CZ4031 Database Systems Principles
Project 2 Group 31 Report

Wang Chuhan	U2022829K
Leong Ruo Qing	U1921880A
Jerome Chew	U2021304E
Karanam Akshit	U2020311E
Lau Chen Yi Wynne	U2020016B

Content

Content **2**

1. Overview **3**

1.1 Project Overview 3

1.2 Instructions 3

2. QEP and AQP Generation **4**

2.1 Processing of SQL query 5

2.2 QEP and AQP Generation and Pre-Processing 7

2.3 Mapping SCAN operation to SQL query line 8

2.4 Mapping JOIN operation to SQL query line 9

3. QEP and AQP comparision **12**

3.1 Comparing the costs 12

3.2 Annotations 13

4. Graphical User Interface (GUI) **15**

5. Limitations **16**

6. Appendix **17**

1. Overview

1.1 Project Overview

Structured Query Language (SQL) queries are used to search relational databases to retrieve information for different tasks. Database management systems (DBMS) softwares such as PostgreSQL will execute a query execution plan (QEP) to process each query, which is chosen from a large number of alternative query plans (AQP). However, the system usually returns only the results of the query with no information on what operations are being carried out and why.

Thus, this project aims to connect an SQL query and its query plan-related information by retrieving relevant information from a QEP and representative AQPs to annotate the corresponding SQL query. The annotation shows how different components of the query are executed and why the operators are chosen among other alternatives.

Our group has implemented algorithms to identify and process the raw query plan output and retrieve the corresponding QEP and AQPs. Additionally, we used visualization tools such as tkinter to show the annotations side-by-side with the SQL query. More of this will be explained in the later part of the report.

1.2 Instructions

Setting up:

- Change the host, database, user and password in the database.ini file according to the user's setup.
- Create the tables and import data for each table in PostgreSQL database (pgAdmin4)

Step 1: Click the run button for project.py.

Step 2: Input the query in the 'ENTER QUERY HERE' box. Please take note that the SQL query should only be in **lower case** for the algorithm to work except for keywords, for example 'SELECT', 'FROM' and 'AND', which can be in **upper case**.

Step 3: Click the EXECUTE button to generate the result. Do note that, on average, the processing time for each query is 15 seconds before the results are displayed.

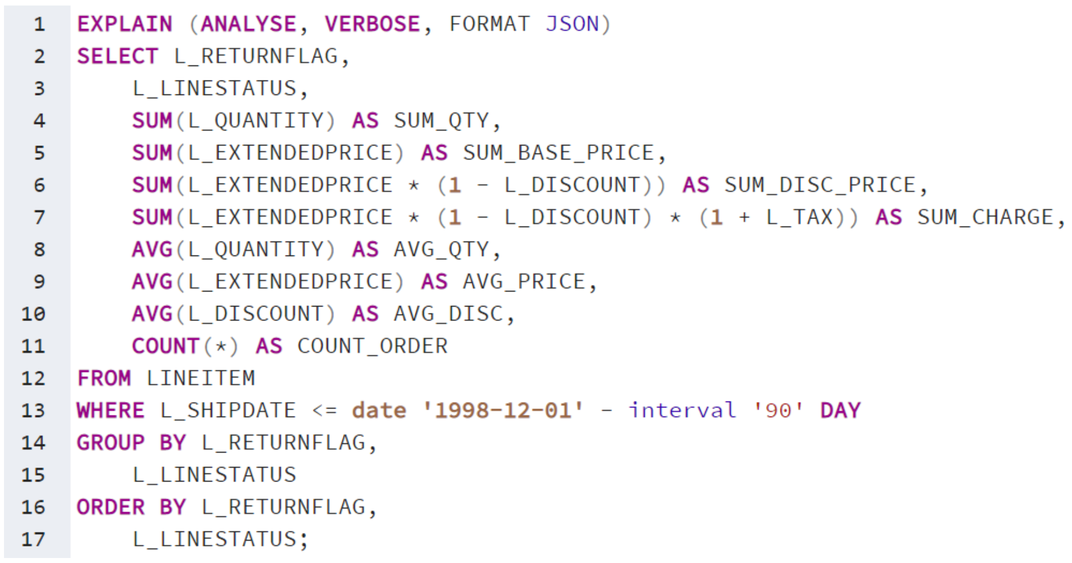
Step 4: Move the scroll bar to view the Formatted Query and the corresponding Annotations.

There are some example queries that are given below in the apendix that can be used for.

2. QEP and AQP Generation

We used pgAdmin4 to visualise the QEP and the AQPs. To generate the QEP and AQPs, we used `EXPLAIN (ANALYSE, VERBOSE, FORMAT JSON)`. This would generate the plan in the json format for our analysis.

To generate the QEP, we simply type the query and pass it to the backend. Below is an example of the query typed into the query tool of pgAdmin4.



```sql
EXPLAIN (ANALYSE, VERBOSE, FORMAT JSON)
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_SHIPDATE <= date '1998-12-01' - interval '90' DAY
GROUP BY L_RETURNFLAG,
    L_LINESTATUS
ORDER BY L_RETURNFLAG,
    L_LINESTATUS;
```

To generate the AQPs, we make the optimizer choose different plans from the QEP. For example, if we do not want the plan to include bitmap scans, we will then use `SET LOCAL enable_bitmapscan = off;` to not take bitmap scans into consideration when creating the plan. The same can be done with index scans, sequential scans, nested loop joins, merge joins and hash joins by replacing `enable_bitmapscan` with the other configuration parameters. In our project, the number of different AQPs that we can generate is 63 by disabling the types of scan and join operations, but in order to take care of the response time to users, we only select a few such that we would be able to make good cost calculations for each type of join operation.

2.1 Processing of SQL query

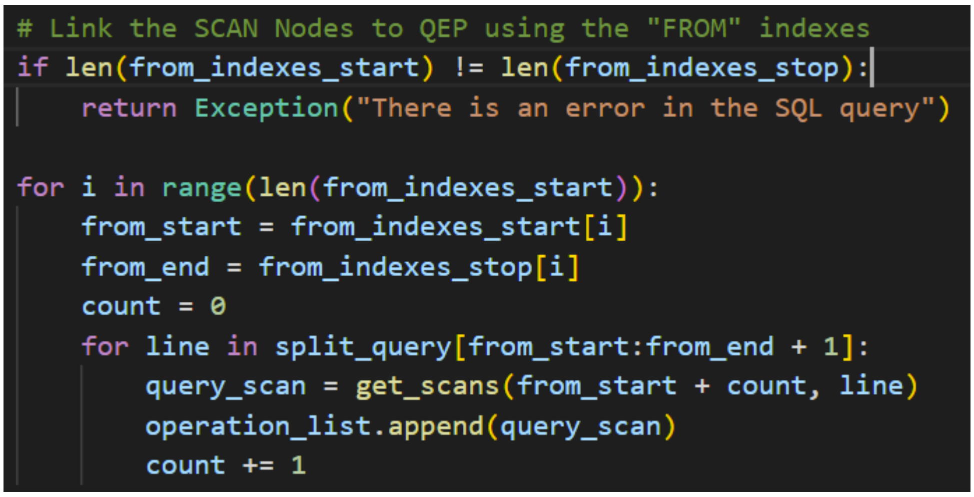
STEP 1

Firstly, we started off by formatting and then splitting each SQL query into an array of strings for easier annotation. To do this, we first split the SQL into single statements using the `sqlparse.split()` function. Then, using `sqlparse.format()`, we formatted the SQL statements. To split each SQL statement into different strings for annotation, we used `query.splitlines()`, where the split is made at every new line.

STEP 2

The main purpose of this project relates to the cost of joins and scans. Hence, we grouped the SQL statements into 2 main parts - "FROM" and "WHERE" . This is achieved by getting the start and end index of the "FROM" and "WHERE" clauses.

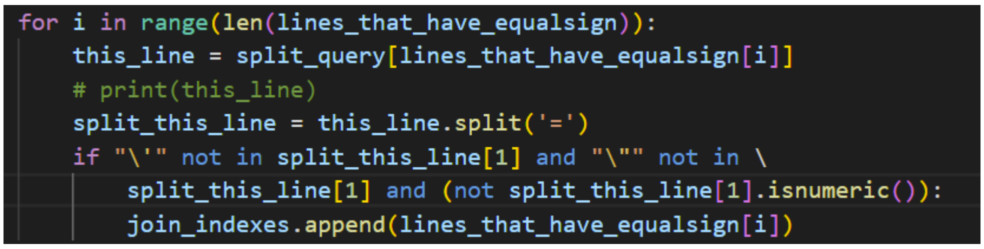
Using the start and end indexes of the "FROM" clause, mapping between the SCAN nodes of the QEP and each line in this clause is performed using the algorithm below.



```
# Link the SCAN Nodes to QEP using the "FROM" indexes
if len(from_indexes_start) != len(from_indexes_stop):
    return Exception("There is an error in the SQL query")

for i in range(len(from_indexes_start)):
    from_start = from_indexes_start[i]
    from_end = from_indexes_stop[i]
    count = 0
    for line in split_query[from_start:from_end + 1]:
        query_scan = get_scans(from_start + count, line)
        operation_list.append(query_scan)
        count += 1
```

STEP 3

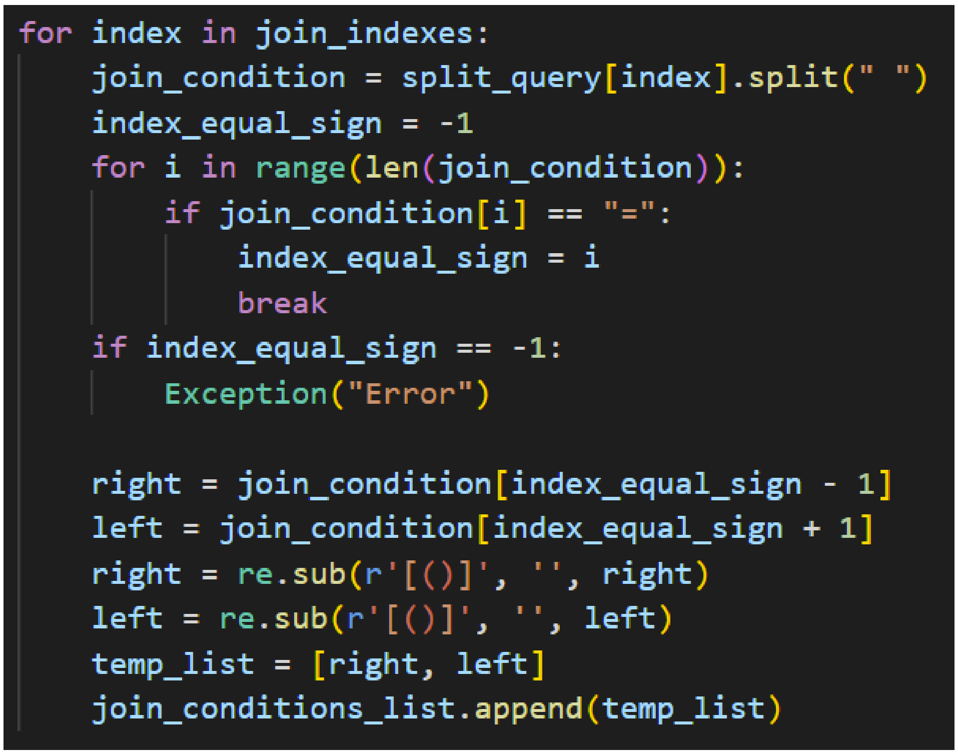
For the "WHERE" clause however, preprocessing has to be done to find which lines are used in join operations. A "WHERE" clause typically contains many equality checks such as - [a_key = 10, a_name = 'Tom', a_name = b_name, a_count > 10]. To just select the join operations, we first filter out all the lines that contain "=" , but not ("<" and ">"). Now, we have all lines that just contain the "=" sign. We then check if the right side of the equality sign contains any quotations ("/') or if it's numeric, if these conditions are satisfied, the lines are removed from consideration. At this stage we have all the indexes of the lines that contain the join conditions. The code below is used for this purpose.



```python
for i in range(len(lines_that_have_equalsign)):
    this_line = split_query[lines_that_have_equalsign[i]]
    # print(this_line)
    split_this_line = this_line.split('=')
    if "\'" not in split_this_line[1] and "\"" not in \
        split_this_line[1] and (not split_this_line[1].isnumeric()):
        join_indexes.append(lines_that_have_equalsign[i])
```

STEP 4:

For each of the index of the join conditions obtained in step 3, we split the line such that the left and right join conditions are placed in a list format like this: [right,left]. We then store these formatted join conditions in `join_conditions_list`.



```python
for index in join_indexes:
    join_condition = split_query[index].split(" ")
    index_equal_sign = -1
    for i in range(len(join_condition)):
        if join_condition[i] == "=":
            index_equal_sign = i
            break
    if index_equal_sign == -1:
        Exception("Error")

    right = join_condition[index_equal_sign - 1]
    left = join_condition[index_equal_sign + 1]
    right = re.sub(r'[()]', '', right)
    left = re.sub(r'[()]', '', left)
    temp_list = [right, left]
    join_conditions_list.append(temp_list)
```

STEP 5:

For each join condition obtained in the step 4, we map the join condition to the various JOIN Nodes in the QEP.

2.2 QEP and AQP Generation and Pre-Processing

STEP 1: Generate QEP and AQPs

To link our python environment to the PostgreSQL server, a publicly available library, psycopg2 was used. QEP and AQPs can be generated as explained above by disabling and enabling certain parameters. While exploring, we realised that PostgreSQL creates an index on the primary key of the relation. This is quite useful information as it was observed that index-scan and bitmap index-scan were useful for index based nested loop joins.

There are 4 types of joins we want to find the cost of: Nested-Loop (NL) Join, Sort-Merge Join, Hash Join and Index Based NL Join. The optimal QEP is generated when none of the algorithms as described above are switched off, while that is not the case for AQPs. Our general strategy to get the AQPs for a particular type of join is to switch off the other 3 types. The table below shows the parameters switched off for each type of join.

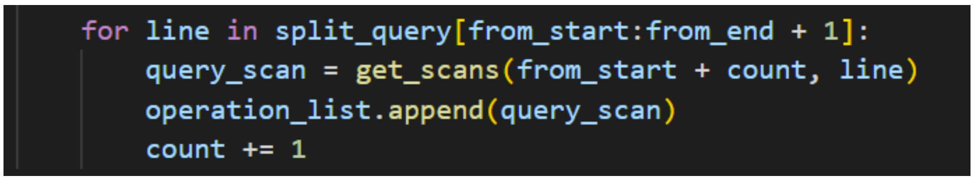
Type of join to enable	**Parameters disabled**
Hash Join	`nestloop, mergejoin, indexscan, bitmapscan`
Merge Join	`nestloop, hashjoin, indexscan, bitmapscan`
NL Join	`hashjoin, mergejoin, indexscan, bitmapscan`
Index NL Join	`mergejoin, hashjoin`

STEP 2: Converting the JSON response to useful lists

The JSON response from the QEP and AQPs are converted to lists. First a queue structure is used to create a tree structure. In-order traversal is then performed on the tree structure, and each node is added to the list in this step. These lists are then used to map the processed SQL query with the Plan Nodes.

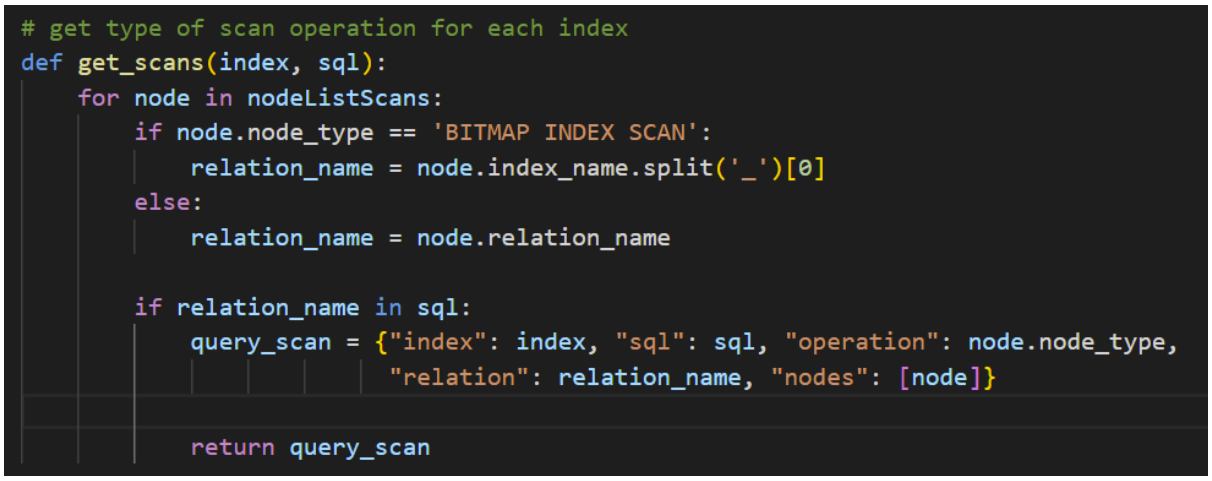
2.3 Mapping SCAN operation to SQL query line

To map each SCAN operation to the line in the SQL query it corresponds to, we go through each line in the SQL query with indexes that belong to the “FROM” clause. We perform the `get_scans` function on each line.



```python
    for line in split_query[from_start:from_end + 1]:
        query_scan = get_scans(from_start + count, line)
        operation_list.append(query_scan)
        count += 1
```

In the `get_scans` function, we first go through each node in the list of SCAN nodes to get the relation name for each node. We then check if this relation name exists in the line of SQL query passed in, and return the data in the specified format as `query_scan`.

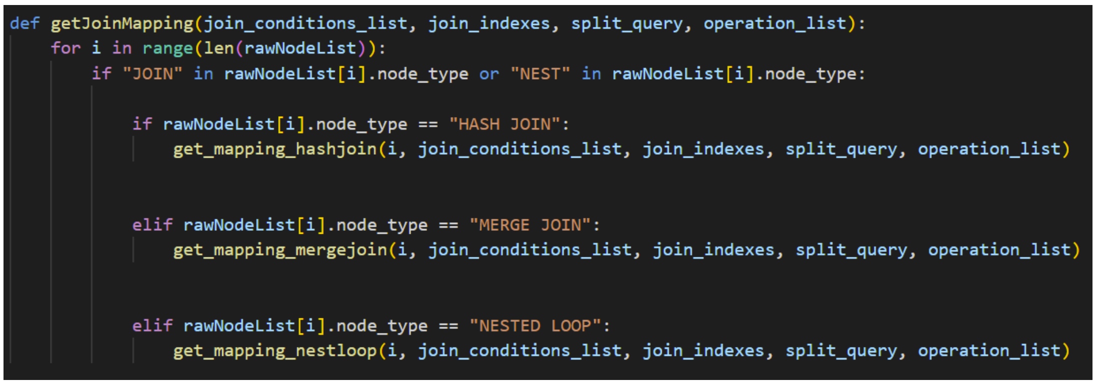


```python
# get type of scan operation for each index
def get_scans(index, sql):
    for node in nodeListScans:
        if node.node_type == 'BITMAP INDEX SCAN':
            relation_name = node.index_name.split('_')[0]
        else:
            relation_name = node.relation_name

        if relation_name in sql:
            query_scan = {"index": index, "sql": sql, "operation": node.node_type,
                          "relation": relation_name, "nodes": [node]}

            return query_scan
```

Lastly, we append the information into `operation_list`.

2.4 Mapping JOIN operation to SQL query line

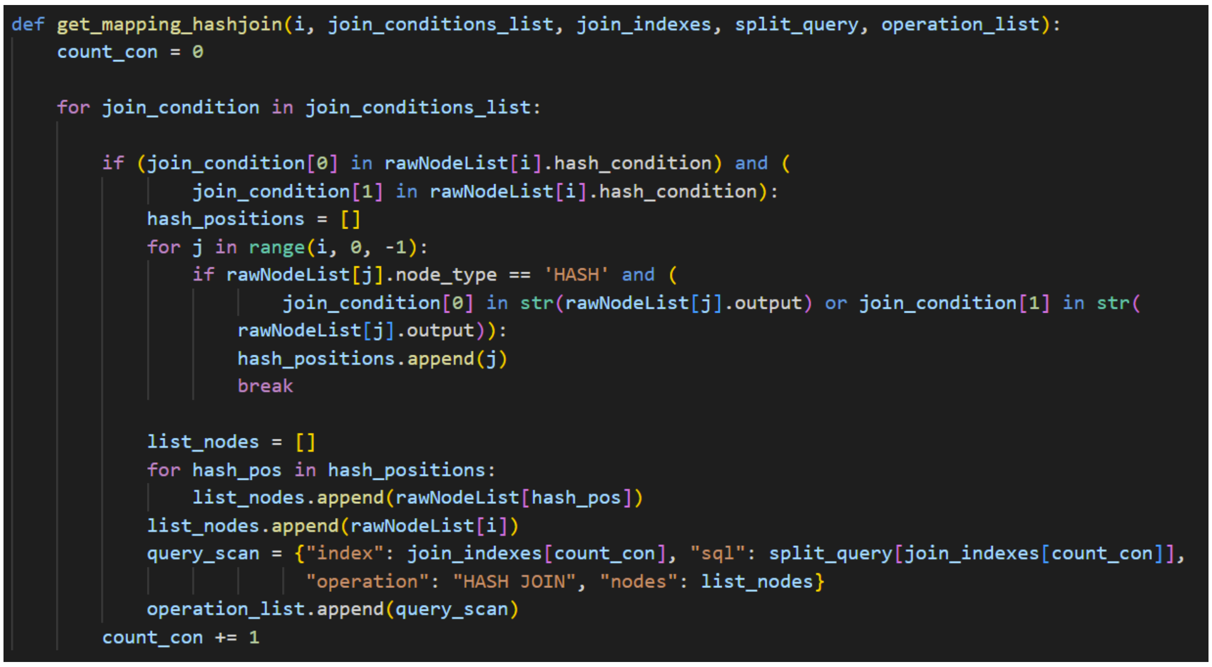
To map each JOIN operation to the line in the SQL query it corresponds to, we go through the JSON output of the QEP and map it to the filtered lines in the "WHERE" clause using the algorithm below. We do so by looking for each JOIN node and then comparing the conditions to every filtered line of the "WHERE" clause.



```
def getJoinMapping(join_conditions_list, join_indexes, split_query, operation_list):
    for i in range(len(rawNodeList)):
        if "JOIN" in rawNodeList[i].node_type or "NEST" in rawNodeList[i].node_type:

            if rawNodeList[i].node_type == "HASH JOIN":
                get_mapping_hashjoin(i, join_conditions_list, join_indexes, split_query, operation_list)


            elif rawNodeList[i].node_type == "MERGE JOIN":
                get_mapping_mergejoin(i, join_conditions_list, join_indexes, split_query, operation_list)


            elif rawNodeList[i].node_type == "NESTED LOOP":
                get_mapping_nestloop(i, join_conditions_list, join_indexes, split_query, operation_list)
```

HASH JOIN

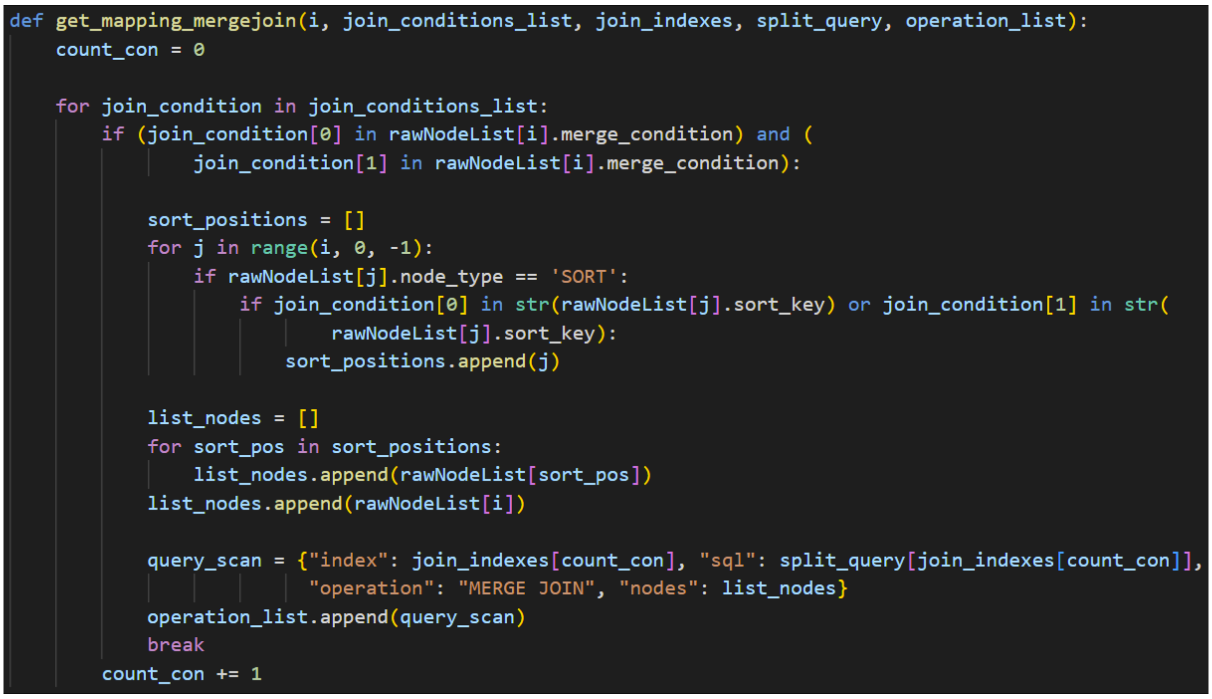
For the HASH JOINs, we first check if the left and right condition of each filtered line in the "WHERE" clause appears in the `hash_condition` of the HASH JOIN node. Then, we append the information into `operation_list`. Also, it has to be noted that before a HASH JOIN, hashing is usually performed. Hence, the HASH node that occurs before the HASH JOIN is also essential when making cost calculations. Both the HASH JOIN and the HASH nodes are mapped to that particular line.



```
def get_mapping_hashjoin(i, join_conditions_list, join_indexes, split_query, operation_list):
    count_con = 0

    for join_condition in join_conditions_list:

        if (join_condition[0] in rawNodeList[i].hash_condition) and (
                join_condition[1] in rawNodeList[i].hash_condition):
            hash_positions = []
            for j in range(i, 0, -1):
                if rawNodeList[j].node_type == 'HASH' and (
                        join_condition[0] in str(rawNodeList[j].output) or join_condition[1] in str(
                    rawNodeList[j].output)):
                    hash_positions.append(j)
                    break

            list_nodes = []
            for hash_pos in hash_positions:
                list_nodes.append(rawNodeList[hash_pos])
            list_nodes.append(rawNodeList[i])
            query_scan = {"index": join_indexes[count_con], "sql": split_query[join_indexes[count_con]],
                          "operation": "HASH JOIN", "nodes": list_nodes}
            operation_list.append(query_scan)
        count_con += 1
```

MERGE JOIN

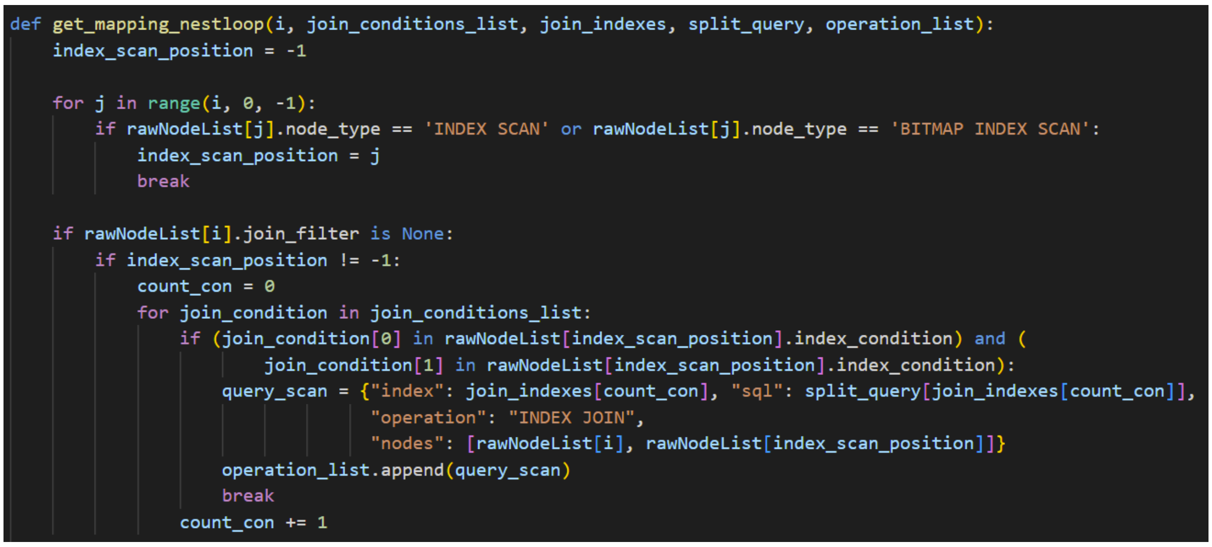
For MERGE JOINs, we also do something similar to how we processed the HASH JOINs, but instead of looking out for HASH nodes, we look out for the SORT nodes.



```python
def get_mapping_mergejoin(i, join_conditions_list, join_indexes, split_query, operation_list):
    count_con = 0

    for join_condition in join_conditions_list:
        if (join_condition[0] in rawNodeList[i].merge_condition) and (
                join_condition[1] in rawNodeList[i].merge_condition):

            sort_positions = []
            for j in range(i, 0, -1):
                if rawNodeList[j].node_type == 'SORT':
                    if join_condition[0] in str(rawNodeList[j].sort_key) or join_condition[1] in str(
                            rawNodeList[j].sort_key):
                        sort_positions.append(j)

            list_nodes = []
            for sort_pos in sort_positions:
                list_nodes.append(rawNodeList[sort_pos])
            list_nodes.append(rawNodeList[i])

            query_scan = {"index": join_indexes[count_con], "sql": split_query[join_indexes[count_con]],
                          "operation": "MERGE JOIN", "nodes": list_nodes}
            operation_list.append(query_scan)
            break
        count_con += 1
```

NL Join and Index NL Joins

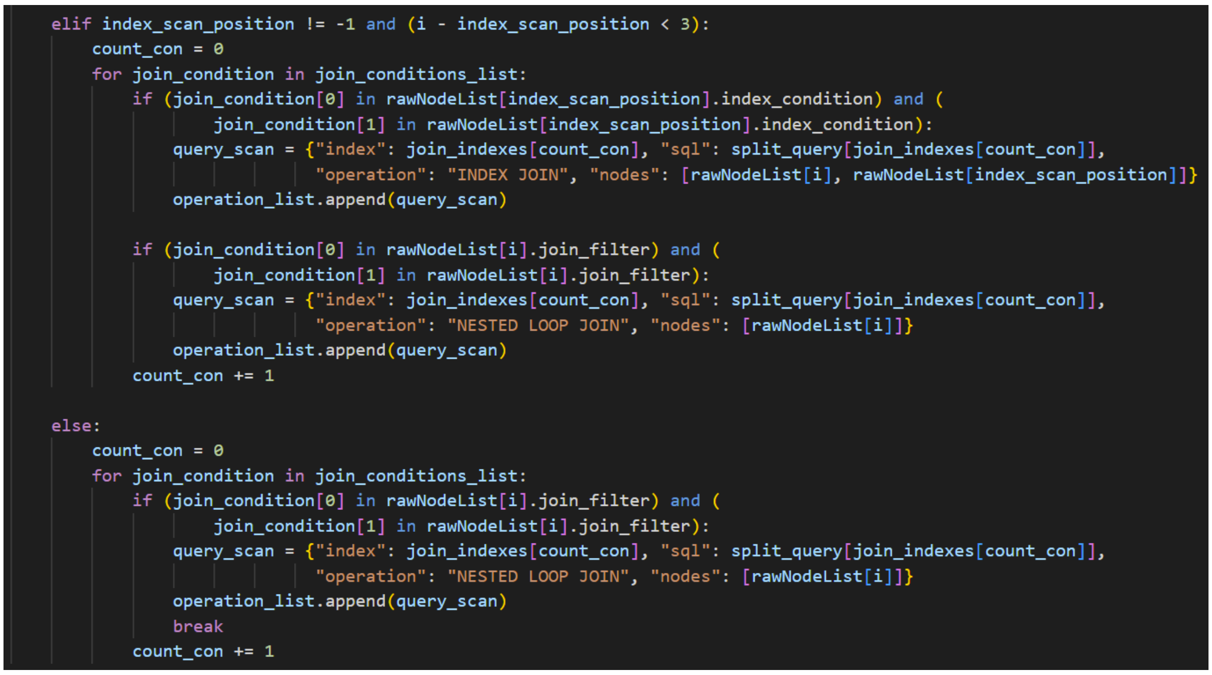
For NESTED LOOP nodes, things are a little complicated. There are 3 scenarios that can occur. The first being that some of them do not have conditions in their `join_filter`, which suggests that this operation would be a Index-Based NL join, hence we need to also find the first previously occurring INDEX SCAN or BITMAP INDEX SCAN node to get the `index_condition`, which actually corresponds to the join condition.



```python
def get_mapping_nestloop(i, join_conditions_list, join_indexes, split_query, operation_list):
    index_scan_position = -1

    for j in range(i, 0, -1):
        if rawNodeList[j].node_type == 'INDEX SCAN' or rawNodeList[j].node_type == 'BITMAP INDEX SCAN':
            index_scan_position = j
            break

    if rawNodeList[i].join_filter is None:
        if index_scan_position != -1:
            count_con = 0
            for join_condition in join_conditions_list:
                if (join_condition[0] in rawNodeList[index_scan_position].index_condition) and (
                        join_condition[1] in rawNodeList[index_scan_position].index_condition):
                    query_scan = {"index": join_indexes[count_con], "sql": split_query[join_indexes[count_con]],
                                  "operation": "INDEX JOIN",
                                  "nodes": [rawNodeList[i], rawNodeList[index_scan_position]]}
                    operation_list.append(query_scan)
                    break
                count_con += 1
```

The second scenario is that the NESTED LOOP node contains a `join_filter`, which suggests that normal nested-loop join is performed on the `join_filter`. However, if there is a previous INDEX SCAN or BITMAP INDEX SCAN just before the NESTED LOOP JOIN, which would mean that in this particular scenario, the NESTED LOOP node is performing 2 join operations.

The last scenario occurs when the NESTED LOOP node contains a `join_filter` and also does not contain a INDEX SCAN or BITMAP-SCAN previously. This would correspond to a normal NL join.



```
elif index_scan_position != -1 and (i - index_scan_position < 3):
    count_con = 0
    for join_condition in join_conditions_list:
        if (join_condition[0] in rawNodeList[index_scan_position].index_condition) and (
                join_condition[1] in rawNodeList[index_scan_position].index_condition):
            query_scan = {"index": join_indexes[count_con], "sql": split_query[join_indexes[count_con]],
                          "operation": "INDEX JOIN", "nodes": [rawNodeList[i], rawNodeList[index_scan_position]]}
            operation_list.append(query_scan)

        if (join_condition[0] in rawNodeList[i].join_filter) and (
                join_condition[1] in rawNodeList[i].join_filter):
            query_scan = {"index": join_indexes[count_con], "sql": split_query[join_indexes[count_con]],
                          "operation": "NESTED LOOP JOIN", "nodes": [rawNodeList[i]]}
            operation_list.append(query_scan)
        count_con += 1

else:
    count_con = 0
    for join_condition in join_conditions_list:
        if (join_condition[0] in rawNodeList[i].join_filter) and (
                join_condition[1] in rawNodeList[i].join_filter):
            query_scan = {"index": join_indexes[count_con], "sql": split_query[join_indexes[count_con]],
                          "operation": "NESTED LOOP JOIN", "nodes": [rawNodeList[i]]}
            operation_list.append(query_scan)
            break
        count_con += 1
```

Taking these 3 scenarios into account, the respective nodes are mapped similar to that of the previous joins.

3. QEP and AQP comparision

3.1 Comparing the costs

All the QEP and AQPs generated are mapped to their respective indexes on the SQL query as described in the previous sub-sections. The table below shows an illustration of how the comparison between each node can look like.

Type of Plan	Index of SQL	Nodes	Total Cost	Applicable
QEP	1	IndexScan, NL Join	IndexScan.cost + NLJoin.cost	
AQP 1 (on MERGEJOIN)	1	Sort, MergeJoin	Sort.cost + MergeJoin.cost	Yes
AQP 2 (on NEST JOIN)	1	NL Join	NLJoin.cost	Yes
AQP 3 (on HASH JOIN)	1	IndexScan, NL Join	IndexScan, NL Join	No

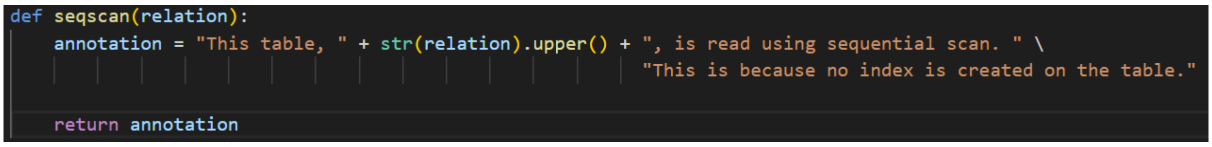
For each Index of SQL that contains a join (retrieved from SQL Pre-precossing), the various QEP and AQPs are appended into a list. A comparison is made between the QEP and the AQPs by comparing the total cost, this would be reflected in the annotations.

If the Nodes in the QEP and AQP contain the same type of nodes,we consider that type of AQP not applicable. For example, referring to the table above, HASH JOIN is not applicable to the QEP. In addition, sometimes, the time taken to retrieve queries takes a long time, hence, we have implemented a 10 seconds timeout. This is usually observed for the AQPs on NESTED LOOP Join.

3.2 Annotations

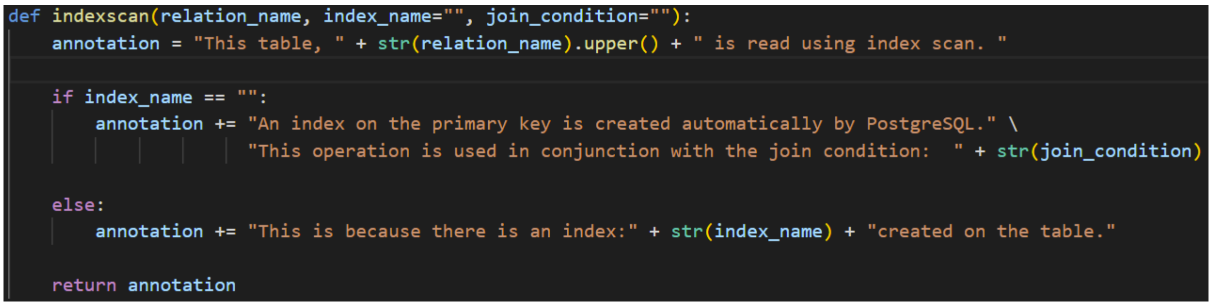
For each type of scan and join, there will be a different set of annotations to be displayed on the GUI. For JOIN operations, the QEP and AQPs will be compared to get the annotations.

For SEQ SCAN, the `relation_name` is passed into the annotation, which explains why the table is read using sequential scan.



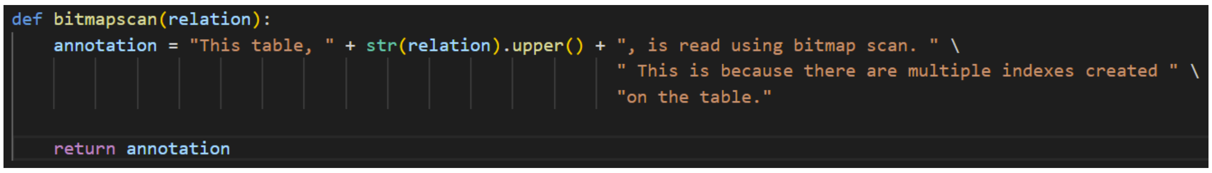
```python
def seqscan(relation):
    annotation = "This table, " + str(relation).upper() + ", is read using sequential scan. " \
                                                          "This is because no index is created on the table."

    return annotation
```

For INDEX SCAN, the `relation_name` is passed into the annotation. The explanation will differ based on the parameters passed in. If the `index_name` is not empty, the annotation will state the name of the index created on the table. Else, since an index scan may be used for a future join operation, the annotation will state that the particular index scan is used in conjunction with the `join_condition` passed in.



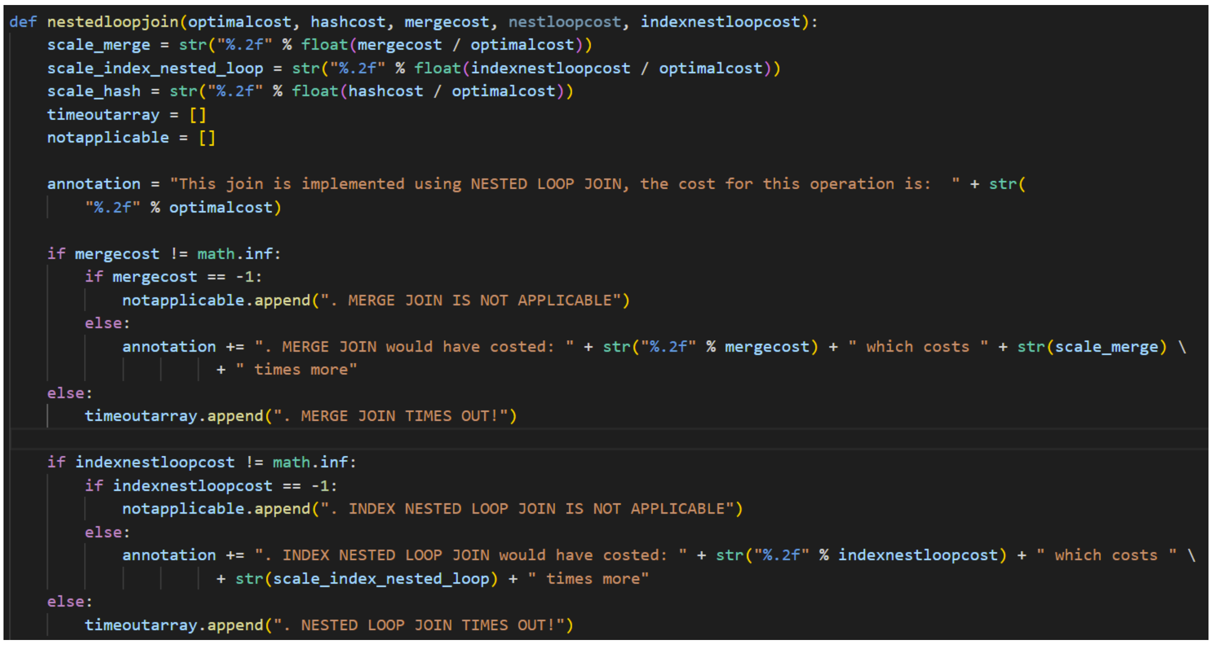
```python
def indexscan(relation_name, index_name="", join_condition=""):
    annotation = "This table, " + str(relation_name).upper() + " is read using index scan. "

    if index_name == "":
        annotation += "An index on the primary key is created automatically by PostgreSQL." \
                      "This operation is used in conjunction with the join condition:  " + str(join_condition)

    else:
        annotation += "This is because there is an index:" + str(index_name) + "created on the table."

    return annotation
```

For BITMAP SCAN, the `relation_name` is passed into the annotation, which explains why the table is read using bitmap scan.

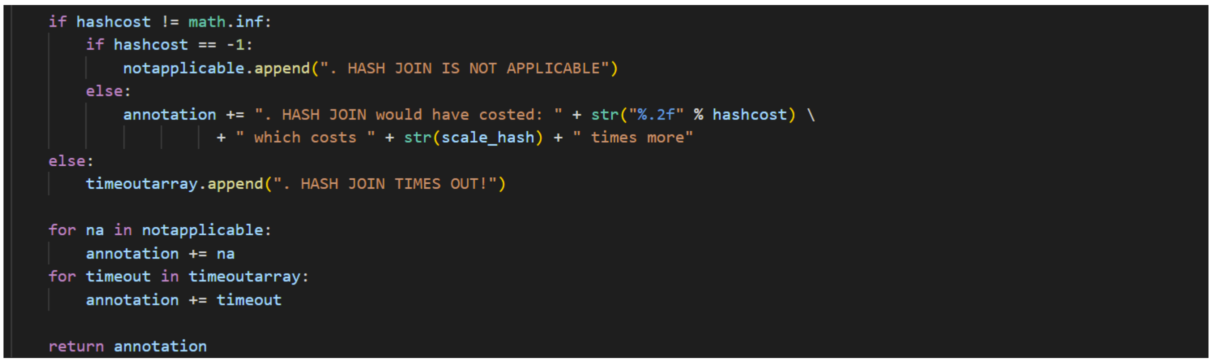


```python
def bitmapscan(relation):
    annotation = "This table, " + str(relation).upper() + ", is read using bitmap scan. " \
                                                          " This is because there are multiple indexes created " \
                                                          "on the table."

    return annotation
```

For NESTED LOOP JOIN, we compare the respective costs of the other JOIN operations against the cost of the NESTED LOOP JOIN. We then append the annotation to the `annotation` array. However, there are some cases where other types of joins are not applicable, hence we append annotations such as ". MERGE JOIN IS NOT APPLICABLE" to the `notapplicable` array. In other cases, some types of joins might take a very long time to run, hence we would stop running for that join if it exceeds a run time of 10 seconds and we will append annotations such as ". MERGE JOIN TIMES OUT!" to the `timeoutarray` array.



```python
def nestedloopjoin(optimalcost, hashcost, mergecost, nestloopcost, indexnestloopcost):
    scale_merge = str("%.2f" % float(mergecost / optimalcost))
    scale_index_nested_loop = str("%.2f" % float(indexnestloopcost / optimalcost))
    scale_hash = str("%.2f" % float(hashcost / optimalcost))
    timeoutarray = []
    notapplicable = []

    annotation = "This join is implemented using NESTED LOOP JOIN, the cost for this operation is:  " + str(
        "%.2f" % optimalcost)

    if mergecost != math.inf:
        if mergecost == -1:
            notapplicable.append(". MERGE JOIN IS NOT APPLICABLE")
        else:
            annotation += ". MERGE JOIN would have costed: " + str("%.2f" % mergecost) + " which costs " + str(scale_merge) \
                          + " times more"
    else:
        timeoutarray.append(". MERGE JOIN TIMES OUT!")

    if indexnestloopcost != math.inf:
        if indexnestloopcost == -1:
            notapplicable.append(". INDEX NESTED LOOP JOIN IS NOT APPLICABLE")
        else:
            annotation += ". INDEX NESTED LOOP JOIN would have costed: " + str("%.2f" % indexnestloopcost) + " which costs " \
                          + str(scale_index_nested_loop) + " times more"
    else:
        timeoutarray.append(". NESTED LOOP JOIN TIMES OUT!")
```



```python
    if hashcost != math.inf:
        if hashcost == -1:
            notapplicable.append(". HASH JOIN IS NOT APPLICABLE")
        else:
            annotation += ". HASH JOIN would have costed: " + str("%.2f" % hashcost) \
                          + " which costs " + str(scale_hash) + " times more"
    else:
        timeoutarray.append(". HASH JOIN TIMES OUT!")

    for na in notapplicable:
        annotation += na
    for timeout in timeoutarray:
        annotation += timeout

    return annotation
```

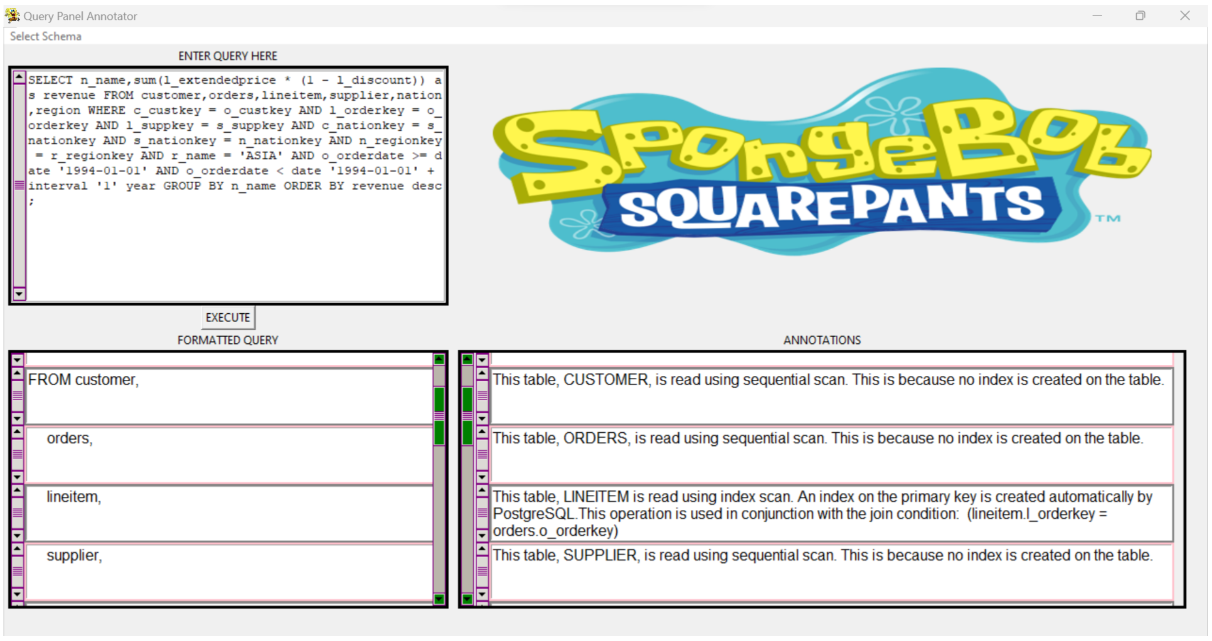
For the remaining JOIN operations, annotations will be similar to that for NL JOIN as explained above.

4. Graphical User Interface (GUI)

We made use of the python tkinter module to create our Graphical User Interface.

At the top left hand corner, we have a drop down menu labeled “Select Schema” which would allow the user to select between different database schemas.

We also created 3 frames for different uses. Firstly, we have a frame for the user to input the query which is labeled “ENTER QUERY HERE”. Below, we have 2 frames side by side, the frame on the left would display the formatted query which are query lines broken down for different annotations. The frame on the right would then contain the corresponding annotations for each line of query. The execute button would run our code to format and analyze the query while producing the following annotations in the annotations frame. Since the query and its annotations could be quite long, we have added a scrollbar which is coloured green for the user to scroll through the different lines of query to view the corresponding annotations on the right hand side.



```
SELECT n_name,sum(l_extendedprice * (1 - l_discount)) a
s 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 >= d
ate '1994-01-01' AND o_orderdate < date '1994-01-01' +
interval '1' year GROUP BY n_name ORDER BY revenue desc
;
```

FROM customer,	This table, CUSTOMER, is read using sequential scan. This is because no index is created on the table.
orders,	This table, ORDERS, is read using sequential scan. This is because no index is created on the table.
lineitem,	This table, LINEITEM is read using index scan. An index on the primary key is created automatically by PostgreSQL.This operation is used in conjunction with the join condition: (lineitem.l_orderkey = orders.o_orderkey)
supplier,	This table, SUPPLIER, is read using sequential scan. This is because no index is created on the table.

5. Limitations

There are many possible query plans that can be derived from a single query and the limitation of our software is that we do not compare the QEP with all AQPs by testing all combinations of enable/disable conditions due to the response time to the user. We decide to limit the number of AQPs by several specific conditions, for example, making all the nodes in the query plan choose a different type of join/scan. Generally, we feel that it would be better if we could individually change the type of join for each node, rather than the whole sub-space. This would allow us to get more accurate information in lesser amount of time. A heuristic approach could be developed to find the best AQPs, where individual nodes are targeted.

For query plans involving JOIN operations, some of them take a long time to generate. As mentioned above, we have set a timeout of 10 seconds for these cases. Hence, another limitation is that the exact costs for these JOIN operations will not be calculated, and will just be taken as very high.

We have noticed that there are some queries that have transitive join conditions, where A=B and B=C. The query planner sometimes makes joins between A=C first before making joins between B=C. Although the end outcome is the same, it is not possible to map such “new” conditions to the SQL query, as it does not really exist in the query.

Our system currently does not work with the JOIN keyword, and nested JOINS. However, such a thing can be quite easily refactored in, as the base operations still remain the same. But more complicated and thorough Query Preprocessing has to be performed, which was not possible in the given tight schedule.

Simple subqueries generally work with our system, but more testing and analysis would be required for more complicated nested queries.

The last limitation involves the SQL query input to the GUI. The SQL query should only be in lower case for the algorithm to work. Only keywords, for example ‘SELECT’, ‘FROM’ and ‘AND’, can be in upper case.

6. Appendix

The queries below were retrieved from the TPC-H V3.0.1\dbgen\queries file with slight modifications such that they work with pgAdmin4.

Query1:

```
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_shipdate <= date '1998-12-01' - interval '90' DAY
 GROUP BY l_returnflag,
         l_linestatus
 ORDER BY l_returnflag,
         l_linestatus;
```

Query2:

```
SELECT o_orderpriority,
        count(*) AS order_count
 FROM orders
 WHERE o_orderdate >= date '1996-03-01'
   AND EXISTS
     (SELECT *
      FROM lineitem
      WHERE l_orderkey = o_orderkey
        AND l_commitdate < l_receiptdate)
GROUP BY o_orderpriority
ORDER BY o_orderpriority
LIMIT 1;
```

Query3:

```
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_orderdate < date '1995-03-15'
   AND l_shipdate > date '1995-03-15'
 GROUP BY l_orderkey,
          o_orderdate,
          o_shippriority
 ORDER BY revenue DESC,
          o_orderdate
 LIMIT 20;
```

Query5:

```
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 >= date '1994-01-01'
   AND o_orderdate < date '1994-01-01' + interval '1' YEAR
 GROUP BY n_name
 ORDER BY revenue DESC;
```

Query6:

```
SELECT sum(l_extendedprice * l_discount) AS revenue
 FROM lineitem
 WHERE l_shipdate >= date '1994-01-01'
   AND l_shipdate < date '1994-01-01' + interval '1' YEAR
   AND l_discount BETWEEN 0.06 - 0.01 AND 0.06 + 0.01
```

```
  AND l_quantity < 24;
```

Query10:

```
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 >= date '1993-10-01'
   AND o_orderdate < date '1993-10-01' + interval '3' MONTH
   AND l_returnflag = 'R'
   AND c_nationkey = n_nationkey
 GROUP BY c_custkey,
         c_name,
         c_acctbal,
         c_phone,
         n_name,
         c_address,
         c_comment
 ORDER BY revenue DESC
 LIMIT 20;
```

Query12:

```
 SELECT l_shipmode,
       sum(CASE
               WHEN o_orderpriority = '1-URGENT'
                    OR o_orderpriority = '2-HIGH' THEN 1
               ELSE 0
           END) AS high_line_count,
       sum(CASE
               WHEN o_orderpriority <> '1-URGENT'
                    AND o_orderpriority <> '2-HIGH' THEN 1
               ELSE 0
           END) AS low_line_count
 FROM orders,
      lineitem
 WHERE o_orderkey = l_orderkey
   AND l_shipmode in ('MAIL',
                  'SHIP')
   AND l_commitdate < l_receiptdate
   AND l_shipdate < l_commitdate
   AND l_receiptdate >= date '1994-01-01'
   AND l_receiptdate < date '1994-01-01' + interval '1' YEAR
 GROUP BY l_shipmode
 ORDER BY l_shipmode;
```

Query14:

```
SELECT 100.00 * sum(CASE
                     WHEN p_type like 'PROMO%' THEN l_extendedprice * (1 - l_discount)
                     ELSE 0
                 END) / sum(l_extendedprice * (1 - l_discount)) AS promo_revenue
 FROM lineitem,
      part
 WHERE l_partkey = p_partkey
   AND l_shipdate >= date '1995-09-01'
   AND l_shipdate < date '1995-09-01' + interval '1' MONTH;
```

Query 18:

```
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) > 312)
           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
LIMIT 100;
```

Query19:

```
SELECT sum(l_extendedprice* (1 - l_discount)) AS revenue
FROM lineitem,
    part
WHERE (p_partkey = l_partkey
      AND p_brand = 'Brand#12'
      AND p_container in ('SM CASE',
                  'SM BOX',
                  'SM PACK',
                  'SM PKG')
      AND l_quantity >= 1
      AND l_quantity <= 1 + 10
      AND p_size BETWEEN 1 AND 5
      AND l_shipmode in ('AIR',
                  'AIR REG')
      AND l_shipinstruct = 'DELIVER IN PERSON')
  OR (p_partkey = l_partkey
     AND p_brand = 'Brand#23'
     AND p_container in ('MED BAG',
                  'MED BOX',
                  'MED PKG',
                  'MED PACK')
     AND l_quantity >= 10
     AND l_quantity <= 10 + 10
     AND p_size BETWEEN 1 AND 10
     AND l_shipmode in ('AIR',
                 'AIR REG')
     AND l_shipinstruct = 'DELIVER IN PERSON')
  OR (p_partkey = l_partkey
     AND p_brand = 'Brand#34'
     AND p_container in ('LG CASE',
                  'LG BOX',
                  'LG PACK',
                  'LG PKG')
     AND l_quantity >= 20
     AND l_quantity <= 20 + 10
     AND p_size BETWEEN 1 AND 15
     AND l_shipmode in ('AIR',
                 'AIR REG')
     AND l_shipinstruct = 'DELIVER IN PERSON');
```