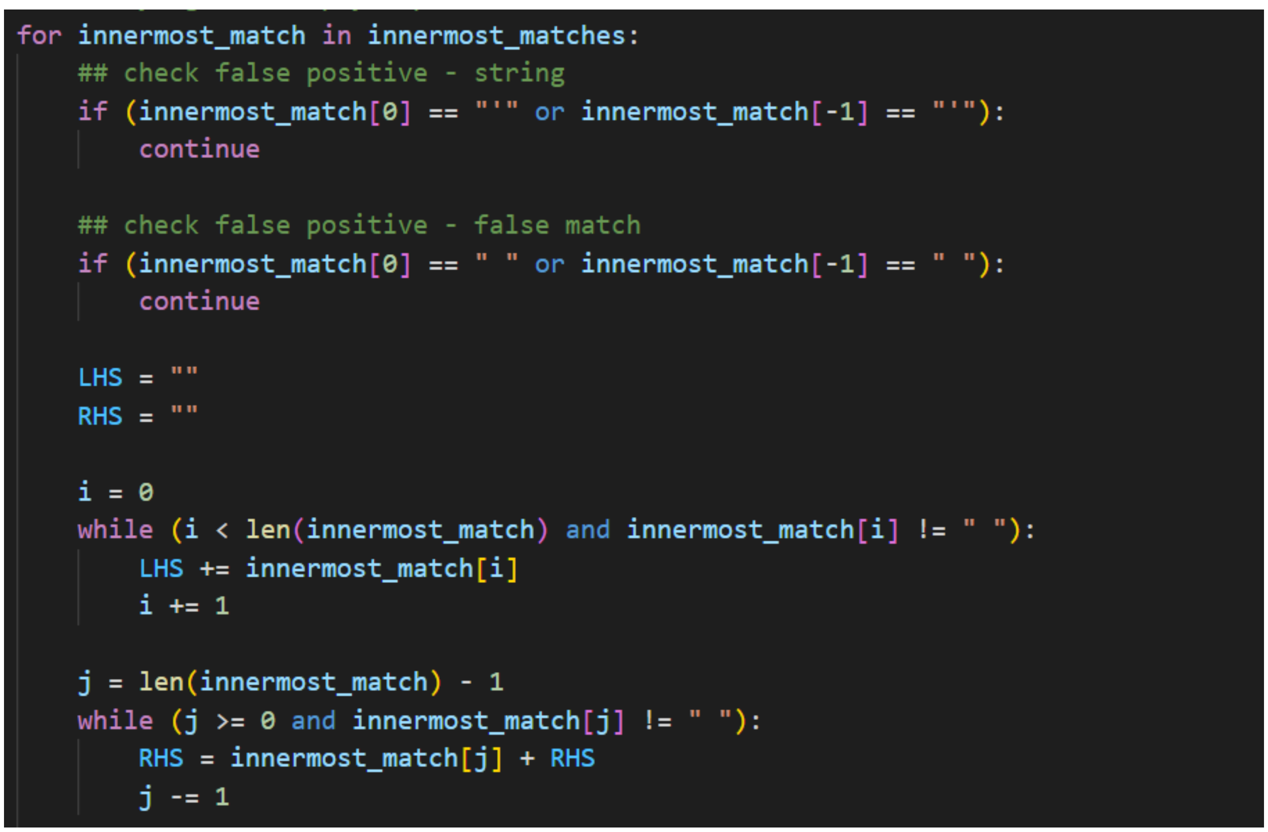
The detail of the heuristic can be explained as follow,

By retrieving numerous QEPs, we observe that the join conditions are always in the format of "`(xxx <operator> yyy)`", even when the conditions are nested. Thus, we first apply regular expressions to extract those contents within the innermost parentheses.



```python
pattern = r'\(([^()]+)\)'

innermost_matches = []
return_set = set()

def extract_innermost(match):
    inner_text = match.group(1)
    inner_matches = re.findall(pattern, inner_text)

    if inner_matches:
        innermost_matches.extend(extract_innermost(match) for match in inner_matches)
    else:
        innermost_matches.append(inner_text)

matches = re.finditer(pattern, text)
for match in matches:
    extract_innermost(match)
```

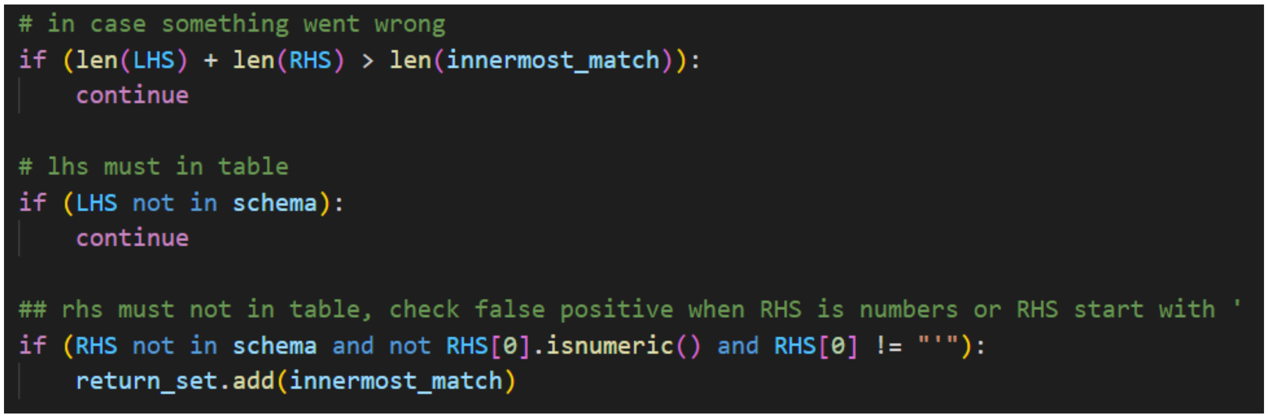
After that, we perform extra checking as shown in the following figure to avoid false positives. We then proceed to extract both the left hand side (`LHS`) and the right hand side (`RHS`) of that particular condition. We use the whitespace as the terminator, as this is the fixed format used by PostgreSQL in QEP.



```python
for innermost_match in innermost_matches:
    ## check false positive - string
    if (innermost_match[0] == "'" or innermost_match[-1] == "'"):
        continue

    ## check false positive - false match
    if (innermost_match[0] == " " or innermost_match[-1] == " "):
        continue

    LHS = ""
    RHS = ""

    i = 0
    while (i < len(innermost_match) and innermost_match[i] != " "):
        LHS += innermost_match[i]
        i += 1

    j = len(innermost_match) - 1
    while (j >= 0 and innermost_match[j] != " "):
        RHS = innermost_match[j] + RHS
        j -= 1
```

Extra checking is then again performed to avoid false positives. Based on our observation, in those scan nodes, the join conditions are always displayed in the format of "(`attribute_name <operator> other_table_name.attribute_name`)". We've utilised this observation in our heuristics.

To be considered as a join condition, the LHS must be included in the list of attributes. Other than not being in the list of attributes, RHS must also not be a number nor a string. If all conditions are satisfied, the matched condition will be added to the `return_set`, which will be eventually returned by this function.



```python
# in case something went wrong
if (len(LHS) + len(RHS) > len(innermost_match)):
    continue

# lhs must in table
if (LHS not in schema):
    continue

## rhs must not in table, check false positive when RHS is numbers or RHS start with '
if (RHS not in schema and not RHS[0].isnumeric() and RHS[0] != "'"):
    return_set.add(innermost_match)
```

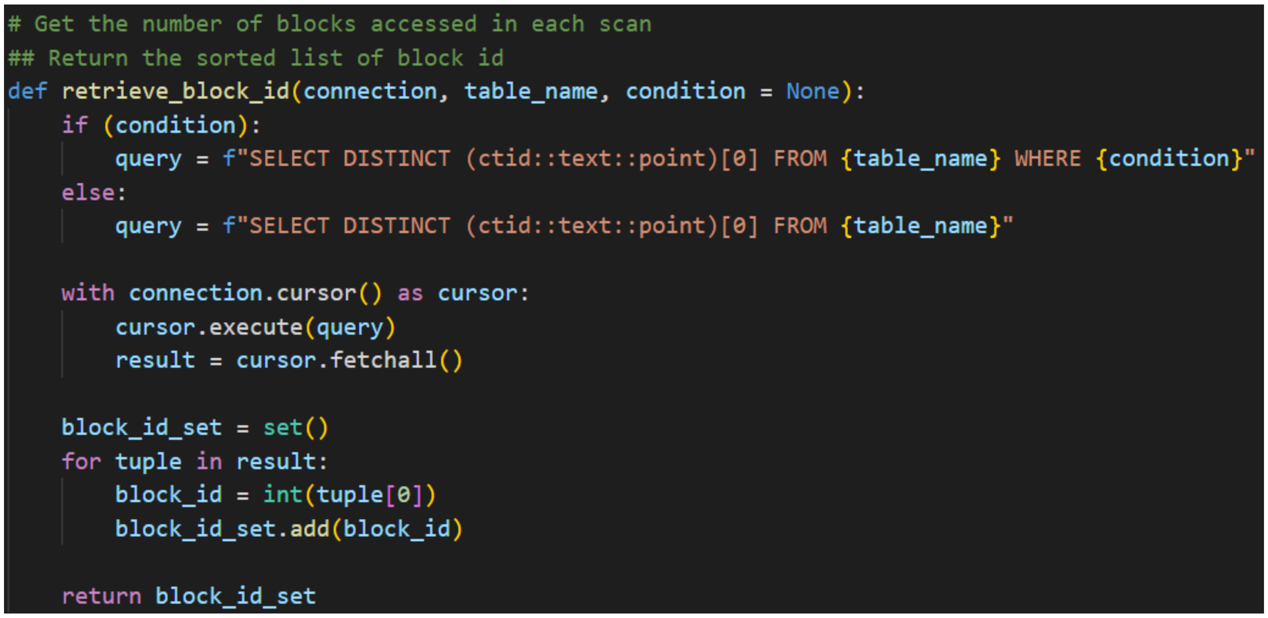
Do note that this heuristic is **not 100% perfect** and might still have some false negatives. However, due to the lack of available off-the-shelf software, the heuristic approach is adopted as a compromise.

Obtain Block ID

Once the `table_name` and `condition` have been properly processed and cleaned in the calling function when building the QEP tree, we will then proceed to execute the respective SQL statement to obtain the relevant ids of those blocks. We utilised the `cursor` provided in the `connection` class of `psycopg2` to execute the query.

When executing the query, we use `(ctid::text::point)[0]` for projection since we are only interested in the block ID, not the row ID. Besides, we decided to utilise the `DISTINCT` keyword to reduce the number of tuples that will be returned in the process. Through this method, we have **significantly reduced the latency** incurred when executing the query from **minutes to seconds**.

After executing the statement, we will iterate through the list of tuples, adding those block IDs to a set. The set containing all these block IDs will be returned by this function for subsequent processing.

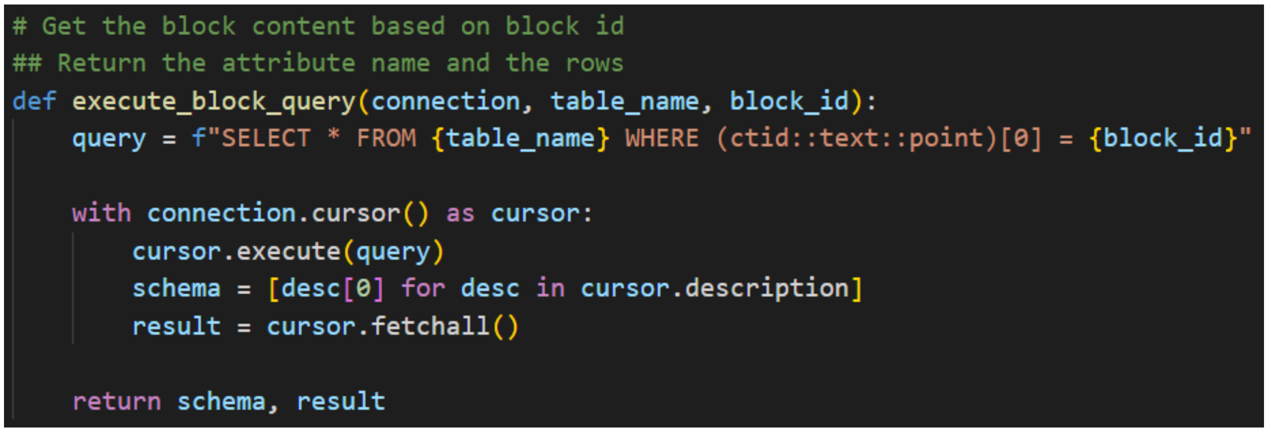


```python
# Get the number of blocks accessed in each scan
## Return the sorted list of block id
def retrieve_block_id(connection, table_name, condition = None):
    if (condition):
        query = f"SELECT DISTINCT (ctid::text::point)[0] FROM {table_name} WHERE {condition}"
    else:
        query = f"SELECT DISTINCT (ctid::text::point)[0] FROM {table_name}"

    with connection.cursor() as cursor:
        cursor.execute(query)
        result = cursor.fetchall()

    block_id_set = set()
    for tuple in result:
        block_id = int(tuple[0])
        block_id_set.add(block_id)

    return block_id_set
```

Retrieve Disk Block Content

Once the `table_name` and the `block_id` is provided, the process of retrieving content from the respective block is trivial, as shown in the following code. We utilised the `cursor` provided in the `connection` class of `psycopg2` to execute the query.

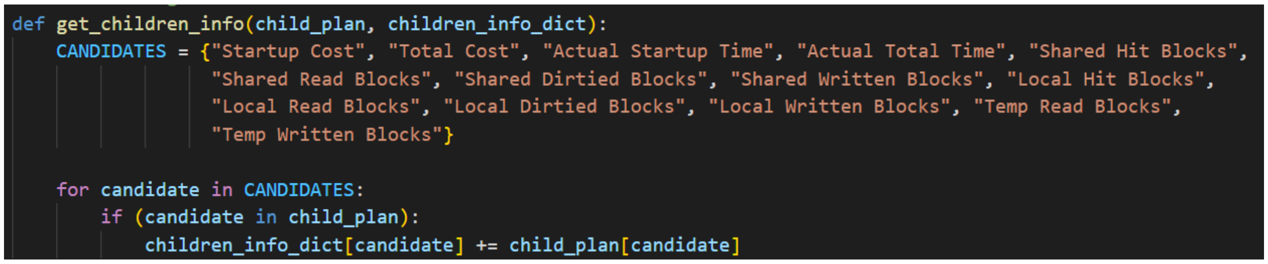
In this function, two parameters will be returned (in the form of tuple), namely `schema`, which contains the list of attribute names to be displayed in the GUI and let the users better understand the results displayed, and `result`, which contains a list of tuples stored the respective `block_id`.



```python
# Get the block content based on block id
## Return the attribute name and the rows
def execute_block_query(connection, table_name, block_id):
    query = f"SELECT * FROM {table_name} WHERE (ctid::text::point)[0] = {block_id}"

    with connection.cursor() as cursor:
        cursor.execute(query)
        schema = [desc[0] for desc in cursor.description]
        result = cursor.fetchall()

    return schema, result
```

Get Information of Child Nodes

As some of the information provided in the PostgreSQL QEP includes data collected from child nodes. To facilitate the understanding of QEP by beginners, we've decided to aggregate those values and subtract them when performing annotations. The information that will be collected is specified in the `CANDIDATES` set.

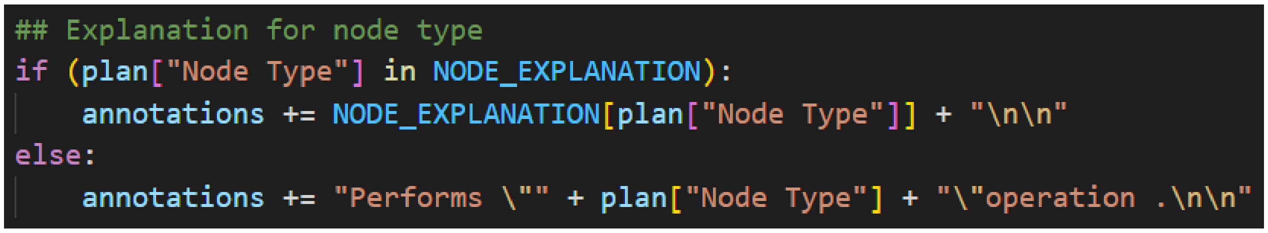


```python
def get_children_info(child_plan, children_info_dict):
    CANDIDATES = {"Startup Cost", "Total Cost", "Actual Startup Time", "Actual Total Time", "Shared Hit Blocks",
                  "Shared Read Blocks", "Shared Dirtied Blocks", "Shared Written Blocks", "Local Hit Blocks",
                  "Local Read Blocks", "Local Dirtied Blocks", "Local Written Blocks", "Temp Read Blocks",
                  "Temp Written Blocks"}

    for candidate in CANDIDATES:
        if (candidate in child_plan):
            children_info_dict[candidate] += child_plan[candidate]
```

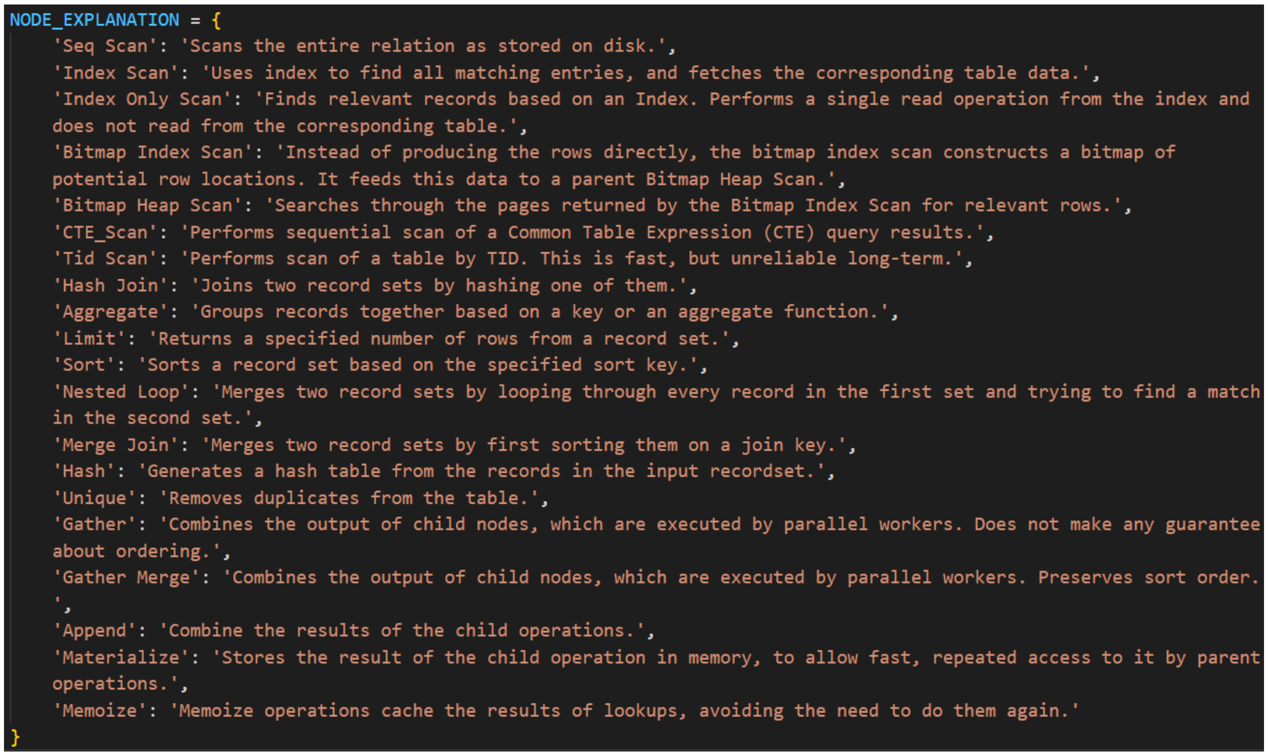
Annotation

To help users better understand what is actually happening in each of the QEP nodes, we implemented the `annotate_node()` function. This function explains different aspects of the QEP in simple, natural language that is easy to understand.

First, we explain what the QEP nodes do based on their node type. Since there are so many possible node types in `PostgreSQL`'s QEP, and some of them are rarely seen, we have included only a portion of them inside the `NODE_EXPLANATION` dictionary. The remaining node types will be appended to the `annotations` without further processing.



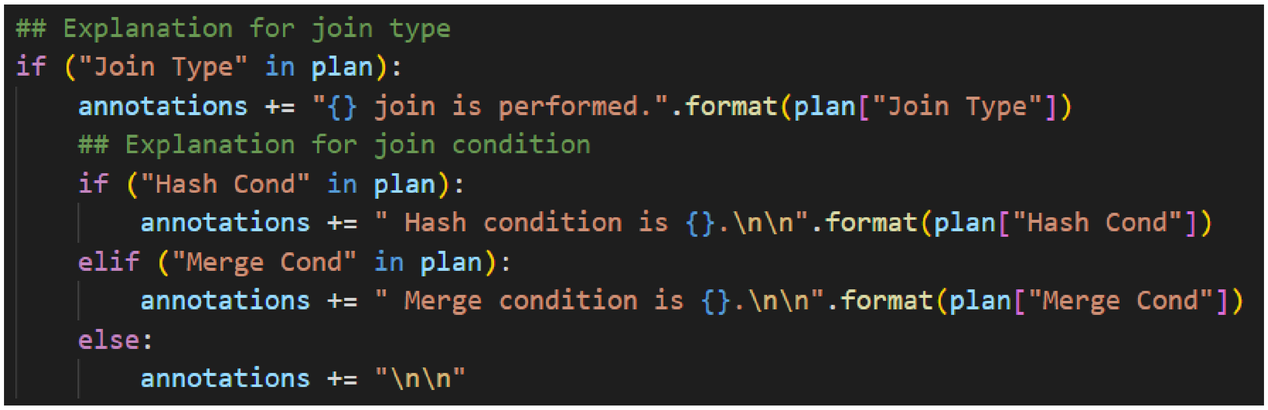
```
## Explanation for node type
if (plan["Node Type"] in NODE_EXPLANATION):
    annotations += NODE_EXPLANATION[plan["Node Type"]] + "\n\n"
else:
    annotations += "Performs \"" + plan["Node Type"] + "\"operation .\n\n"
```



```
NODE_EXPLANATION = {
    'Seq Scan': 'Scans the entire relation as stored on disk.',
    'Index Scan': 'Uses index to find all matching entries, and fetches the corresponding table data.',
    'Index Only Scan': 'Finds relevant records based on an Index. Performs a single read operation from the index and
    does not read from the corresponding table.',
    'Bitmap Index Scan': 'Instead of producing the rows directly, the bitmap index scan constructs a bitmap of
    potential row locations. It feeds this data to a parent Bitmap Heap Scan.',
    'Bitmap Heap Scan': 'Searches through the pages returned by the Bitmap Index Scan for relevant rows.',
    'CTE_Scan': 'Performs sequential scan of a Common Table Expression (CTE) query results.',
    'Tid Scan': 'Performs scan of a table by TID. This is fast, but unreliable long-term.',
    'Hash Join': 'Joins two record sets by hashing one of them.',
    'Aggregate': 'Groups records together based on a key or an aggregate function.',
    'Limit': 'Returns a specified number of rows from a record set.',
    'Sort': 'Sorts a record set based on the specified sort key.',
    'Nested Loop': 'Merges two record sets by looping through every record in the first set and trying to find a match
    in the second set.',
    'Merge Join': 'Merges two record sets by first sorting them on a join key.',
    'Hash': 'Generates a hash table from the records in the input recordset.',
    'Unique': 'Removes duplicates from the table.',
    'Gather': 'Combines the output of child nodes, which are executed by parallel workers. Does not make any guarantee
    about ordering.',
    'Gather Merge': 'Combines the output of child nodes, which are executed by parallel workers. Preserves sort order.
    ',
    'Append': 'Combine the results of the child operations.',
    'Materialize': 'Stores the result of the child operation in memory, to allow fast, repeated access to it by parent
    operations.',
    'Memoize': 'Memoize operations cache the results of lookups, avoiding the need to do them again.'
}
```

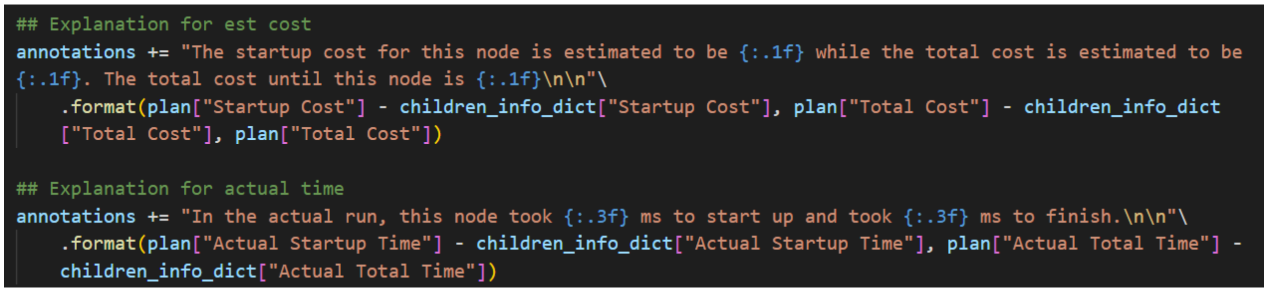
Next, if the node is responsible for the joining operation, we will add the join type to the `annotations`. Additionally, the join condition will be extracted from the attributes `Hash Cond` or `Merge Cond` and added to the `annotations`. Please note that the nested loop join is not considered in this algorithm. The reason why we do this is because, if an index nested loop join is performed, the join condition will be pushed to the child nodes, which may result in less accurate information.

We have chosen not to traverse down to the index scan node to extract relevant join conditions because it is very complex, and there is no off-the-shelf software available that can guarantee the accuracy of this operation (other index conditions might have been included during the process).



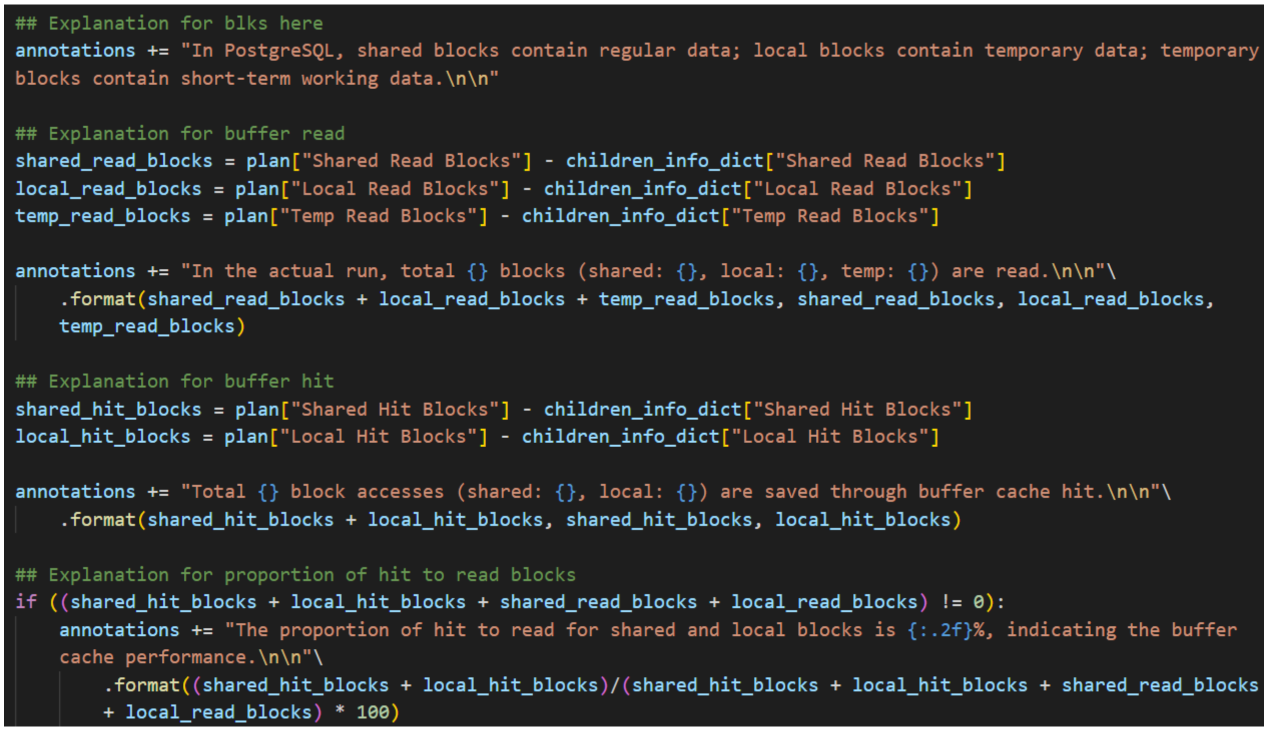
```python
## Explanation for join type
if ("Join Type" in plan):
    annotations += "{} join is performed.".format(plan["Join Type"])
    ## Explanation for join condition
    if ("Hash Cond" in plan):
        annotations += " Hash condition is {}.\n\n".format(plan["Hash Cond"])
    elif ("Merge Cond" in plan):
        annotations += " Merge condition is {}.\n\n".format(plan["Merge Cond"])
    else:
        annotations += "\n\n"
```

After that, the explanation for the estimated cost, as well as the actual time taken, is added to the `annotations` for each node. Since the information provided in the QEP is aggregated from the information in the child nodes, we perform further processing (***subtraction***) to gain a better understanding of the cost and time for that particular node, as shown in the following code.



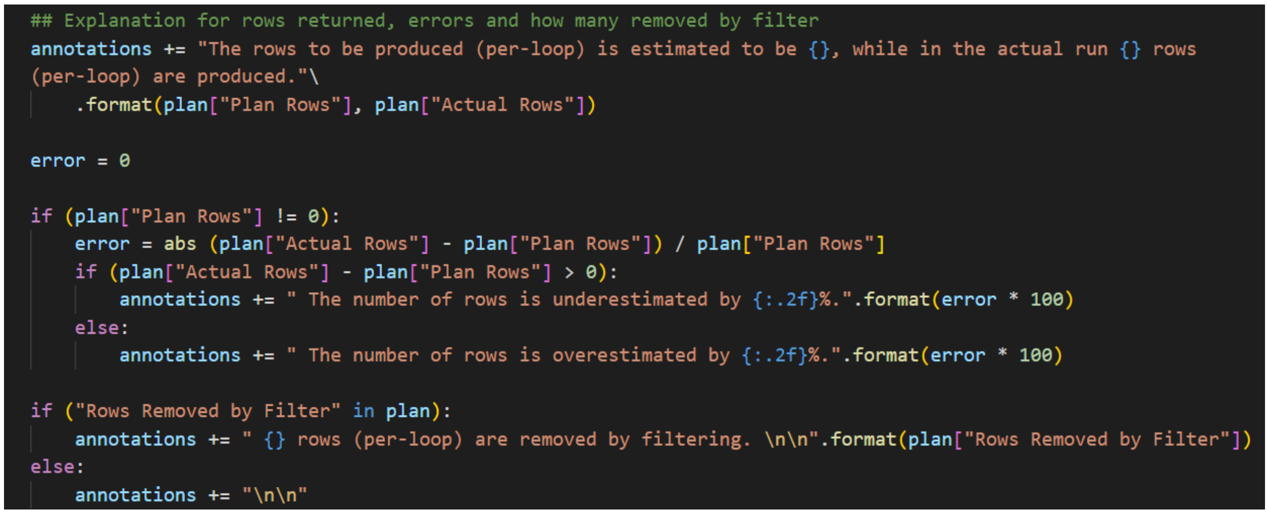
```python
## Explanation for est cost
annotations += "The startup cost for this node is estimated to be {:.1f} while the total cost is estimated to be
{:.1f}. The total cost until this node is {:.1f}\n\n"\
    .format(plan["Startup Cost"] - children_info_dict["Startup Cost"], plan["Total Cost"] - children_info_dict
    ["Total Cost"], plan["Total Cost"])

## Explanation for actual time
annotations += "In the actual run, this node took {:.3f} ms to start up and took {:.3f} ms to finish.\n\n"\
    .format(plan["Actual Startup Time"] - children_info_dict["Actual Startup Time"], plan["Actual Total Time"] -
    children_info_dict["Actual Total Time"])
```

We have also included detailed explanations about the buffer information for each node in the `annotations`. As these attributes are also aggregated from the information in the child nodes, similar to the previous code, subtraction is performed to gain a better understanding of those explanations. We have also added the proportion of hits to reads for shared and local blocks to help users gain a better understanding of the buffer cache performance.



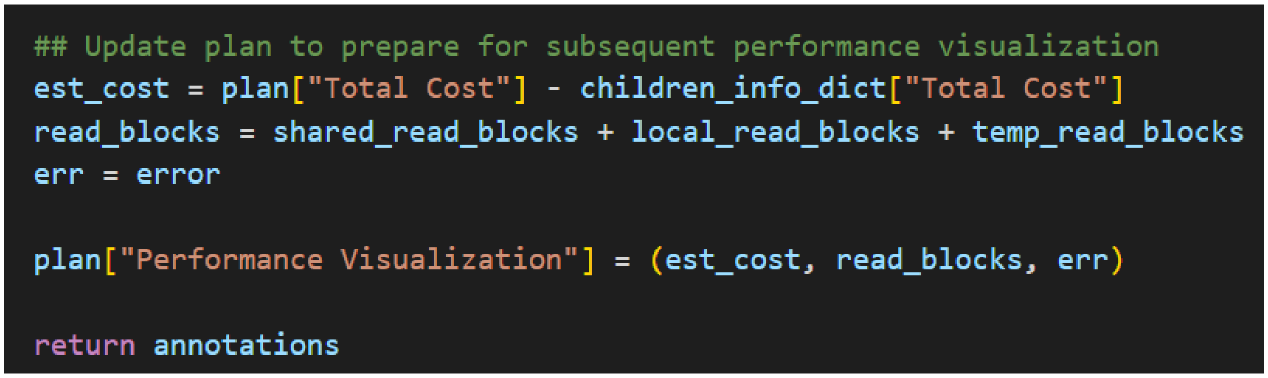
```python
## Explanation for blks here
annotations += "In PostgreSQL, shared blocks contain regular data; local blocks contain temporary data; temporary
blocks contain short-term working data.\n\n"

## Explanation for buffer read
shared_read_blocks = plan["Shared Read Blocks"] - children_info_dict["Shared Read Blocks"]
local_read_blocks = plan["Local Read Blocks"] - children_info_dict["Local Read Blocks"]
temp_read_blocks = plan["Temp Read Blocks"] - children_info_dict["Temp Read Blocks"]

annotations += "In the actual run, total {} blocks (shared: {}, local: {}, temp: {}) are read.\n\n"\
    .format(shared_read_blocks + local_read_blocks + temp_read_blocks, shared_read_blocks, local_read_blocks,
    temp_read_blocks)

## Explanation for buffer hit
shared_hit_blocks = plan["Shared Hit Blocks"] - children_info_dict["Shared Hit Blocks"]
local_hit_blocks = plan["Local Hit Blocks"] - children_info_dict["Local Hit Blocks"]

annotations += "Total {} block accesses (shared: {}, local: {}) are saved through buffer cache hit.\n\n"\
    .format(shared_hit_blocks + local_hit_blocks, shared_hit_blocks, local_hit_blocks)

## Explanation for proportion of hit to read blocks
if ((shared_hit_blocks + local_hit_blocks + shared_read_blocks + local_read_blocks) != 0):
    annotations += "The proportion of hit to read for shared and local blocks is {:.2f}%, indicating the buffer
    cache performance.\n\n"\
        .format((shared_hit_blocks + local_hit_blocks)/(shared_hit_blocks + local_hit_blocks + shared_read_blocks
        + local_read_blocks) * 100)
```

Moreover, we include the estimation of rows to be produced and the actual rows produced for each node in the `annotations`. The estimation error is calculated and also added to the `annotations` to help the user gain a better understanding of the usefulness of this QEP, as the estimation of the number of rows produced at each node significantly contributes to the decision of selecting the optimal QEP.



```python
## Explanation for rows returned, errors and how many removed by filter
annotations += "The rows to be produced (per-loop) is estimated to be {}, while in the actual run {} rows
(per-loop) are produced."\
    .format(plan["Plan Rows"], plan["Actual Rows"])

error = 0

if (plan["Plan Rows"] != 0):
    error = abs (plan["Actual Rows"] - plan["Plan Rows"]) / plan["Plan Rows"]
    if (plan["Actual Rows"] - plan["Plan Rows"] > 0):
        annotations += " The number of rows is underestimated by {:.2f}%.".format(error * 100)
    else:
        annotations += " The number of rows is overestimated by {:.2f}%.".format(error * 100)

if ("Rows Removed by Filter" in plan):
    annotations += " {} rows (per-loop) are removed by filtering. \n\n".format(plan["Rows Removed by Filter"])
else:
    annotations += "\n\n"
```

Finally, before returning the function, we will add a tuple to the plan containing the **estimated cost**, the number of **blocks read**, and the **error of row estimation**. This tuple will facilitate the subsequent visualisation of the performance of each node. The number of **blocks hit** is disregarded, as these blocks are directly fetched from the cache, incurring significantly lower costs.

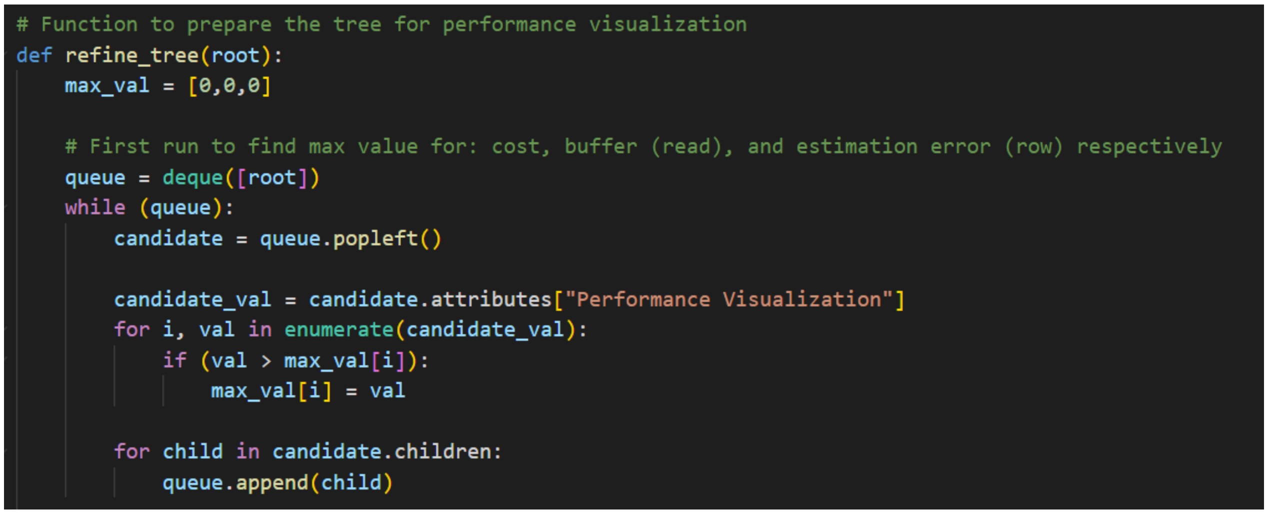


```python
## Update plan to prepare for subsequent performance visualization
est_cost = plan["Total Cost"] - children_info_dict["Total Cost"]
read_blocks = shared_read_blocks + local_read_blocks + temp_read_blocks
err = error

plan["Performance Visualization"] = (est_cost, read_blocks, err)

return annotations
```

The `annotations` will be returned by the function as string, which will then be added to the QEP node and displayed in the GUI.

Refine Tree for Visualization

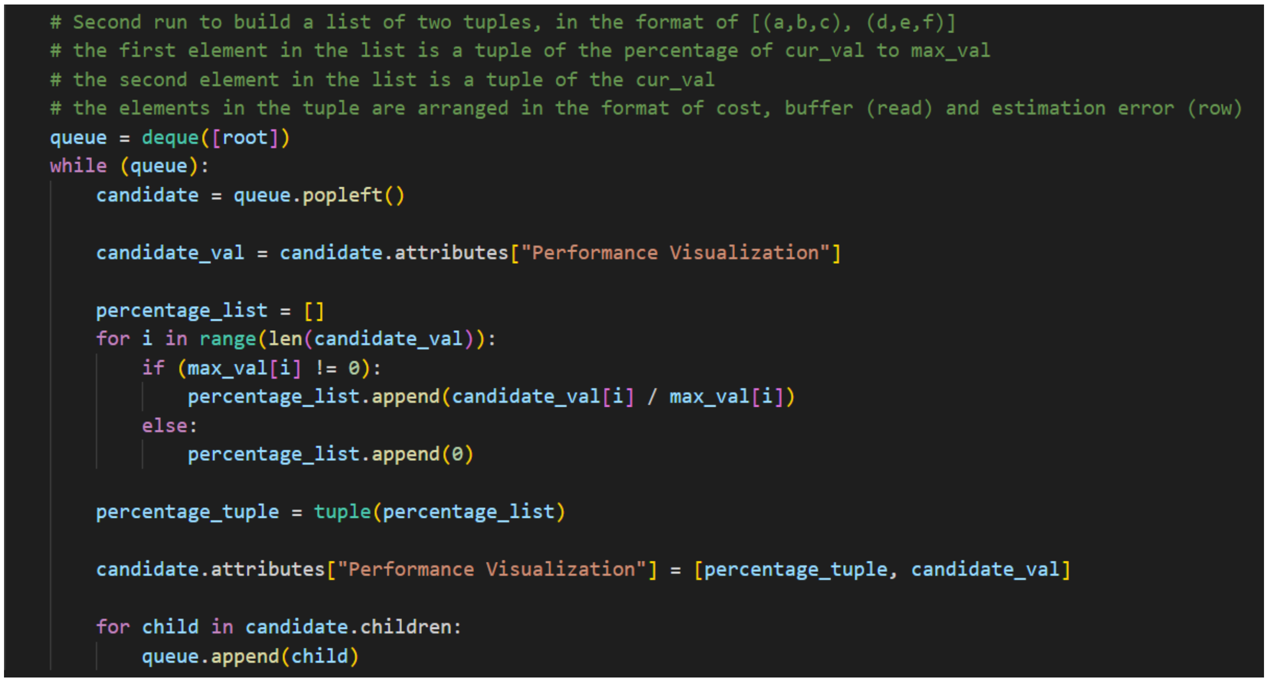
The `refine_tree()` function is developed to facilitate performance visualisation in our GUI. We traverse down the tree using a queue to find the maximum value across the QEP tree for each component, namely: estimated cost, the number of buffers read, and the error of row estimation. The `deque()` function is utilised to better simulate the operation in a queue, with significantly lower cost when popping the first element in a list.



```python
# Function to prepare the tree for performance visualization
def refine_tree(root):
    max_val = [0,0,0]

    # First run to find max value for: cost, buffer (read), and estimation error (row) respectively
    queue = deque([root])
    while (queue):
        candidate = queue.popleft()

        candidate_val = candidate.attributes["Performance Visualization"]
        for i, val in enumerate(candidate_val):
            if (val > max_val[i]):
                max_val[i] = val

        for child in candidate.children:
            queue.append(child)
```

After that, we traverse the QEP tree again using the same method as in the previous traversal to find the “percentage” for each component. For each component (namely estimated cost, the number of buffers read, and the error of row estimation), this percentage is defined as the ratio of the value of that particular node to the maximum value across the QEP tree, which will eventually be utilised for colour changing in the performance visualisation in the GUI.

Finally, we pack the tuple containing the percentage for each component, as well as the tuple containing the original value at each node, into a list. The `Performance Visualization` attribute for each node will be replaced by this list of tuples, ready for visualisation in the GUI.



```
# Second run to build a list of two tuples, in the format of [(a,b,c), (d,e,f)]
# the first element in the list is a tuple of the percentage of cur_val to max_val
# the second element in the list is a tuple of the cur_val
# the elements in the tuple are arranged in the format of cost, buffer (read) and estimation error (row)
queue = deque([root])
while (queue):
    candidate = queue.popleft()

    candidate_val = candidate.attributes["Performance Visualization"]

    percentage_list = []
    for i in range(len(candidate_val)):
        if (max_val[i] != 0):
            percentage_list.append(candidate_val[i] / max_val[i])
        else:
            percentage_list.append(0)

    percentage_tuple = tuple(percentage_list)

    candidate.attributes["Performance Visualization"] = [percentage_tuple, candidate_val]

    for child in candidate.children:
        queue.append(child)
```

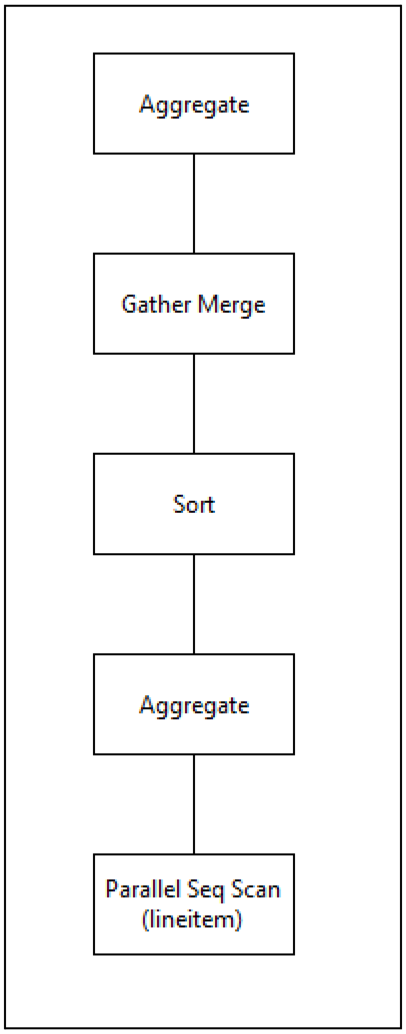
Test Cases

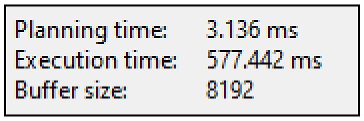
To verify the functionality of our application and confirm that it correctly presents the expected results on the graphical user interface (GUI), we utilised a variety of queries. In this report, we have chosen to show a few specific queries as demonstrations of our functional application.

Please note that, due to space limitations, only the QEP tree and overall information about that particular query are displayed in this report. Other aspects, such as performance visualisation or disk block access visualisation, are not included.

Test Case 1

```
select
   l_returnflag,
   l_linestatus,
   sum(l_quantity) as sum_qty,
   sum(l_extendedprice) as sum_base_price,
   sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
   sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
   avg(l_quantity) as avg_qty,
   avg(l_extendedprice) as avg_price,
   avg(l_discount) as avg_disc,
   count(*) as count_order
  from
   lineitem
  where
   l_extendedprice > 100
  group by
   l_returnflag,
   l_linestatus
  order by
   l_returnflag,
   l_linestatus;
```

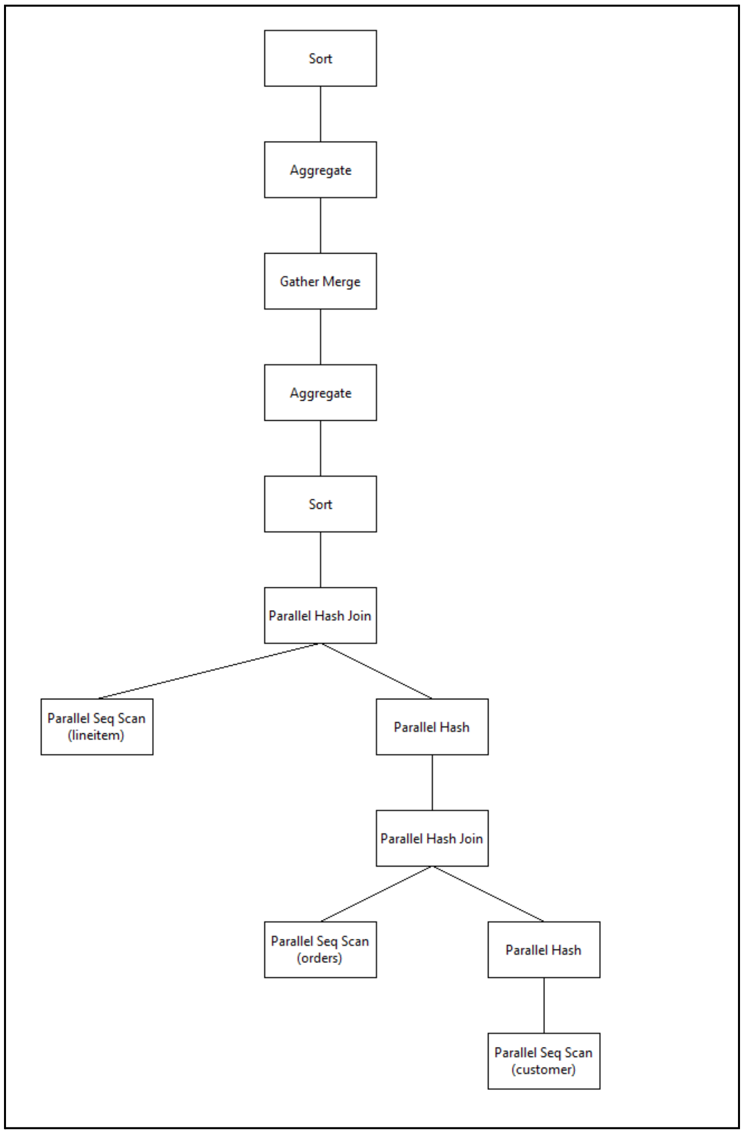


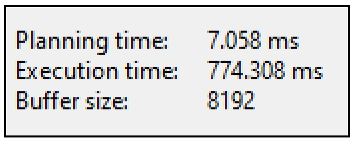


```
Planning time:    3.136 ms
Execution time:   577.442 ms
Buffer size:      8192
```

Test Case 2

```
select
   l_orderkey,
   sum(l_extendedprice * (1 - l_discount)) as revenue,
   o_orderdate,
   o_shippriority
  from
   customer,
   orders,
   lineitem
  where
   c_mktsegment = 'BUILDING'
   and c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_totalprice > 10
   and l_extendedprice > 10
  group by
   l_orderkey,
   o_orderdate,
   o_shippriority
  order by
   revenue desc,
   o_orderdate;
```

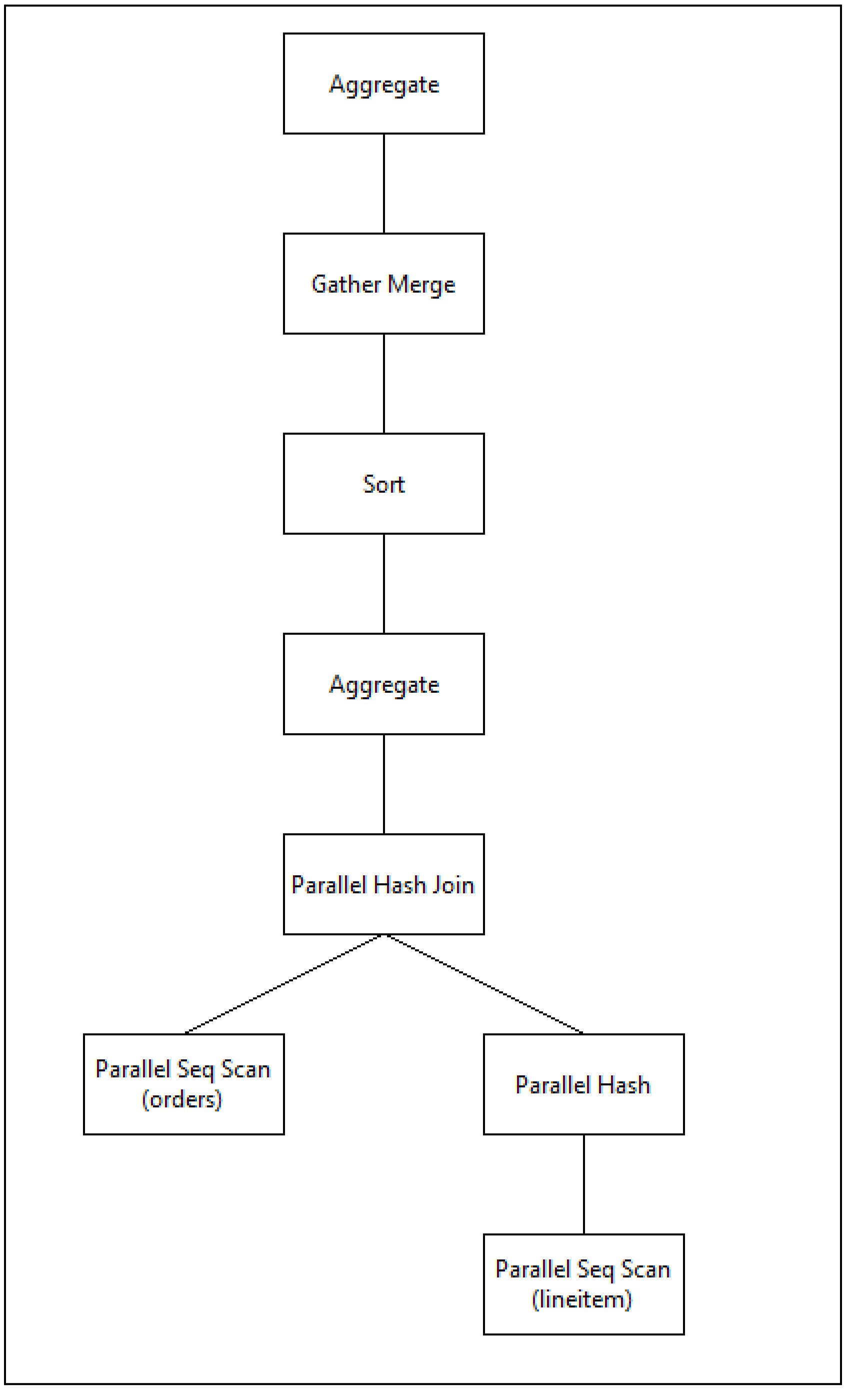


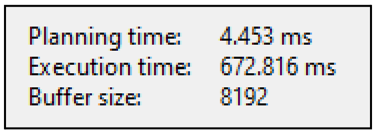


```
Planning time:    7.058 ms
Execution time:   774.308 ms
Buffer size:      8192
```

Test Case 3

```
select
   o_orderpriority,
   count(*) as order_count
  from
   orders
  where
   o_totalprice > 100
   and exists (
    select
     *
    from
     lineitem
    where
     l_orderkey = o_orderkey
     and l_extendedprice > 100
   )
  group by
   o_orderpriority
  order by
   o_orderpriority;
```

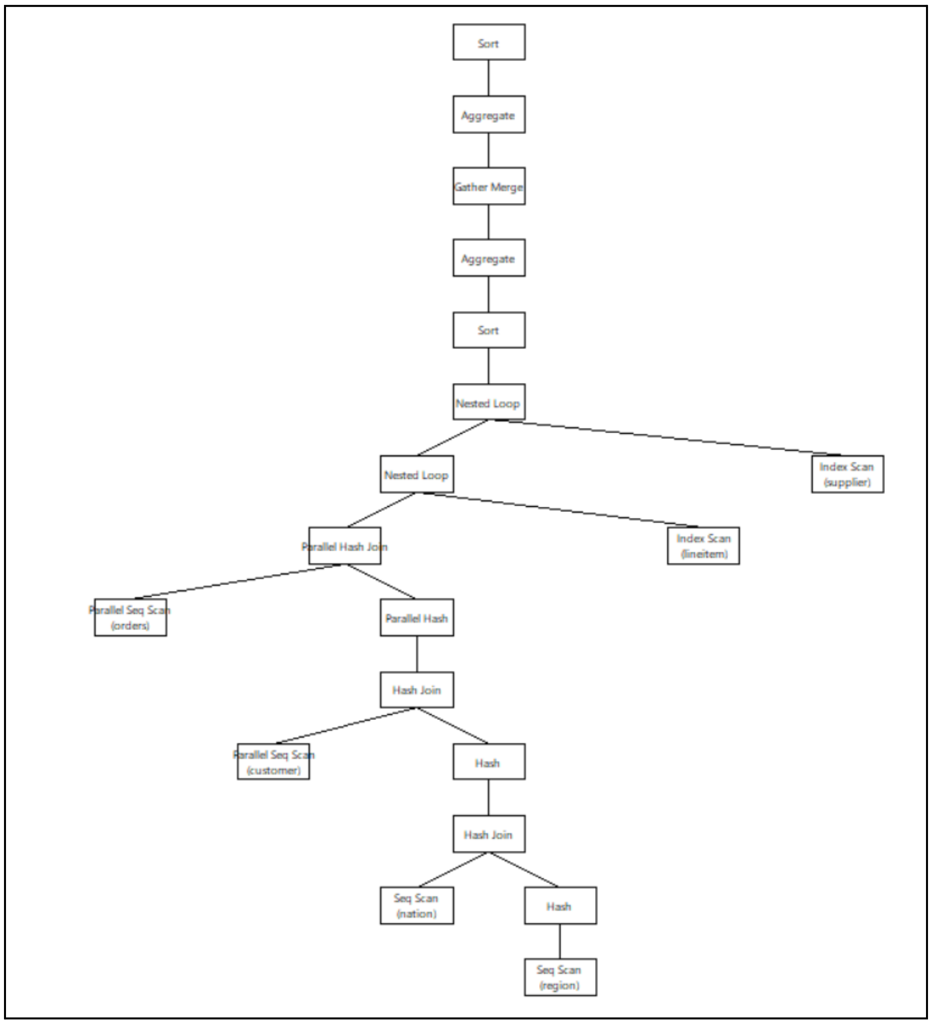


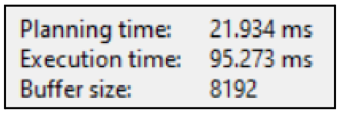


```
Planning time:     4.453 ms
Execution time:    672.816 ms
Buffer size:       8192
```

Test Case 4

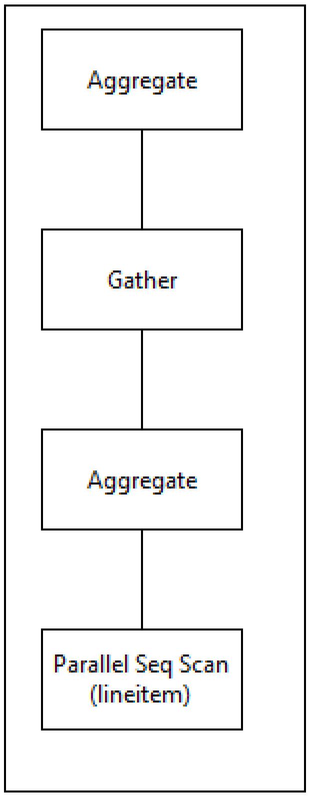
```
select
   n_name,
   sum(l_extendedprice * (1 - l_discount)) as revenue
 from
   customer,
   orders,
   lineitem,
   supplier,
   nation,
   region
 where
   c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and l_suppkey = s_suppkey
   and c_nationkey = s_nationkey
   and s_nationkey = n_nationkey
   and n_regionkey = r_regionkey
   and r_name = 'ASIA'
   and o_orderdate >= '1994-01-01'
   and o_orderdate < '1995-01-01'
   and c_acctbal > 10
   and s_acctbal > 20
 group by
   n_name
 order by
   revenue desc;
```

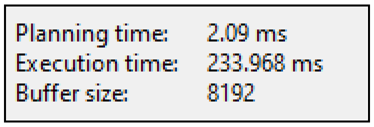



```
Planning time:     21.934 ms
Execution time:    95.273 ms
Buffer size:       8192
```

Test Case 5

```
select
     sum(l_extendedprice * l_discount) as revenue
   from
     lineitem
   where
            l_extendedprice > 100;
```

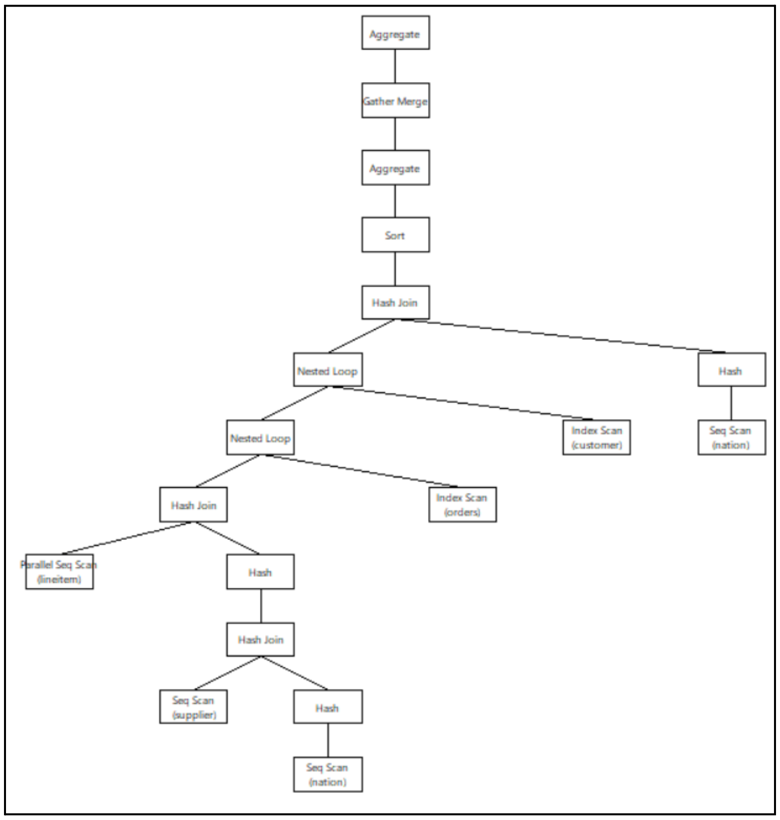


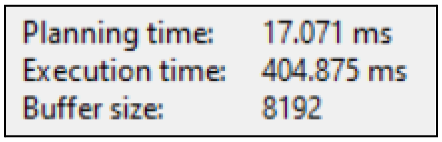


```
Planning time:     2.09 ms
Execution time:    233.968 ms
Buffer size:       8192
```

Test Case 6

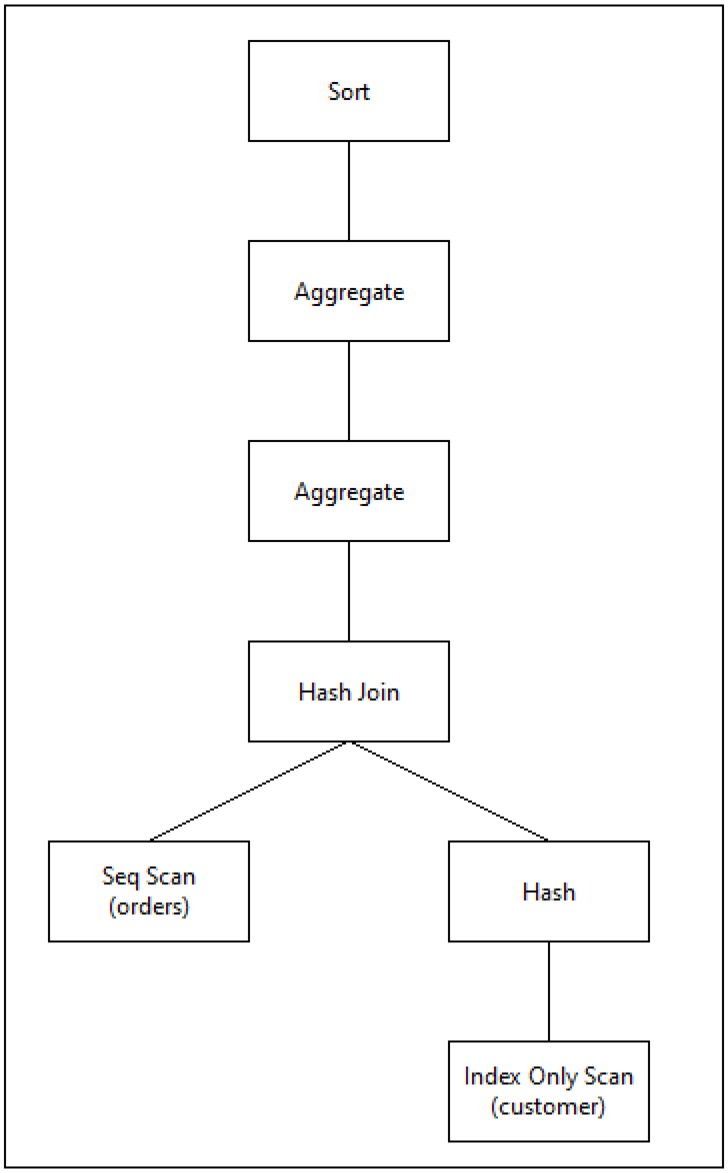
```
select
   supp_nation,
   cust_nation,
   l_year,
   sum(volume) as revenue
  from
   (
    select
     n1.n_name as supp_nation,
     n2.n_name as cust_nation,
     DATE_PART('YEAR',l_shipdate) as l_year,
     l_extendedprice * (1 - l_discount) as volume
    from
     supplier,
     lineitem,
     orders,
     customer,
     nation n1,
     nation n2
    where
     s_suppkey = l_suppkey
     and o_orderkey = l_orderkey
     and c_custkey = o_custkey
     and s_nationkey = n1.n_nationkey
     and c_nationkey = n2.n_nationkey
     and (
      (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
      or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE')
     )
     and l_shipdate between '1995-01-01' and '1996-12-31'
     and o_totalprice > 100
     and c_acctbal > 10
   ) as shipping
  group by
   supp_nation,
   cust_nation,
   l_year
  order by
   supp_nation,
   cust_nation,
   l_year;
```




Planning time: 17.071 ms
Execution time: 404.875 ms
Buffer size: 8192

Test Case 7

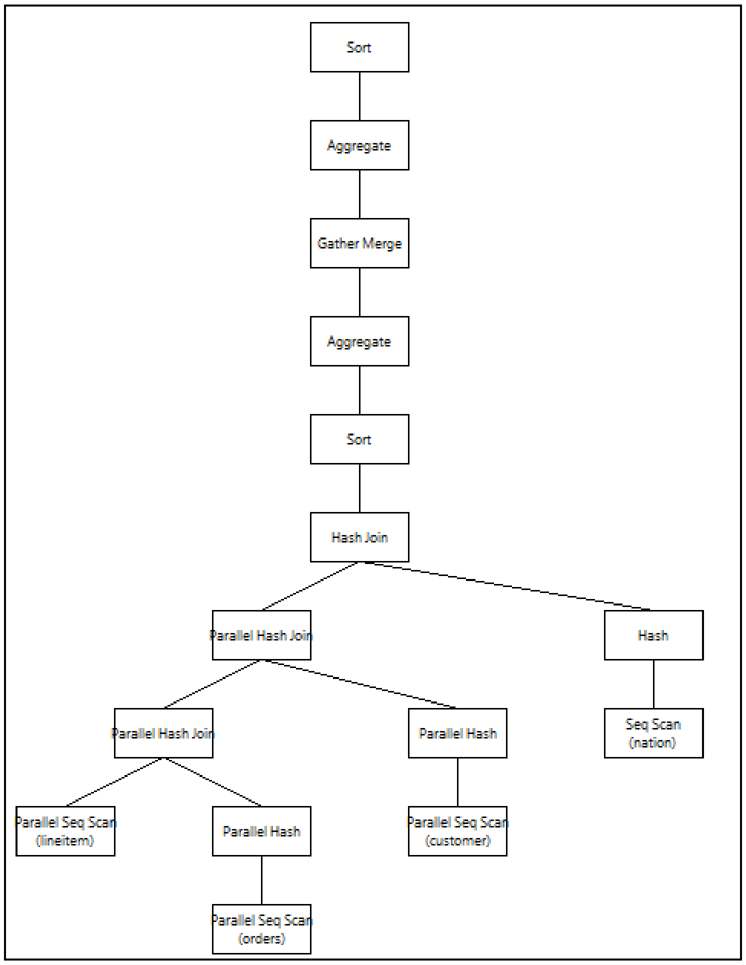
```
select
   c_count,
   count(*) as custdist
  from
   (
    select
     c_custkey,
     count(o_orderkey)
    from
     customer left outer join orders on
     c_custkey = o_custkey
    group by
     c_custkey
     ) as c_orders (c_custkey, c_count)
  group by
   c_count
  order by
   custdist desc,
   c_count desc;
```

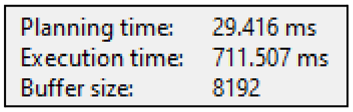


Planning time:	6.666 ms
Execution time:	3103.324 ms
Buffer size:	8192

Test Case 8

```
select
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
  from
    customer,
    orders,
    lineitem,
    nation
  where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= '1993-10-01'
    and o_orderdate < '1994-01-01'
    and c_nationkey = n_nationkey
    and c_acctbal > 10
    and l_extendedprice > 100
  group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
  order by
    revenue desc;
```

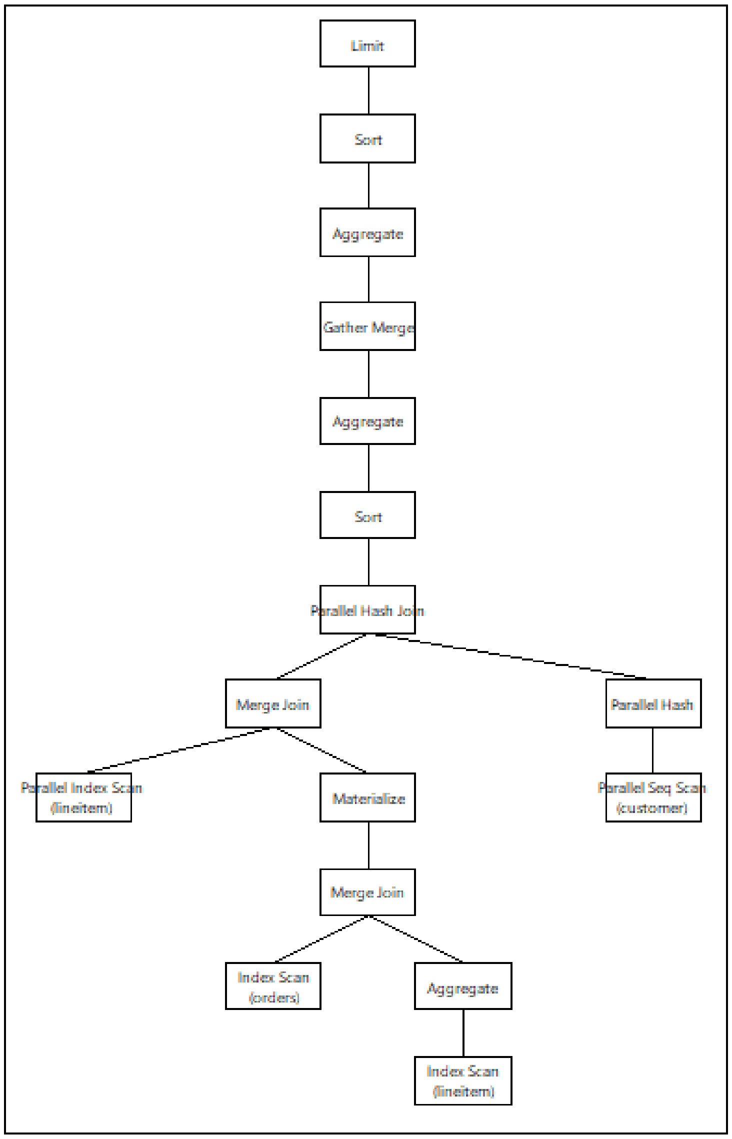




Planning time:	29.416 ms
Execution time:	711.507 ms
Buffer size:	8192

Test Case 9

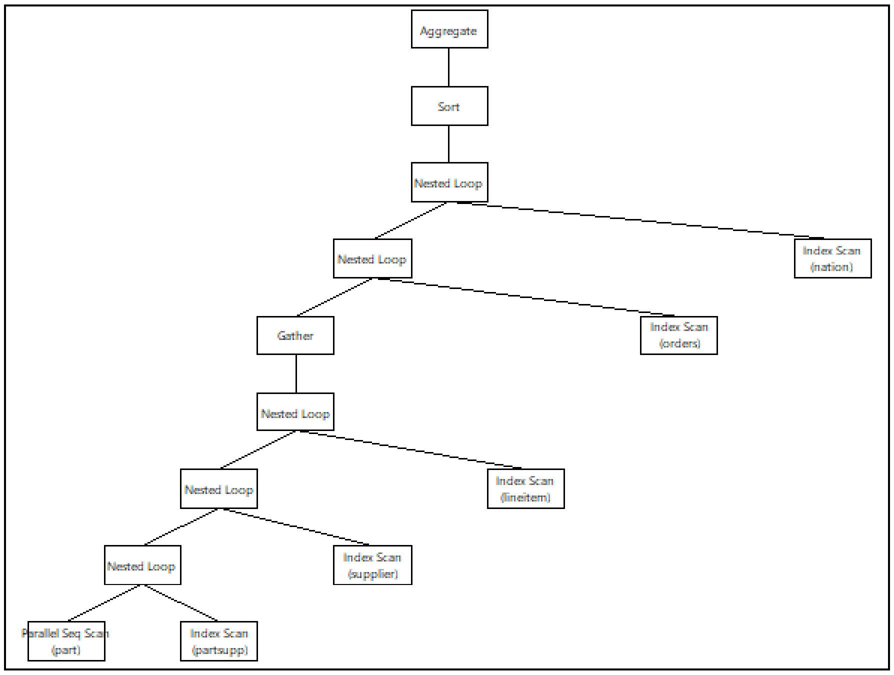
```
select
      c_name,
      c_custkey,
      o_orderkey,
      o_orderdate,
      o_totalprice,
      sum(l_quantity)
    from
      customer,
      orders,
      lineitem
    where
      o_orderkey in (
        select
            l_orderkey
          from
            lineitem
          group by
            l_orderkey having
              sum(l_quantity) > 314
              )
      and c_custkey = o_custkey
      and o_orderkey = l_orderkey
    group by
      c_name,
      c_custkey,
      o_orderkey,
      o_orderdate,
      o_totalprice
    order by
       o_totalprice desc,
       o_orderdate
```



Planning time:	0.889 ms
Execution time:	1377.636 ms
Buffer size:	8192

Test Case 10

```
select
   n_name,
   o_year,
   sum(amount) as sum_profit
 from
   (
    select
      n_name,
      DATE_PART('YEAR',o_orderdate) as o_year,
      l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
    from
      part,
      supplier,
      lineitem,
      partsupp,
      orders,
      nation
    where
      s_suppkey = l_suppkey
      and ps_suppkey = l_suppkey
      and ps_partkey = l_partkey
      and p_partkey = l_partkey
      and o_orderkey = l_orderkey
      and s_nationkey = n_nationkey
      and p_name like '%green%'
      and s_acctbal > 10
      and ps_supplycost > 100
   ) as profit
 group by
   n_name,
   o_year
 order by
   n_name,
   o_year desc;
```



```
Planning time:    18.612 ms
Execution time:   118.166 ms
Buffer size:      8192
```

Limitations

1) In the current system, when retrieving the disk blocks accessed, the join condition is ignored, resulting in a less accurate overall result.

2) Due to high complexity and insufficient time for project completion as well as program user-friendliness, our program is not able to display the blocks accessed and tuples that satisfy each respective operator in the Query Execution Plan.

3) The current implementation relies on Python libraries, posing constraints in the design of the GUI. Take *tkinter* library for example, *tkinter* library is single threaded, hence it faces challenges in handling computationally intensive tasks and long-running operations without causing the GUI to become unresponsive, potentially impacting user experience. This also bottlenecks the complexity and features we can implement for this project.

4) Our program currently only supports one query as inputs to simplify the computations and visualisation of our program.

5) Only Access Methods including Sequential Scan, Index Scan, Bitmap Heap Scan, Index Only Scan and Tid Scan are considered in our program as some other access methods do not provide access to the statistics and disk access needed.

6) The current program implementation is limited in terms of platform compatibility as it is only compatible with PostgreSQL, potentially restricting its use, scalability and user's requirement.

Future Improvements

1) While acknowledging the high complexity involved, for future enhancements, it would be beneficial to display the blocks accessed and their respective contents in the results after each operation. This would provide users with a clearer understanding of which tuples belonging to specific blocks satisfy the conditions imposed by the operators.

2) In the future, we might consider migrating the system to JavaScript and leveraging modern frameworks like ReactJS to create a more user-friendly interface and improve the overall user experience.

3) Customizable reporting can also be implemented to enable users to customise the statistic displayed either visually or in plaintext with comprehensive explanation, allowing for a tailored presentation of information based on individual preferences and specific analytical needs.

4) In the future, we can consider adding animations of the different operations (Hash Join, Parallel Seq Scan etc.) executed in the query, helping users to visualise the query executed in more detail.