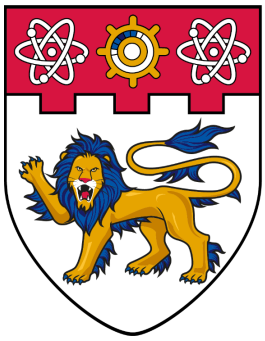


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4031 Database System Principles

Project 2 Report

Group 3

Members		
Name	Matric number	Project contribution
Colin Tan G-Hao	U1922153D	20%
Lim ZhengWei Trevor	U2022401K	20%
Muhammad Ihsan Bin Mohammad Azmi	U2021160B	20%
Nguyen Vinh Quang	U2020371D	20%
Pandey Pratyush Kumar	U1923491H	20%

Table of contents

1 Introduction

For this project, we were tasked to design and implement an efficient algorithm and user-friendly graphical user interface that takes in an SQL query as input and returns an annotated SQL query that describes the execution of various components of the query.

This report will cover the theory behind our algorithm, as we observe the plans created by PostgreSQL for an SQL query, analysing how the database management system (DBMS) selects a specific Query Execution Plan (QEP) amongst the Alternative Query Plans (AQPs).

This report also covers our program implementation. This includes AQP/QEP tree generation, how the join and scan conditions are extracted from the query and mapped to the plan, how the relevant justifications are computed and allocated, and finally how the curated query and annotations are displayed on our created GUI.

2 Pipeline Overview

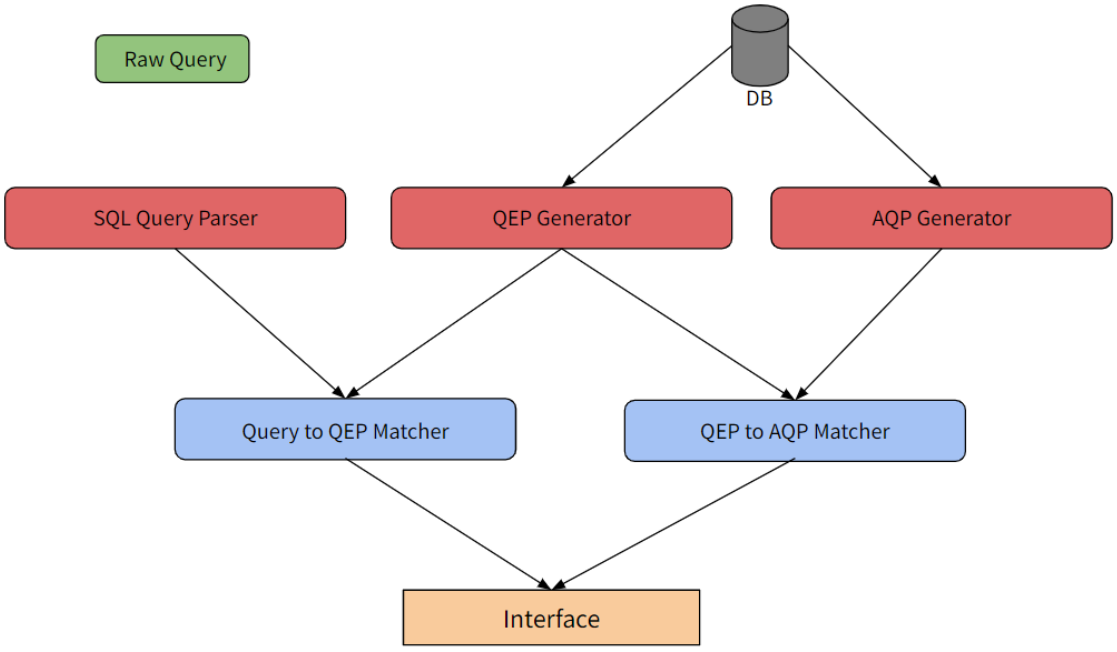


Figure 1: Architecture of the application

Our annotation generator is divided into a variety of components and we will be looking into each component in detail. To summarise, we follow a 3 tier layered architecture.

1. The layer in red is the **Data access layer**, with data being transferred from the database and raw query given by the user. It is also the pre-processing layer for the query and query execution plan.
2. The layer in blue is the **Controller layer**. The algorithm logic resides in this layer.
3. The final layer is the **Interface layer**. This layer interacts with the UI to display results to the end user.

3 Data Access layer

3.1 SQL Query parser

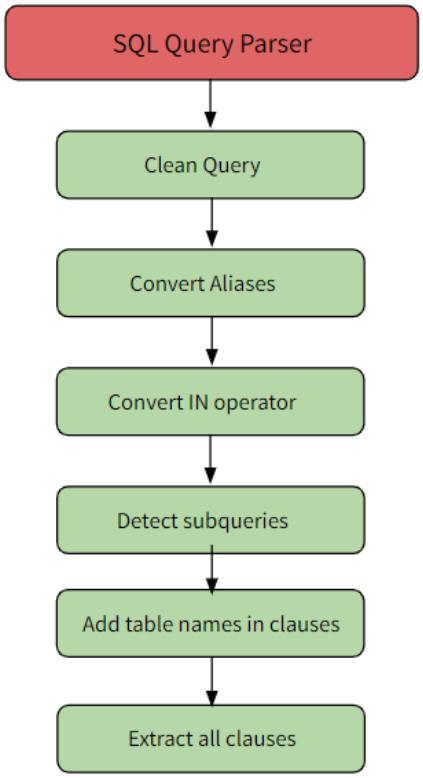


Figure 2: Subroutines in SQL Query Parse

The figure above shows the different subroutines in the ***SQL Query Parser*** component. The ultimate objective is to extract all the parts of the query that we need to annotate in the form of a list of strings. Before we do that we need to do a couple of pre-processing steps for which we use the *"sqlparser"* library.

1. **Clean query:**
   This function helps us to clean and format the query by converting keywords to uppercase, remove unnecessary spaces and extract only the first query in case there are multiple queries. This is essential when parsing and displaying the query string on the UI.

2. **Convert Aliases:**
   This function helps us to create a map of Alias as the key and the actual name as the value. For this we leverage the *"sqlparser"* library. In the process, we try to identify identifiers and the "AS" keyword in the query string. We do this because the query optimizer substitutes the actual name for the alias in the scan or join condition in the query execution plan.
   Example,
   Query:

```
select
```

```
  n_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue
```

Dictionary output:

```
{
    "revenue": "sum(l_extendedprice * (1 - l_discount))"
}
```

3. **Convert IN operator**
   This function helps in converting IN operators to a Join with an equality condition. We realised that the query optimizer recognizes an IN operator as a join condition. Hence, we annotate the IN operator by converting it to the equivalent join condition which is seen in the query execution plan.

4. **Detect the sub-queries**
   The function helps detect and number the subqueries in the order of occurrence. This is essential because we realised that POSTGRES query optimizer appends the sub-query number to the table name in all conditions.
   For example, when executing a join in the first subquery that is subquery number is 1, the filter condition for a sequential scan would be :

```
"region_1.r_name <> 'AMERICA'"
```

   The “region” table became region_1.

5. **Add table names**
   As we can see from the filter condition used earlier, table name is prepended to a column name in the conditions. Hence, in order to replicate this we prepend the following to all column names seen in the query :

```
 "{table_name}_{sub-query-number}"
```

6. **Extract all clauses**
   Now after all the pre-processing steps, we extract all the clauses in “Having” and in “where”. These clauses are then sent across to the QEP and query matcher.

3.2 QEP Generator

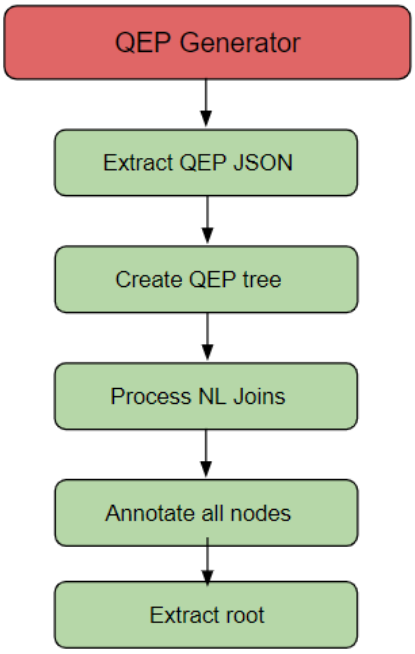


Figure 3: Subroutines in QEP Generator

The figure above shows the different subroutines in the ***QEP Generator*** component.The end result is the root of the query execution plan in the form of a tree data structure. Before that the following pre-processing is performed:

1. **Extract QEP Json**
   This function helps retrieve the query execution plan for a sql query by prepending the "***EXPLAIN (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON)***" to the raw string and running the query on the database directly.

2. **Create QEP tree**
   This function parses the json retrieved from the databases and creates a tree data structure out of it. The class performing this function stores the root of the tree for further pre-processing.

3. **Process NL Joins**
   After analysing a couple of queries with NL joins, we realised that there is no node in the Query Execution plan with the node type "Index based Nested Loop Join". However, there are Nested loop join nodes without any filter condition or join condition. This happens because the join condition gets pushed to "Index Scan" and becomes the "Index condition". If such a case occurs, we push the Index condition upwards to "Nested Loop" join and we change the node type to "Index based Nested Loop Join"

4. **Annotate all nodes**

Now we annotate all nodes based on rules like for example:
When annotating a join node, we provide the following information:

1. Join Algorithm
2. Actual Join condition
3. Filter condition if any
4. Intermediate results (found by doing a simple BFS search for subsequent joins or scans)

When annotating a scan node, we provide the following information:

1. Scan Algorithm
2. Index Condition, if any
3. Relation Name (If no relation name is found, we skip)
4. Filter condition, if any

These annotations are added as meta-data to every node object in the tree.

Note: The third and fourth subroutines described above are performed in a single tree traversal to avoid redundancy.

5. **Extract root**
   The root of the the pre-processed query execution plan tree is now sent to the Query and QEP matcher for generating annotations

3.3 AQP Generator

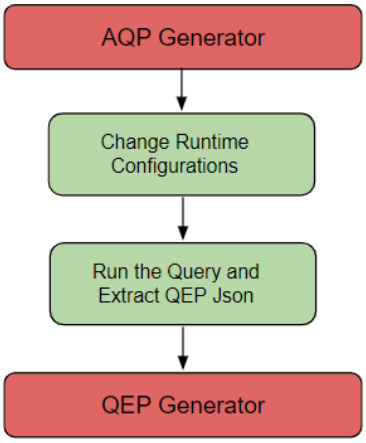


Figure 4: Subroutines in AQP Generator

There are 3 types of join planner and 5 types of scan planner. In order to get a list of AQPs, we send the query to the database iteratively, each time enabling only one planner (among join planners or scan planners) and disabling the rest (among join planners or scan planners), as opposed to the QEP where all join and scan methods are enabled. For instance, we would ask the database to perform the same query, once only using hash joins, once only using merge joins, and once only using nested loops (all executed without disabling any scan planner).

We enable or disable different algorithms by enabling/disabling the runtime algorithms. There are a total of 8 config variables we can enable or disable. All possible combinations would be

$2^8 - 2$ (excluding enabling all and disabling all). Hence, we only retrieve the one-hot combinations for Join (3 constants) and Scan (5 constants). Total = 8 Query plans.

After changing the runtime configurations, we invoke the QEP generator routine and append all the different QEP trees in an AQP List.

4. Controller Layer

4.1 QEP and AQP matcher

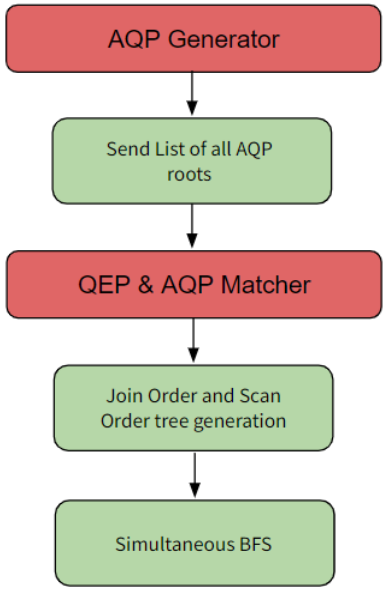


Figure 5: Subroutines in QEP and AQP Matcher

With an input query, the database will execute a Query Execution Plan that may be close to ideal. However, there are many other possible plans generated and their costs estimated before deciding on one to perform the execution on. In this section, we explore the reasoning behind choosing one plan over another. To do so, we force the database to perform the query multiple times, each using a different method. We then compare the costs with the chosen Query Execution Plan that was used to execute the query.
Ultimately, a justification is appended to the meta-data stored in the node object by the routine.

4.1.1 Join Order trees and Scan order trees

We iteratively generate ***join-order and scan-order trees*** for the Query Execution Plan and all the Alternative Query Plans. The following diagram shows what a join order tree looks like:

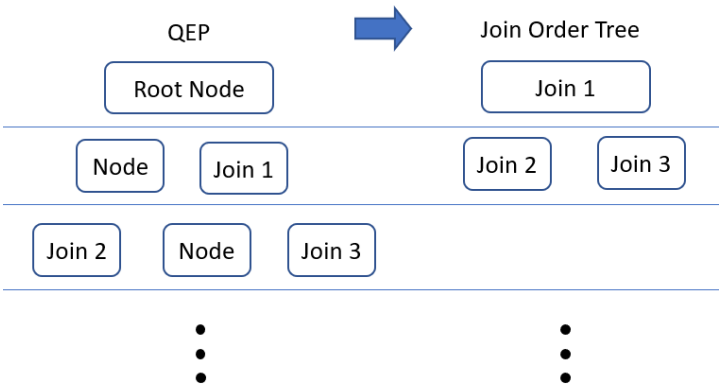


Figure 6: Join order tree for a QEP tree

The connotation of a scan order tree is similar and can be reproduced in a similar fashion. Our intuition behind using such a tree was that due to change in Algorithms, additional pre-processing nodes may be generated like Sort-Merge join requires an additional “Sort” node in the QEP. This may push a join/scan node a level up or down. In order to resolve this issue, we came up with a pre-processing step of creating these join-order trees.
These trees circumvent the problem as the additional processing nodes are not placed inside the Join Order tree/ Scan Order tree.

4.1.2 Overview of Simultaneous BFS

We traverse the nodes of the QEP and compare them to that of each AQP in the same level. To achieve this, we utilise the concept of a simultaneous Breadth First Search.

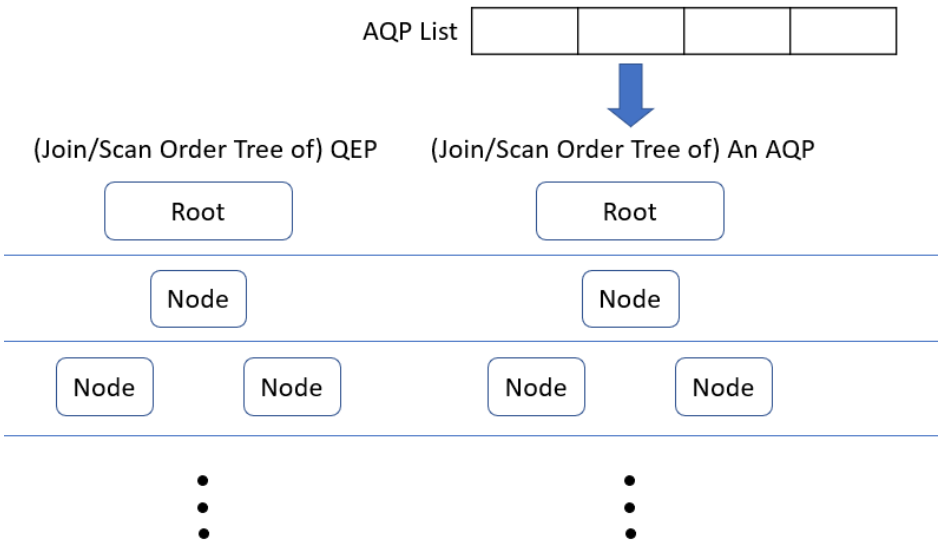


Figure 7: Simultaneous BFS

By utilising the root-node-childnode structure of our plans, we are able to compare each node in the QEP with the corresponding node in the same level of the AQP when traversing level by

level. We originally compared QEP nodes with AQP nodes directly. Realised that this later gave rise to an ordering issue which we will discuss in section 4.4.
Hence the simultaneous BFS is either only performed between all join-order trees and scan-order trees. The diagram above shows the solution we decided to use, using a tree of join/scan orders instead.
While comparing an AQP node to a QEP Node, we ***match the join condition or scan condition strings*** of the 2 nodes using ***"thefuzz"*** library (open-source). If the match score is greater than a certain ***"threshold" (in our case 95%)***, we match the 2 nodes and generate the justification using the method described in the section below.

4.1.3 Generating Justifications

Amongst the objects that each node contains, the total cost is calculated in the total_cost object. In order to justify the choice of a join/scan method in the QEP over another, we need to first perform matching of this particular node with its corresponding node in an AQP. This is achieved in section 4.2. Now, we compare the costs. The unit we use is factor - calculated by

$$\left\lceil \frac{Total\ cost\ of\ AQP\ node}{Total\ cost\ of\ QEP\ node} \right\rceil$$

Note: To calculate the total cost of a QEP or AQP Node, we need to perform 2 steps:

1. Subtract the total cost of the parent (provided by QEP) with total cost of all its children
2. Multiply the new total cost with number of loops ("Actual Loops" key in QEP)

This represents how much more the cost of executing the AQP node will be, as a ratio. Here we use the ceiling function instead of a round function. This way, as long as any factor is above 1, we can conclude that the join/scan method chosen by the QEP has a lower cost than that of the AQP. Conversely, any factor less than 1 means otherwise.

A point to note is that AQP nodes may not always have a higher cost than the QEP. This is because the database system does not calculate the exact costs of all exhaustive possible plans. Instead, it makes estimates and chooses a decent plan - it may not be the best plan. Therefore, in our calculation, it would be prudent to expect that the factor for some comparisons may be less than 1, meaning that the AQP node method is better than that of the QEP node. Since we use the ceiling function, we can take any value that is not larger than 1 to mean that.

We use this factor to justify the choice of join/scan method, and write it to the particular node's justification object for later use in the graphical UI.

4.1.4 Changed Join Order in AQPs

An issue encountered is the different order of joins/scans generated by the AQP in regards to the QEP. If we generate a plan with only one join method enabled, the order of the joins may be different from the order in the QEP.
2 Nodes with the ***same join condition*** may not be ***semantically equal*** because they may be

1. Applied on different intermediate results or
2. They exist on a different level in the join order tree. ***(Level Rule)***

Hence ***if we just compare nodes based on the join condition***, this may lead to ***incorrect*** justification. This is because the total cost of a join in AQP may be lower as it may have been performed with a different ordering. As shown in the figure below, *Join 1 and Join 2 are semantically unequal.*

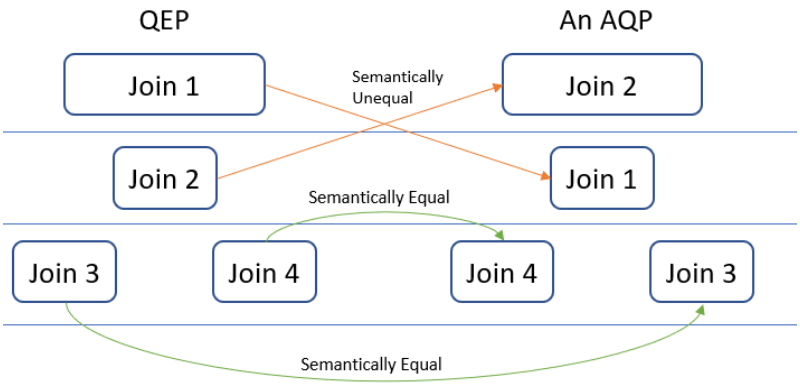


Figure 8: Join ordering problem when comparing nodes in AQP

In order to solve this issue, we generate join order trees and scan order trees separately. We then pass them as a list of join/scan nodes into a function to match, calculate and write the justification. This is a partial resolution as while comparing we ensure that the ***Level Rule*** is not violated. However, a complete resolution involves comparing the intermediate results recursively as well. However, this would eliminate a lot of possible matches between QEP and AQP. Thus, we will have to retrieve a lot more than 9 AQPs.

4.2 QEP and Query Matcher

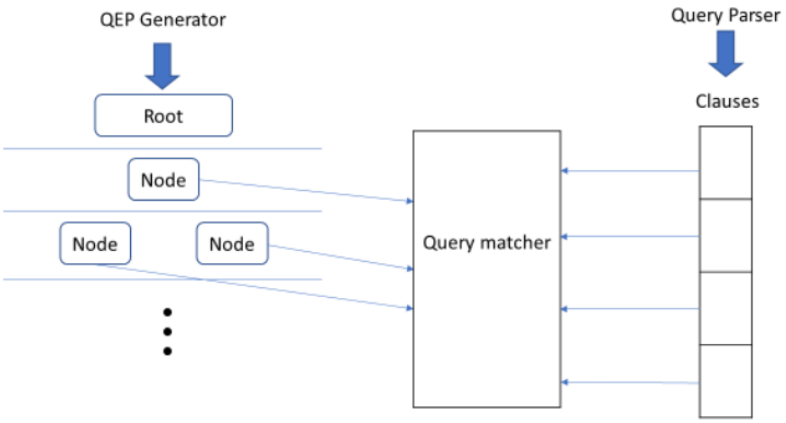


Figure 9: Matching between QEP Nodes and query clauses

Once the justification and annotation for different QEP nodes have been added as metadata in the node, this routine proceeds to match clauses extracted from SQL parser against nodes in QEP. For every clause in the query, the module extracts the best match possible from the Query

Execution Plan using string matching algorithms on the join and scan conditions. The library we use for string matching is ***"thefuzz"***(open-source). Now the generated matching map is sent to the ***Interface*** **displaying to the user.**

Note: A node in the QEP can be matched to different clauses but not the other way around. A clause will only have a single node as a match. The reason for this many-one mapping is the existence of multiple conditions within the same node. For example there could be an Index based NL Join with a join condition and a filter condition. The clauses for these conditions should be mapped to the same Index based NL join node.

5 User Interface

We designed a user friendly GUI which allows users to enter any intended SQL query as text and produce an annotated query.

5.1 GUI framework used

To facilitate the user input, we used the tkinter framework, which is built into the python standard library. We used it because it is cross platform and makes use of native system elements, ensuring a consistent user experience.

5.2 Guide to using interface

Below is the interface produced upon running our project file.

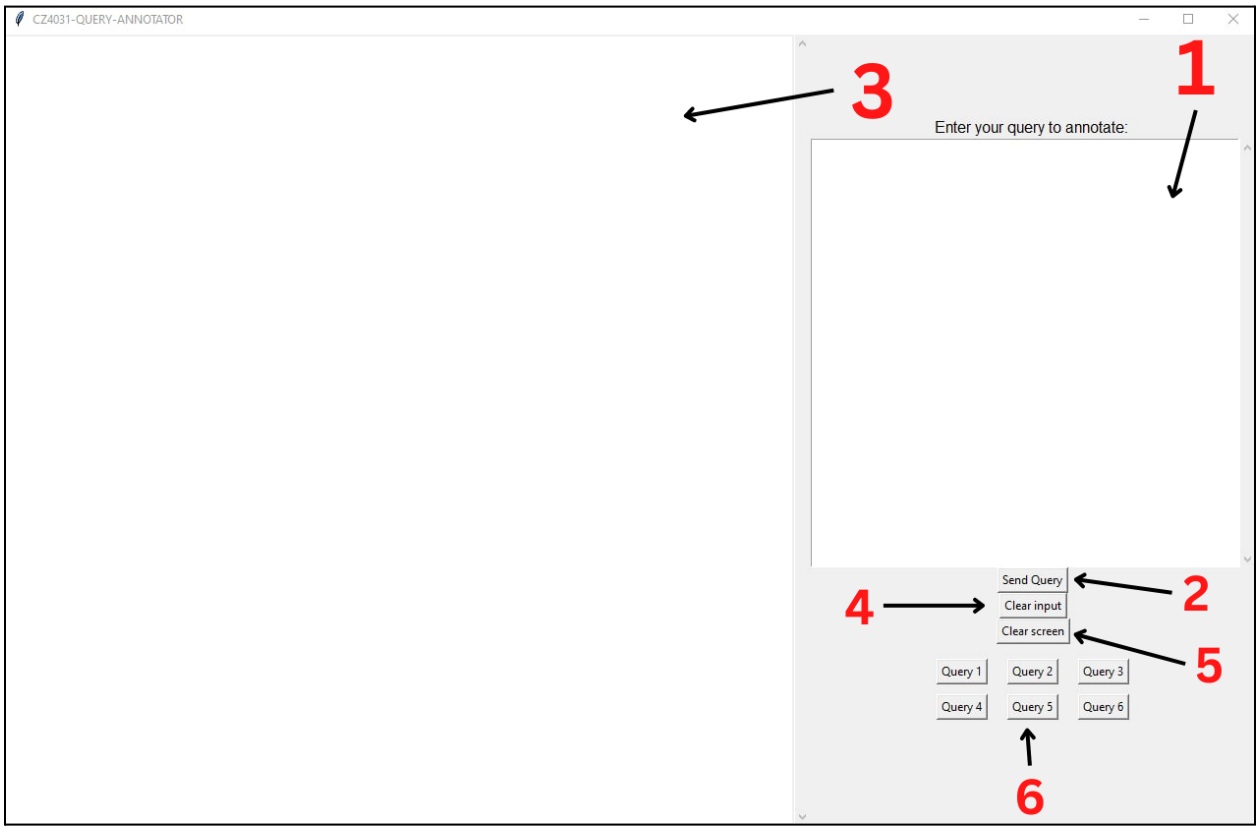


Figure 10: Our application's GUI

The components are as follows (numbered accordingly):

1. Textfield: User enters the intended query here
2. “Send query” button: User clicks on this button to submit the query
3. Annotation screen: Annotated SQL query will appear here
4. “Clear input” button: User clicks this button to clear the textbox
5. “Clear screen” button: User clicks this button to clear the canvas
6. Query buttons 1-6: User clicks buttons to load preloaded queries into the text field

5.3 Annotation process

Upon pressing submit, the query is sent to our main programme. The full pipeline of our main programme has been explained earlier. Overall, the process will produce two outputs:

1. **query** (string): String containing cleaned and properly formatted query
2. **annotationList** (dict<String to be annotated, Annotation(s)>): Dictionary containing the annotations to be added to the query
    a. Values of the key value pairs are a single string if there is only one of that key to be annotated, or a list should there be multiple annotations for the same string to be annotated.
    b. E.g if query = “A B C A B C”, and there are two annotations for B and one annotation for C, then annotationList = {“B”:[“B1”,”B2”],”C”:”C1”}

The annotated query is created with the help of turtle, another pre-installed python library that enables users to create pictures and shapes on a virtual canvas. Our program uses the above as inputs to produce the annotated query. The general process is as follows:

1. Iterate through the keys of annotationList and identify their indexes within query
2. Split the query into words, using the identified indexes to determine which words will have annotations pointing to them
3. Iteratively print the words with turtle using the .write() method, and draw guiding lines by turning and moving the turtle, and switching between .penup() and pendown() appropriately
    a. Turtle goes down a new line when:
        i. when the turtle reaches the section for the annotation on the right while writing the query
        ii. when the turtle reaches the right edge of the annotation screen while writing an annotation
        iii. if the next word to be printed is a query keyword (i.e SELECT, FROM, WHERE)
    b. Yellow highlight added for words with annotations (Yellow box drawn and filled)
    c. All annotations are printed in red on the right side of the canvas
    d. Pointers are drawn in blue from the annotations to the relevant words
    e. Query is printed and drawn live on the annotation screen, turtle set to maximum speed

In the case of many or long annotations, a scroll bar will appear at the side of the canvas to allow the user to view all of them. Processing of the query can take quite a while, the user may

have to wait for 30-60 seconds from when the submit button is pressed before the turtle starts drawing (Window may show as “Not responding” while processing).

6 Example Results

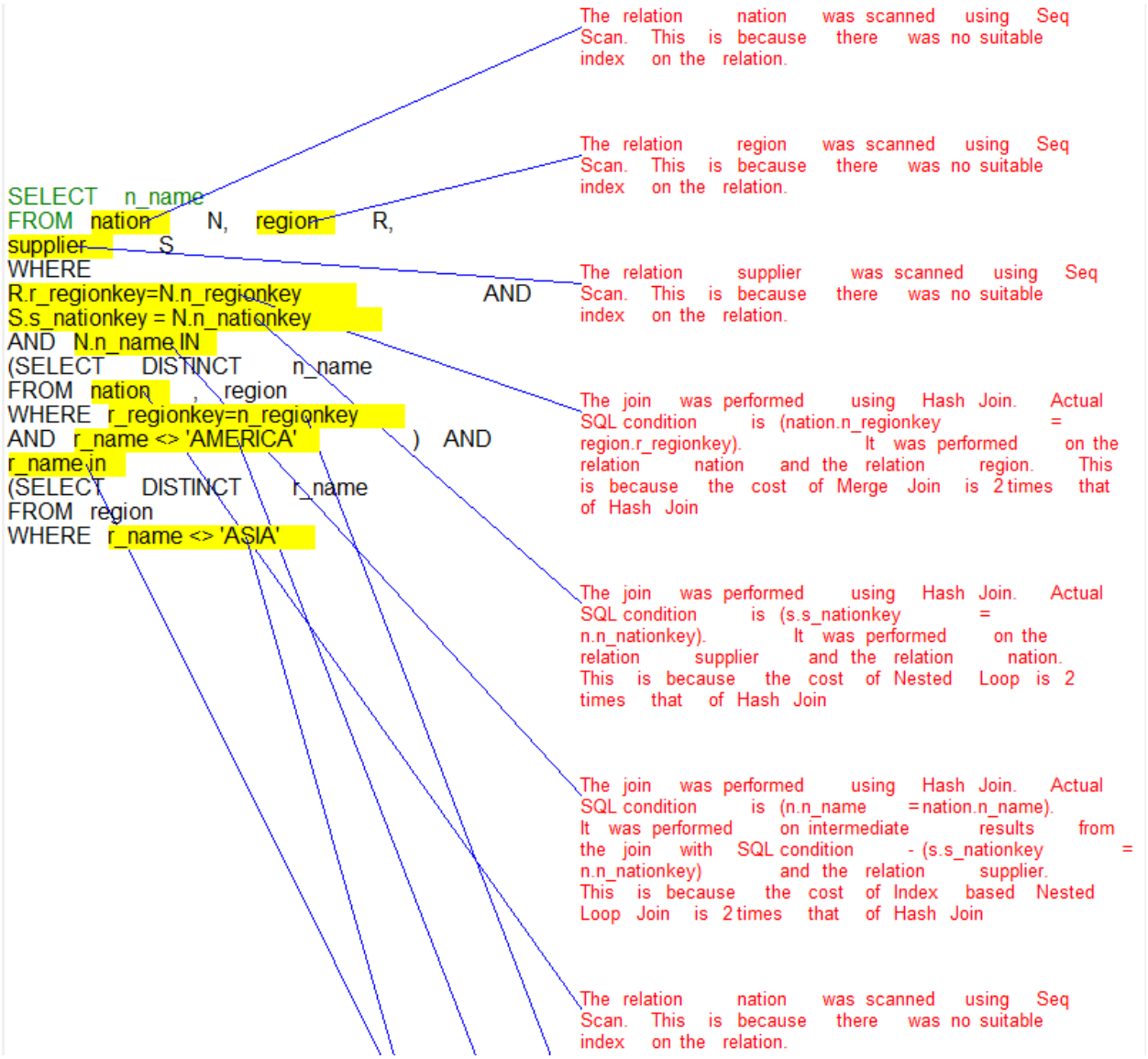
The following 6 queries were run through our program and the resulting annotations are displayed respectively. For the sake of proper viewing, the images for the results are cut off. To view the full image for the result, refer to Appendix A, or you can also try out the program with the respective query buttons implemented.

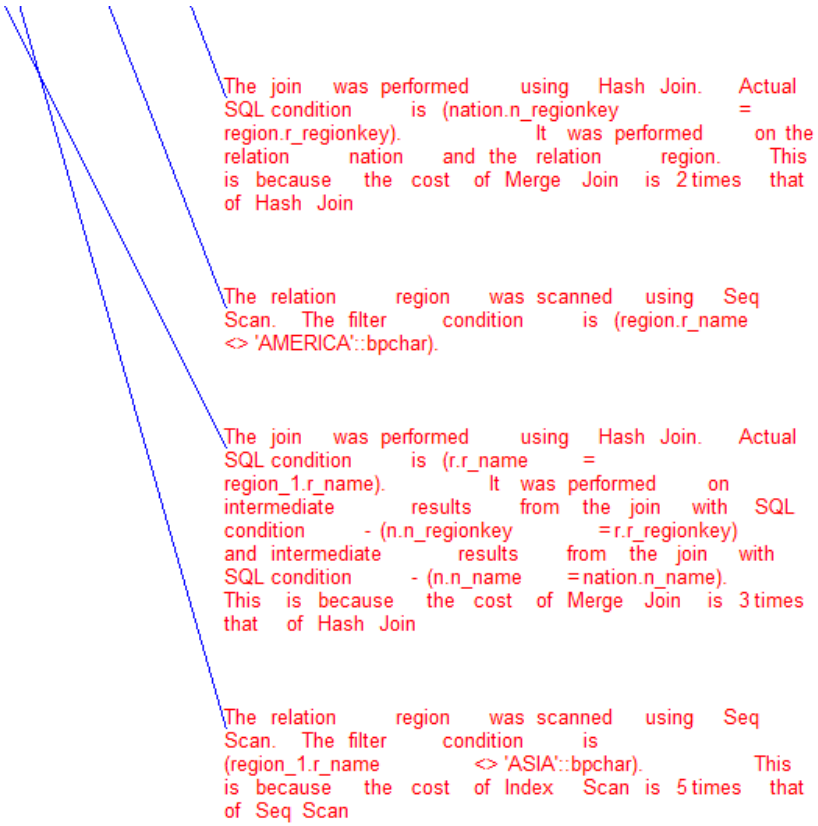
6.1 Query 1

Query

```
SELECT n_name
FROM nation N, region R,supplier S
WHERE R.r_regionkey=N.n_regionkey AND S.s_nationkey = N.n_nationkey AND
N.n_name IN
(SELECT DISTINCT n_name FROM nation,region WHERE r_regionkey=n_regionkey
AND r_name <> 'AMERICA') AND
r_name in (SELECT DISTINCT r_name from region where r_name <> 'ASIA');
```

Results


SELECT n_name
FROM nation N, region R,
supplier S
WHERE
R.r_regionkey=N.n_regionkey AND
S.s_nationkey = N.n_nationkey
AND N.n_name IN
(SELECT DISTINCT n_name
FROM nation , region
WHERE r_regionkey=n_regionkey
AND r_name <> 'AMERICA') AND
r_name in
(SELECT DISTINCT r_name
FROM region
WHERE r_name <> 'ASIA'
The relation nation was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation region was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation supplier was scanned using Seq Scan. This is because there was no suitable index on the relation.
The join was performed using Hash Join. Actual SQL condition is (nation.n_regionkey = region.r_regionkey). It was performed on the relation nation and the relation region. This is because the cost of Merge Join is 2 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (s.s_nationkey = n.n_nationkey). It was performed on the relation supplier and the relation nation. This is because the cost of Nested Loop is 2 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (n.n_name = nation.n_name). It was performed on intermediate results from the join with SQL condition - (s.s_nationkey = n.n_nationkey) and the relation supplier. This is because the cost of Index based Nested Loop Join is 2 times that of Hash Join
The relation nation was scanned using Seq Scan. This is because there was no suitable index on the relation.

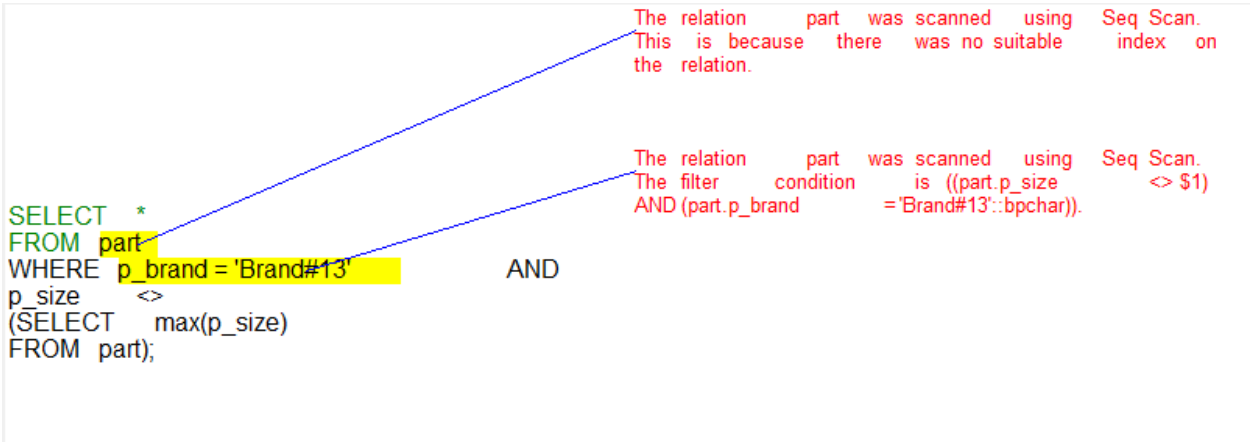

The join was performed using Hash Join. Actual SQL condition is (nation.n_regionkey = region.r_regionkey). It was performed on the relation nation and the relation region. This is because the cost of Merge Join is 2 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region.r_name <> 'AMERICA'::bpchar).
The join was performed using Hash Join. Actual SQL condition is (r.r_name = region_1.r_name). It was performed on intermediate results from the join with SQL condition - (n.n_regionkey = r.r_regionkey) and intermediate results from the join with SQL condition - (n.n_name = nation.n_name). This is because the cost of Merge Join is 3 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region_1.r_name <> 'ASIA'::bpchar). This is because the cost of Index Scan is 5 times that of Seq Scan

6.2 Query 2

Query

```
SELECT *
FROM part
WHERE p_brand = 'Brand#13' AND p_size <> (SELECT max(p_size) FROM part);
```

Results

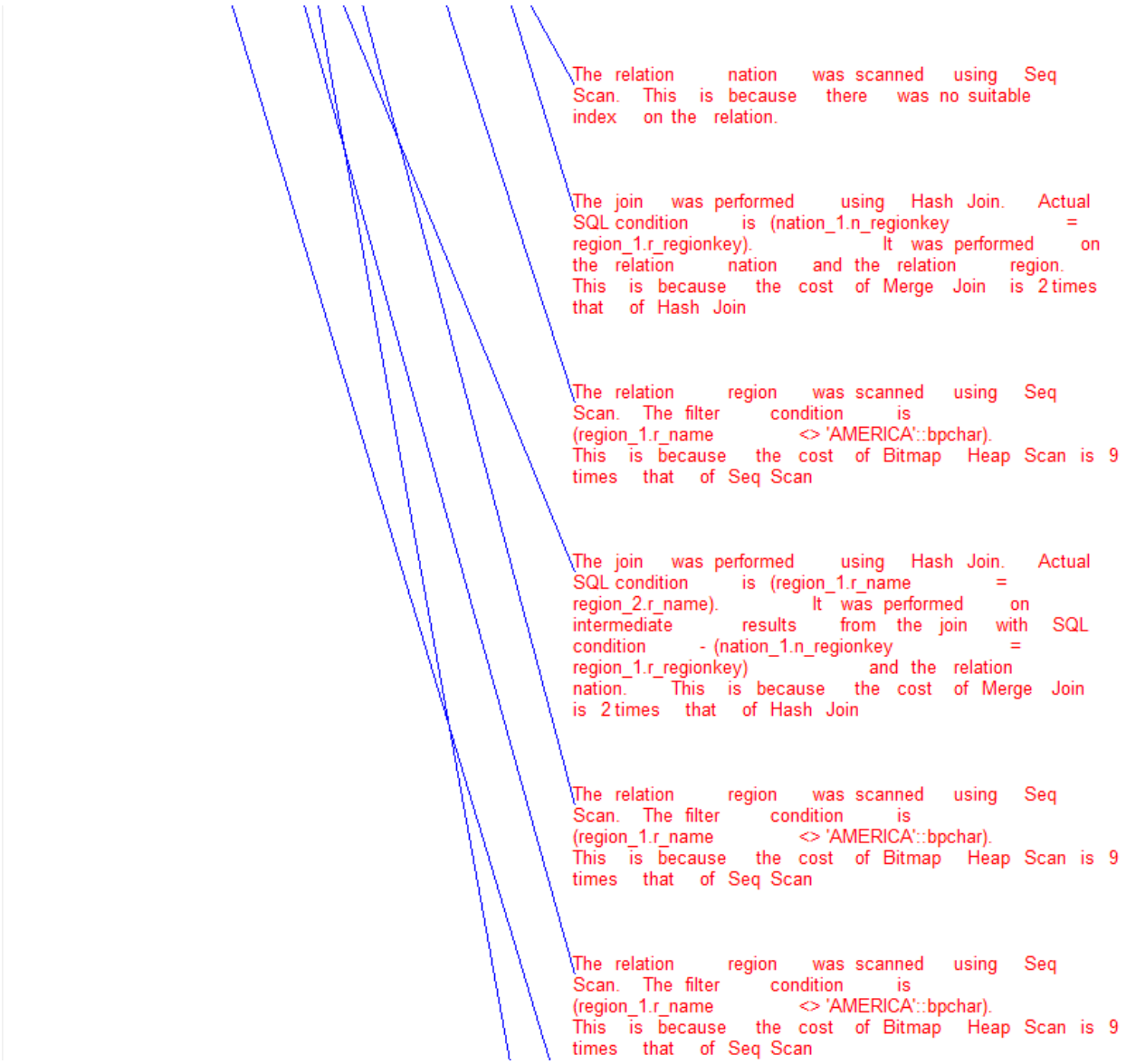


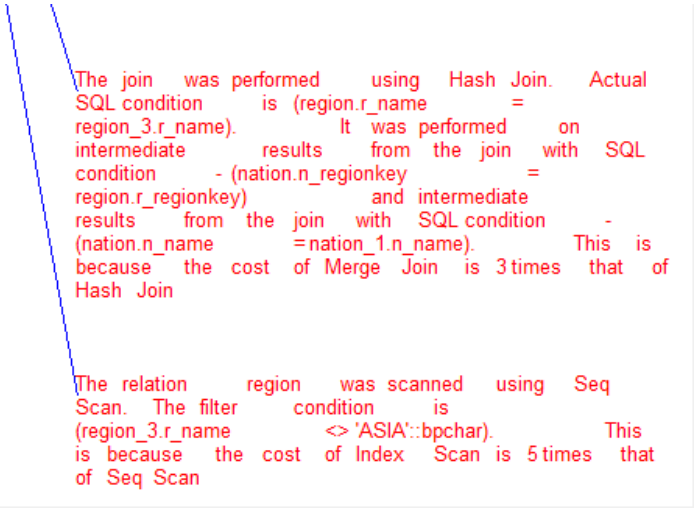
6.3 Query 3

Query

```
SELECT n_name
FROM nation, region,supplier
WHERE r_regionkey=n_regionkey AND s_nationkey = n_nationkey AND n_name IN
(SELECT DISTINCT n_name FROM nation,region WHERE r_regionkey=n_regionkey
AND r_name <> 'AMERICA' AND
r_name in (SELECT DISTINCT r_name from region where r_name <> 'LATIN
AMERICA' AND r_name <> 'AFRICA')) AND
r_name in (SELECT DISTINCT r_name from region where r_name <> 'ASIA');
```

Results


The relation nation was scanned using Seq Scan. This is because there was no suitable index on the relation.
The join was performed using Hash Join. Actual SQL condition is (nation_1.n_regionkey = region_1.r_regionkey). It was performed on the relation nation and the relation region. This is because the cost of Merge Join is 2 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region_1.r_name <> 'AMERICA'::bpchar). This is because the cost of Bitmap Heap Scan is 9 times that of Seq Scan
The join was performed using Hash Join. Actual SQL condition is (region_1.r_name = region_2.r_name). It was performed on intermediate results from the join with SQL condition - (nation_1.n_regionkey = region_1.r_regionkey) and the relation nation. This is because the cost of Merge Join is 2 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region_1.r_name <> 'AMERICA'::bpchar). This is because the cost of Bitmap Heap Scan is 9 times that of Seq Scan
The relation region was scanned using Seq Scan. The filter condition is (region_1.r_name <> 'AMERICA'::bpchar). This is because the cost of Bitmap Heap Scan is 9 times that of Seq Scan

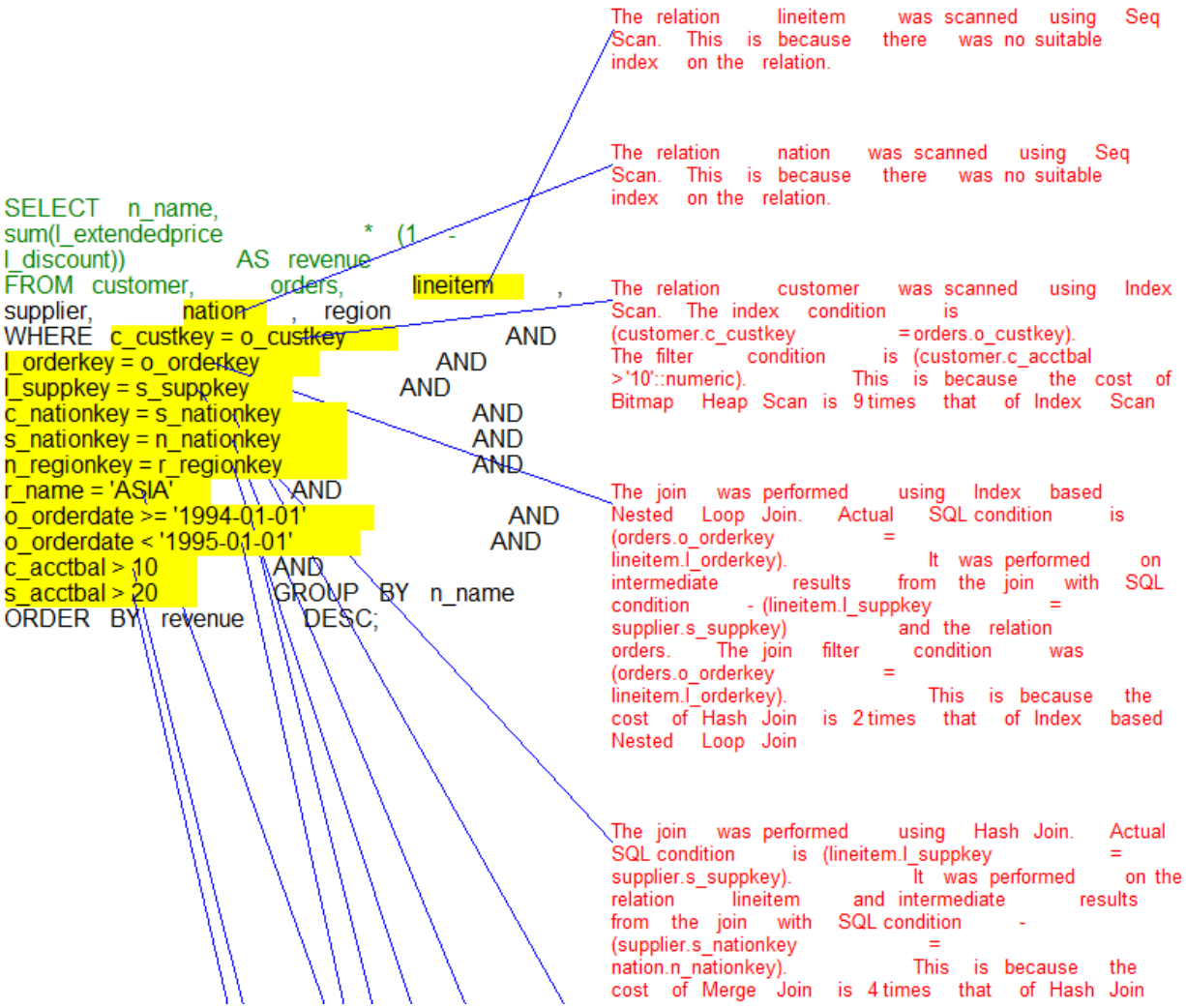

The join was performed using Hash Join. Actual SQL condition is (region.r_name = region_3.r_name). It was performed on intermediate results from the join with SQL condition - (nation.n_regionkey = region.r_regionkey) and intermediate results from the join with SQL condition - (nation.n_name = nation_1.n_name). This is because the cost of Merge Join is 3 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region_3.r_name <> 'ASIA'::bpchar). This is because the cost of Index Scan is 5 times that of Seq Scan

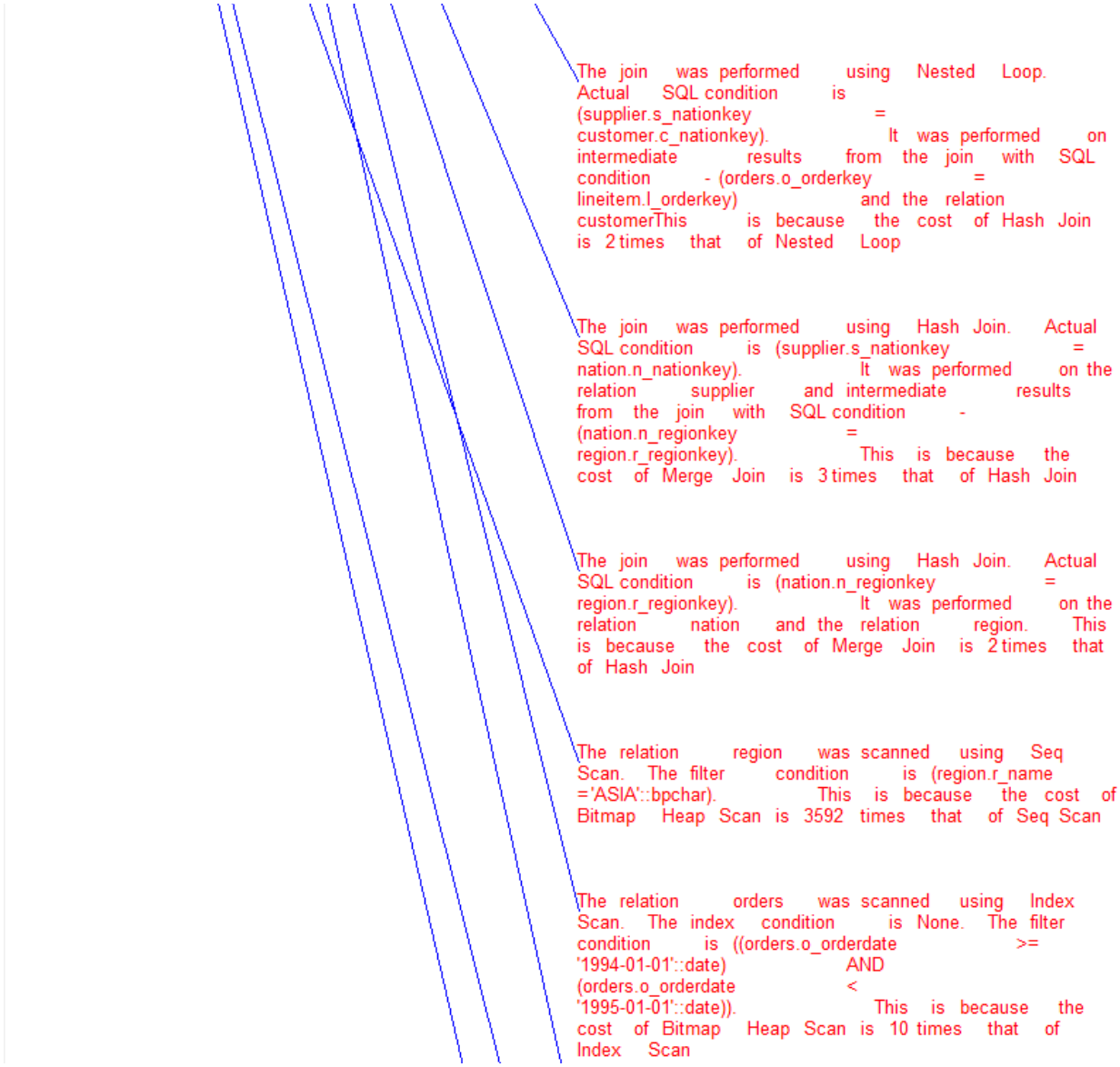
6.4 Query 4

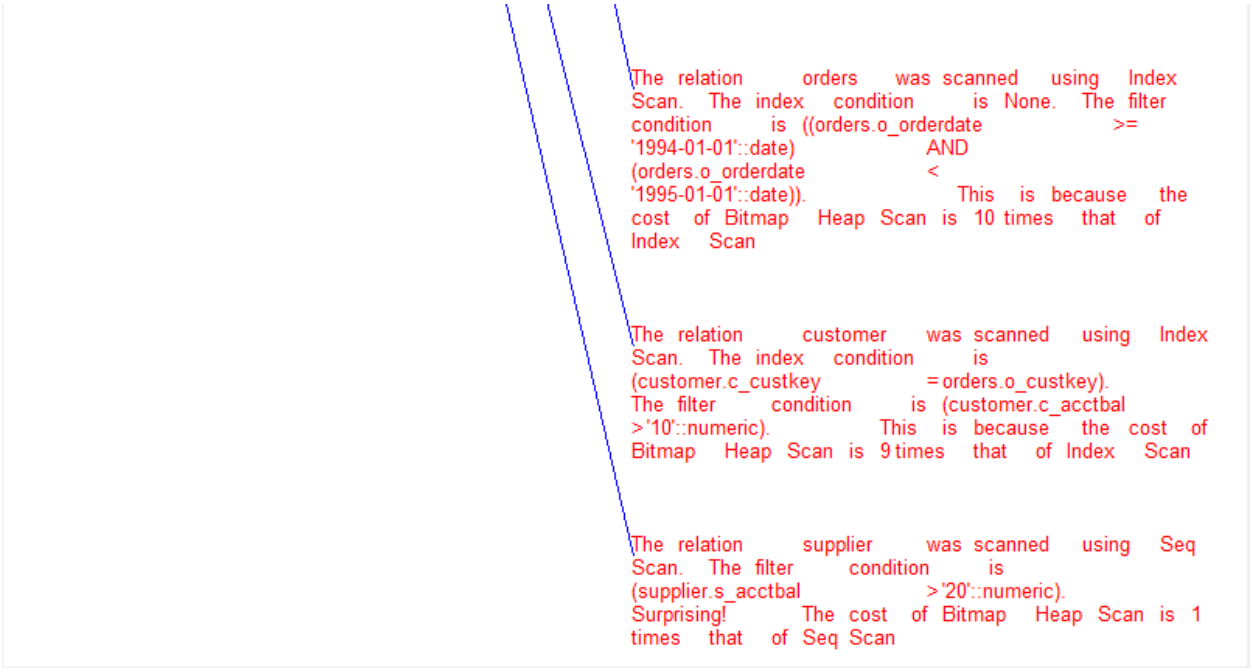
Query

```
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;
```

Results


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;
The relation lineitem was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation nation was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation customer was scanned using Index Scan. The index condition is (customer.c_custkey = orders.o_custkey). The filter condition is (customer.c_acctbal >'10'::numeric). This is because the cost of Bitmap Heap Scan is 9 times that of Index Scan
The join was performed using Index based Nested Loop Join. Actual SQL condition is (orders.o_orderkey = lineitem.l_orderkey). It was performed on intermediate results from the join with SQL condition - (lineitem.l_suppkey = supplier.s_suppkey) and the relation orders. The join filter condition was (orders.o_orderkey = lineitem.l_orderkey). This is because the cost of Hash Join is 2 times that of Index based Nested Loop Join
The join was performed using Hash Join. Actual SQL condition is (lineitem.l_suppkey = supplier.s_suppkey). It was performed on the relation lineitem and intermediate results from the join with SQL condition - (supplier.s_nationkey = nation.n_nationkey). This is because the cost of Merge Join is 4 times that of Hash Join


The join was performed using Nested Loop. Actual SQL condition is (supplier.s_nationkey = customer.c_nationkey). It was performed on intermediate results from the join with SQL condition - (orders.o_orderkey = lineitem.l_orderkey) and the relation customerThis is because the cost of Hash Join is 2 times that of Nested Loop
The join was performed using Hash Join. Actual SQL condition is (supplier.s_nationkey = nation.n_nationkey). It was performed on the relation supplier and intermediate results from the join with SQL condition - (nation.n_regionkey = region.r_regionkey). This is because the cost of Merge Join is 3 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (nation.n_regionkey = region.r_regionkey). It was performed on the relation nation and the relation region. This is because the cost of Merge Join is 2 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region.r_name ='ASIA'::bpchar). This is because the cost of Bitmap Heap Scan is 3592 times that of Seq Scan
The relation orders was scanned using Index Scan. The index condition is None. The filter condition is ((orders.o_orderdate >= '1994-01-01'::date) AND (orders.o_orderdate < '1995-01-01'::date)). This is because the cost of Bitmap Heap Scan is 10 times that of Index Scan

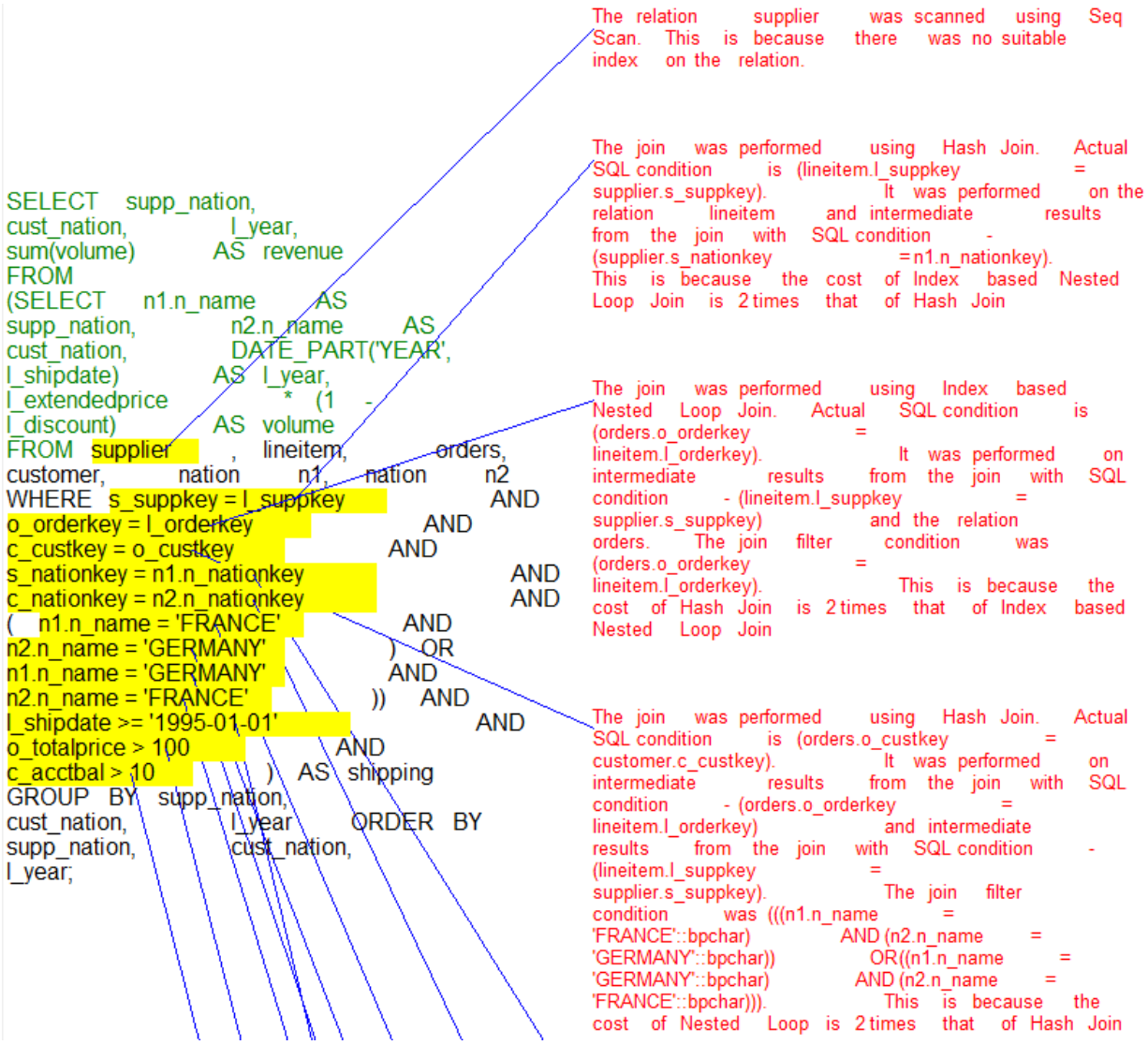

The relation orders was scanned using Index Scan. The index condition is None. The filter condition is ((orders.o_orderdate >= '1994-01-01'::date) AND (orders.o_orderdate < '1995-01-01'::date)). This is because the cost of Bitmap Heap Scan is 10 times that of Index Scan
The relation customer was scanned using Index Scan. The index condition is (customer.c_custkey = orders.o_custkey). The filter condition is (customer.c_acctbal > '10'::numeric). This is because the cost of Bitmap Heap Scan is 9 times that of Index Scan
The relation supplier was scanned using Seq Scan. The filter condition is (supplier.s_acctbal > '20'::numeric). Surprising! The cost of Bitmap Heap Scan is 1 times that of Seq Scan

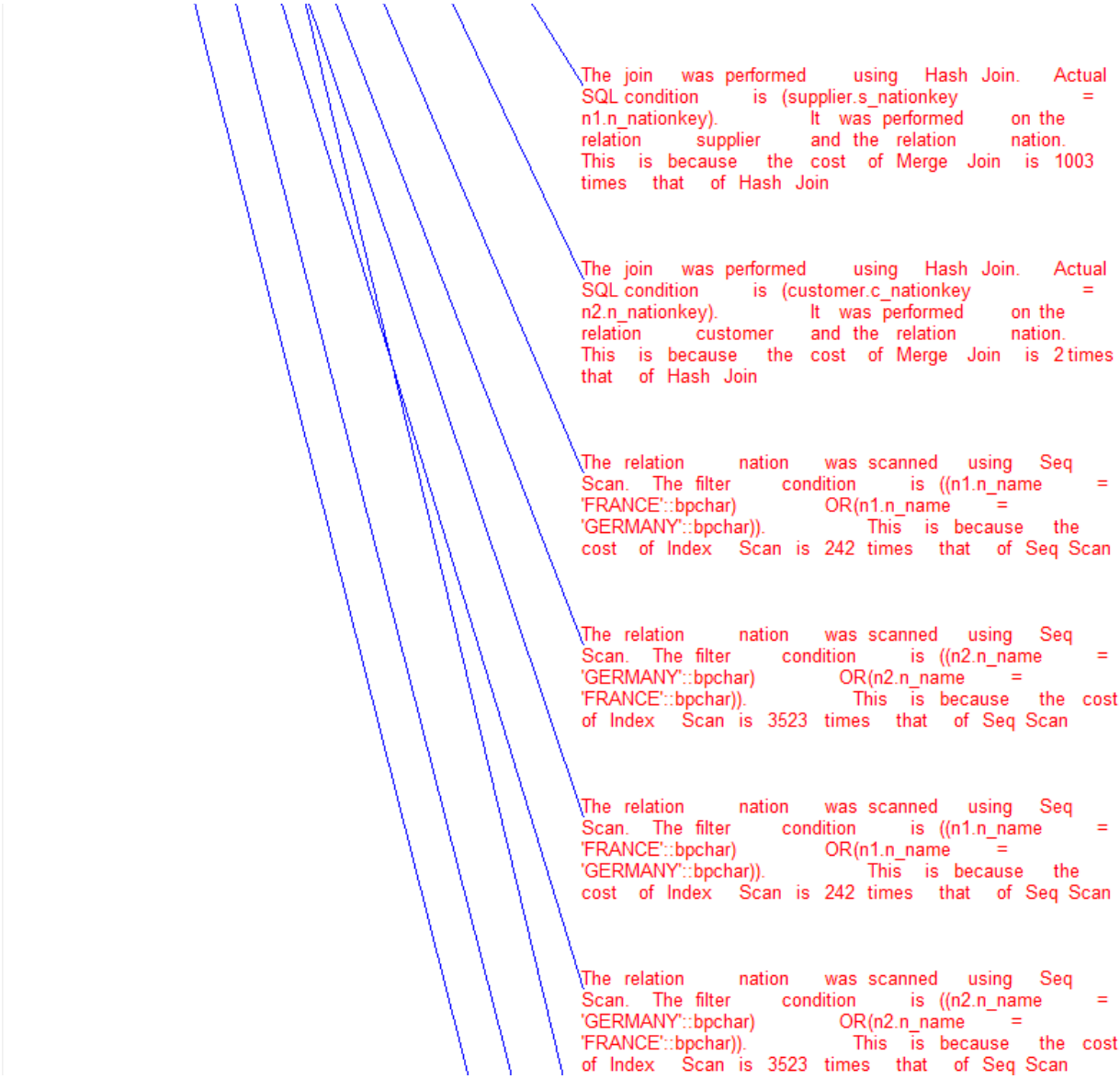
6.5 Query 5

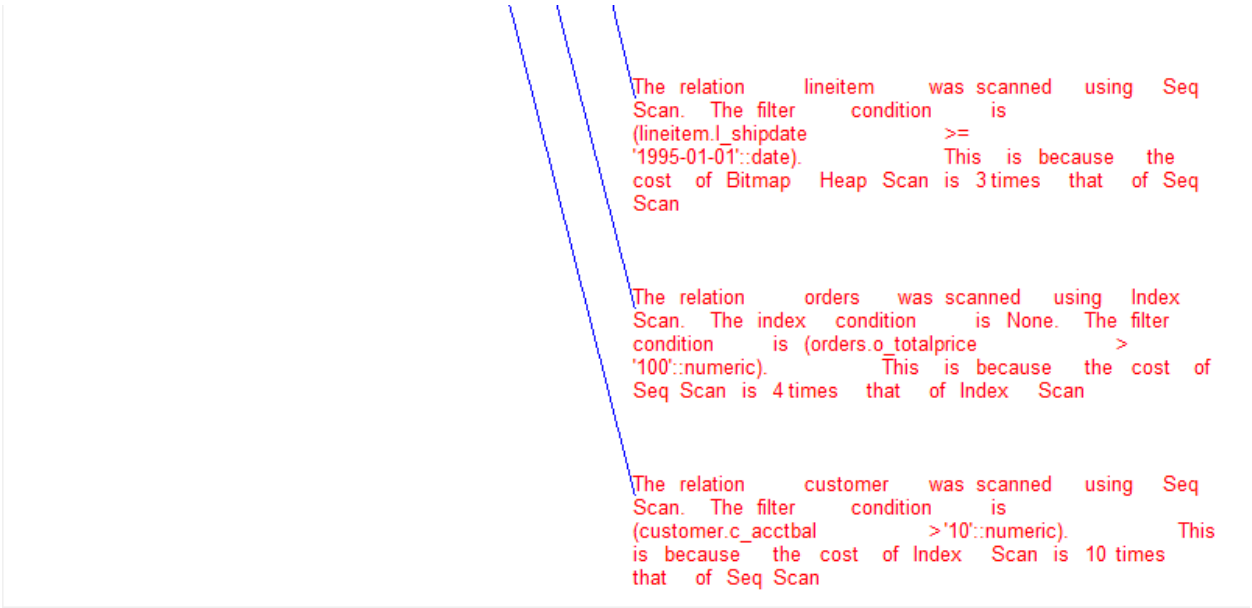
Query

```
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 >= '1995-01-01'
          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;
```

Results


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 >= '1995-01-01' 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;
The relation supplier was scanned using Seq Scan. This is because there was no suitable index on the relation.
The join was performed using Hash Join. Actual SQL condition is (lineitem.l_suppkey = supplier.s_suppkey). It was performed on the relation lineitem and intermediate results from the join with SQL condition - (supplier.s_nationkey = n1.n_nationkey). This is because the cost of Index based Nested Loop Join is 2 times that of Hash Join
The join was performed using Index based Nested Loop Join. Actual SQL condition is (orders.o_orderkey = lineitem.l_orderkey). It was performed on intermediate results from the join with SQL condition - (lineitem.l_suppkey = supplier.s_suppkey) and the relation orders. The join filter condition was (orders.o_orderkey = lineitem.l_orderkey). This is because the cost of Hash Join is 2 times that of Index based Nested Loop Join
The join was performed using Hash Join. Actual SQL condition is (orders.o_custkey = customer.c_custkey). It was performed on intermediate results from the join with SQL condition - (orders.o_orderkey = lineitem.l_orderkey) and intermediate results from the join with SQL condition - (lineitem.l_suppkey = supplier.s_suppkey). The join filter condition was (((n1.n_name = 'FRANCE'::bpchar) AND (n2.n_name = 'GERMANY'::bpchar)) OR((n1.n_name = 'GERMANY'::bpchar) AND (n2.n_name = 'FRANCE'::bpchar))). This is because the cost of Nested Loop is 2 times that of Hash Join


The join was performed using Hash Join. Actual SQL condition is (supplier.s_nationkey = n1.n_nationkey). It was performed on the relation supplier and the relation nation. This is because the cost of Merge Join is 1003 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (customer.c_nationkey = n2.n_nationkey). It was performed on the relation customer and the relation nation. This is because the cost of Merge Join is 2 times that of Hash Join
The relation nation was scanned using Seq Scan. The filter condition is ((n1.n_name = 'FRANCE'::bpchar) OR(n1.n_name = 'GERMANY'::bpchar)). This is because the cost of Index Scan is 242 times that of Seq Scan
The relation nation was scanned using Seq Scan. The filter condition is ((n2.n_name = 'GERMANY'::bpchar) OR(n2.n_name = 'FRANCE'::bpchar)). This is because the cost of Index Scan is 3523 times that of Seq Scan
The relation nation was scanned using Seq Scan. The filter condition is ((n1.n_name = 'FRANCE'::bpchar) OR(n1.n_name = 'GERMANY'::bpchar)). This is because the cost of Index Scan is 242 times that of Seq Scan
The relation nation was scanned using Seq Scan. The filter condition is ((n2.n_name = 'GERMANY'::bpchar) OR(n2.n_name = 'FRANCE'::bpchar)). This is because the cost of Index Scan is 3523 times that of Seq Scan

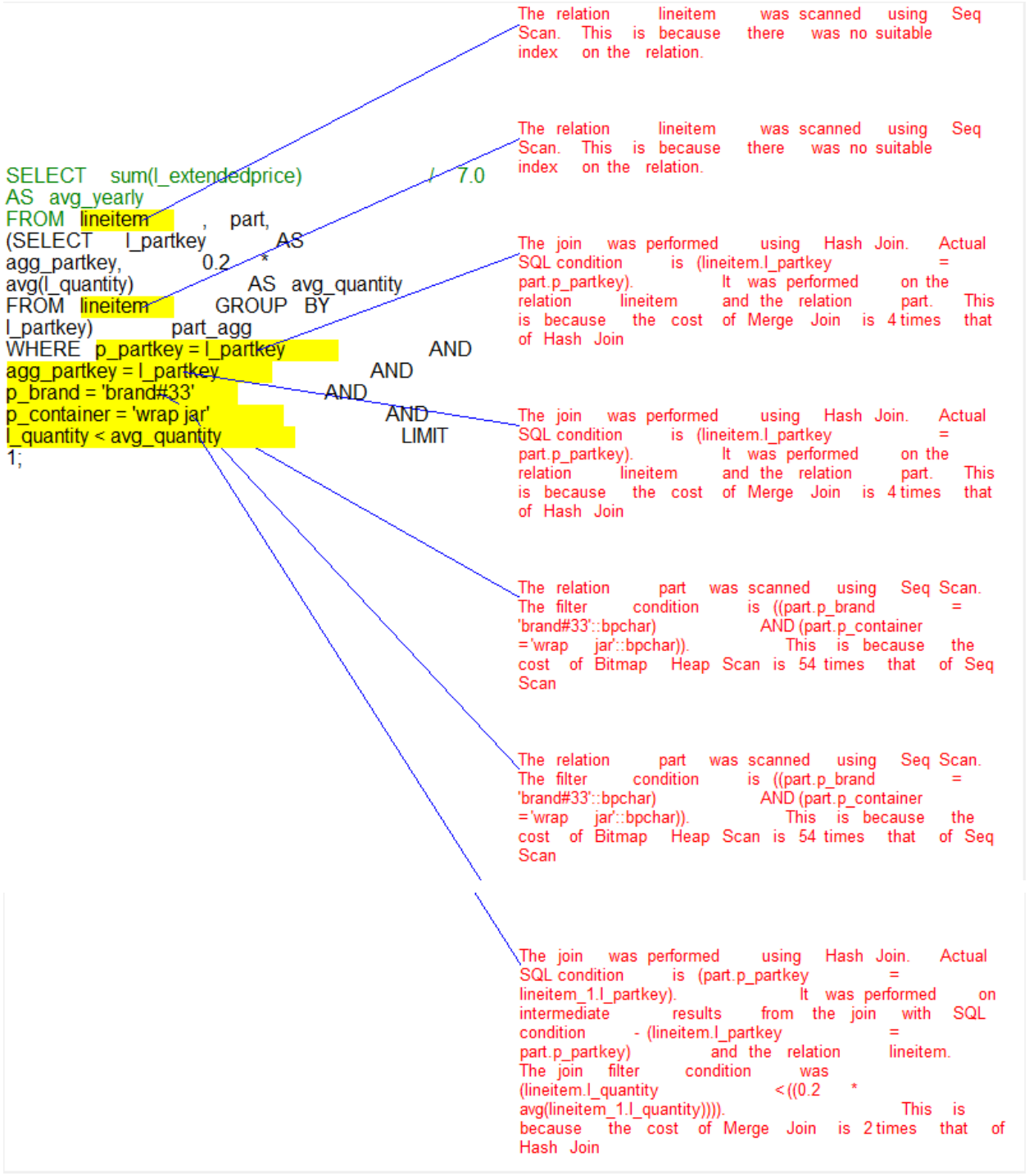


6.6 Query 6

Query

```
SELECT
      sum(l_extendedprice) / 7.0 AS avg_yearly
FROM
      lineitem,
      part,
        (SELECT l_partkey AS agg_partkey, 0.2 * avg(l_quantity) AS
avg_quantity FROM lineitem GROUP BY l_partkey) part_agg
WHERE
      p_partkey = l_partkey
        AND agg_partkey = l_partkey
      AND p_brand = 'brand#33'
      AND p_container = 'wrap jar'
      AND l_quantity < avg_quantity
limit 1;
```

Results


SELECT sum(l_extendedprice) / 7.0
AS avg_yearly
FROM lineitem , part,
(SELECT l_partkey AS
agg_partkey, 0.2 *
avg(l_quantity) AS avg_quantity
FROM lineitem GROUP BY
l_partkey) part_agg
WHERE p_partkey = l_partkey AND
agg_partkey = l_partkey AND
p_brand = 'brand#33' AND
p_container = 'wrap jar' AND
l_quantity < avg_quantity LIMIT
1;
The relation lineitem was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation lineitem was scanned using Seq Scan. This is because there was no suitable index on the relation.
The join was performed using Hash Join. Actual SQL condition is (lineitem.l_partkey = part.p_partkey). It was performed on the relation lineitem and the relation part. This is because the cost of Merge Join is 4 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (lineitem.l_partkey = part.p_partkey). It was performed on the relation lineitem and the relation part. This is because the cost of Merge Join is 4 times that of Hash Join
The relation part was scanned using Seq Scan. The filter condition is ((part.p_brand = 'brand#33'::bpchar) AND (part.p_container = 'wrap jar'::bpchar)). This is because the cost of Bitmap Heap Scan is 54 times that of Seq Scan
The relation part was scanned using Seq Scan. The filter condition is ((part.p_brand = 'brand#33'::bpchar) AND (part.p_container = 'wrap jar'::bpchar)). This is because the cost of Bitmap Heap Scan is 54 times that of Seq Scan
The join was performed using Hash Join. Actual SQL condition is (part.p_partkey = lineitem_1.l_partkey). It was performed on intermediate results from the join with SQL condition - (lineitem.l_partkey = part.p_partkey) and the relation lineitem. The join filter condition was (lineitem.l_quantity < ((0.2 * avg(lineitem_1.l_quantity)))). This is because the cost of Merge Join is 2 times that of Hash Join

7 Limitations of our program

7.1 Unable to handle “between” keyword

The syntax for using the BETWEEN keyword is typically as such:

```
SELECT * FROM table
WHERE value BETWEEN low AND high
```

Our parser has been coded to process raw queries by splitting the raw query into tokens and then extracting table names via FROM keyword, and conditions via WHERE and HAVING keywords. Since BETWEEN requires processing for the table name prior to itself, a value after itself, an AND keyword after that value, and finally another value, our parser is not able to recognise and include BETWEEN conditions.

As an alternative, we can achieve the same outcome with the following query:

```
SELECT * FROM table
WHERE value > low
AND value < high
```

This provides a similar outcome and our parser will be able to handle this query.

7.2 Cannot handle intersect and union queries

Our program has not been coded to include intersect and union queries. Performing intersect and union changes intermediate tuples between levels. If we want to compare costs of performing these two data manipulation operators on different levels, we would have to generate many more AQPs. In order to keep the runtime from exceeding a practical amount of time, we have chosen not to include queries with such operators.

7.3 Uses string matching algorithms instead of string equality

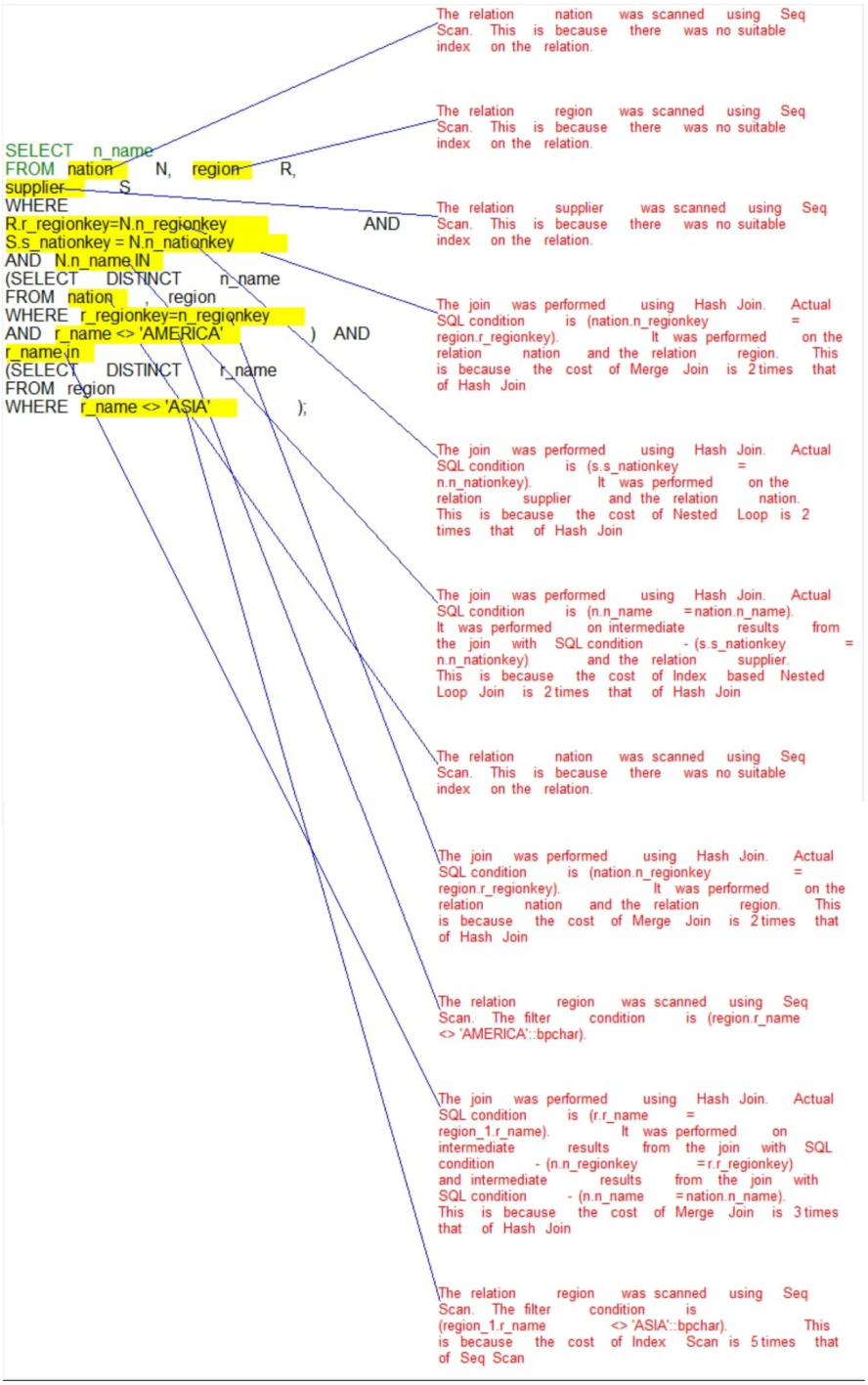
We use theFuzz, an open-source fuzzy string matching module, when comparing the join/scan conditions in nodes. Setting the threshold is a cautious line we have to toe - too tight and we may miss the comparison of some pairs of nodes; too lenient and we might compare nodes that should not be compared.

7.4 No justification due to lack of exhaustive search

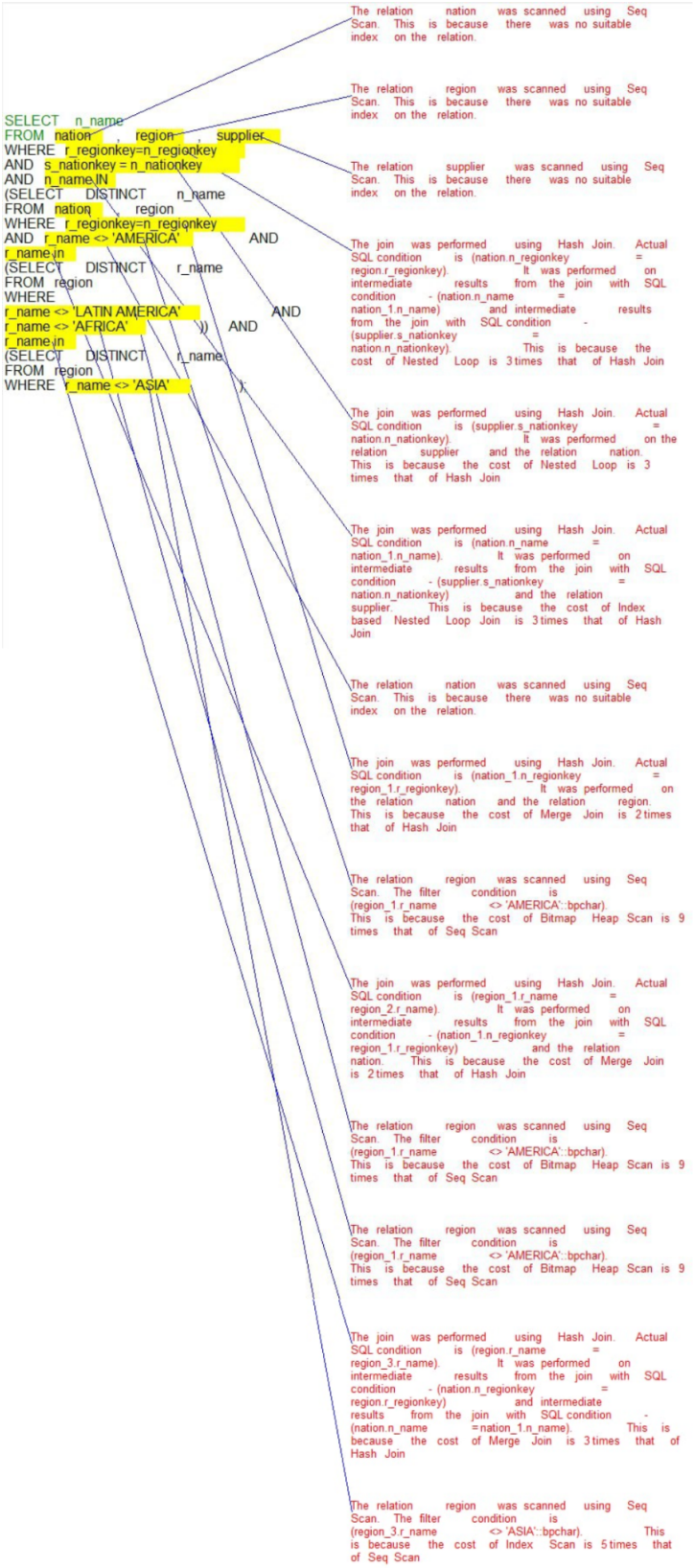
In some rare cases, some nodes may not have justification provided. This is because we do not perform an exhaustive search when generating AQPs. We only generate them based on each join/scan method. This means that some AQP nodes may be deemed as similar to QEP nodes - our program does not provide justification on nodes that have been used in QEP; only on those that have not been used in the QEP. Each AQPs also does not contain mixed joins on different levels. We do this for easier comparisons. As mentioned in section 4.1.4, we would have to generate much more than 8 AQPs if we want to do so. Therefore, we might not be able to find a different AQP node to compare with the QEP’s node and hence, no justification will be generated.

Appendix A

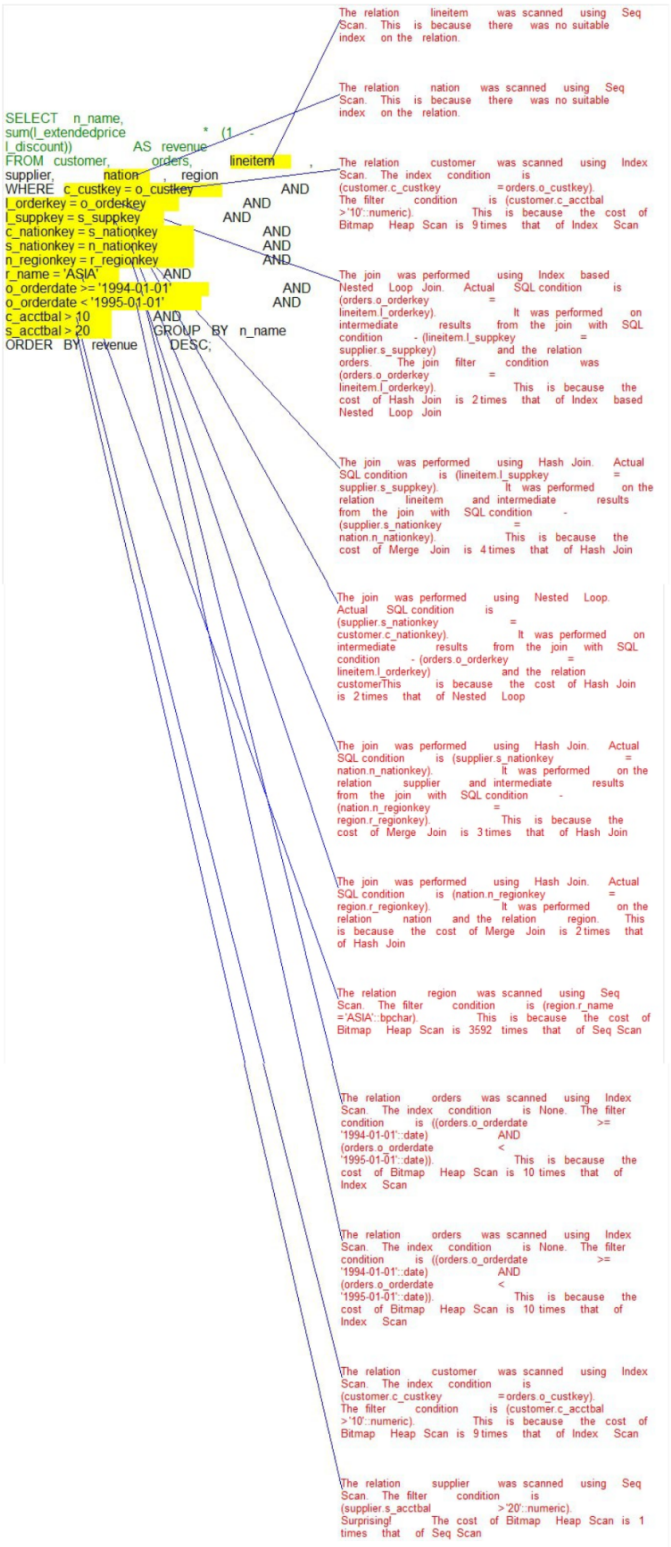
1. Result 1



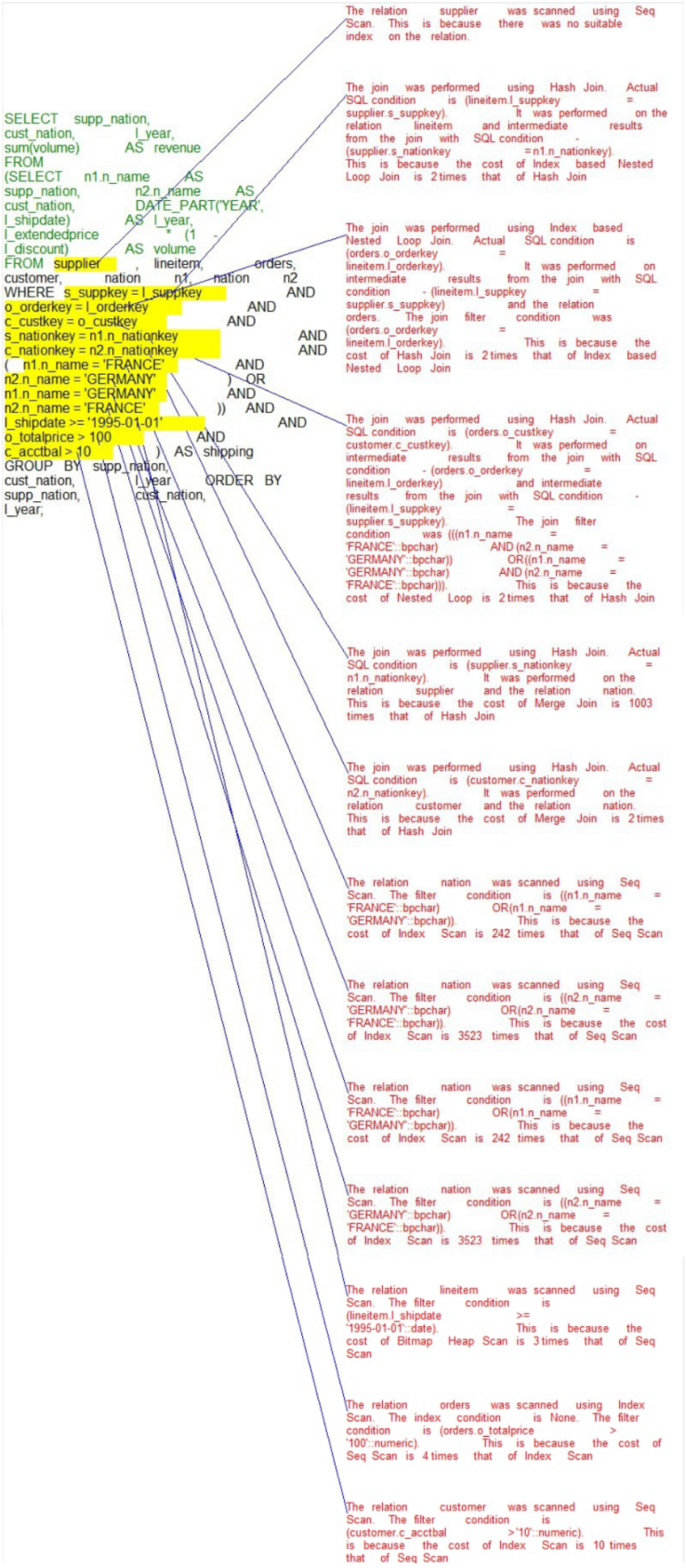
2. Result 3


SELECT n_name
FROM nation , region , supplier
WHERE r_regionkey=n_regionkey
AND s_nationkey = n_nationkey
AND n_name IN
(SELECT DISTINCT n_name
FROM nation , region
WHERE r_regionkey=n_regionkey
AND r_name <> 'AMERICA' AND
r_name in
(SELECT DISTINCT r_name
FROM region
WHERE
r_name <> 'LATIN AMERICA' AND
r_name <> 'AFRICA')) AND
r_name in
(SELECT DISTINCT r_name
FROM region
WHERE r_name <> 'ASIA');
The relation nation was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation region was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation supplier was scanned using Seq Scan. This is because there was no suitable index on the relation.
The join was performed using Hash Join. Actual SQL condition is (nation.n_regionkey = region.r_regionkey). It was performed on intermediate results from the join with SQL condition - (nation.n_name = nation_1.n_name) and intermediate results from the join with SQL condition - (supplier.s_nationkey = nation.n_nationkey). This is because the cost of Nested Loop is 3 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (supplier.s_nationkey = nation.n_nationkey). It was performed on the relation supplier and the relation nation. This is because the cost of Nested Loop is 3 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (nation.n_name = nation_1.n_name). It was performed on intermediate results from the join with SQL condition - (supplier.s_nationkey = nation.n_nationkey) and the relation supplier. This is because the cost of Index based Nested Loop Join is 3 times that of Hash Join
The relation nation was scanned using Seq Scan. This is because there was no suitable index on the relation.
The join was performed using Hash Join. Actual SQL condition is (nation_1.n_regionkey = region_1.r_regionkey). It was performed on the relation nation and the relation region. This is because the cost of Merge Join is 2 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region_1.r_name <> 'AMERICA'::bpchar). This is because the cost of Bitmap Heap Scan is 9 times that of Seq Scan
The join was performed using Hash Join. Actual SQL condition is (region_1.r_name = region_2.r_name). It was performed on intermediate results from the join with SQL condition - (nation_1.n_regionkey = region_1.r_regionkey) and the relation nation. This is because the cost of Merge Join is 2 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region_1.r_name <> 'AMERICA'::bpchar). This is because the cost of Bitmap Heap Scan is 9 times that of Seq Scan
The relation region was scanned using Seq Scan. The filter condition is (region_1.r_name <> 'AMERICA'::bpchar). This is because the cost of Bitmap Heap Scan is 9 times that of Seq Scan
The join was performed using Hash Join. Actual SQL condition is (region.r_name = region_3.r_name). It was performed on intermediate results from the join with SQL condition - (nation.n_regionkey = region.r_regionkey) and intermediate results from the join with SQL condition - (nation.n_name = nation_1.n_name). This is because the cost of Merge Join is 3 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region_3.r_name <> 'ASIA'::bpchar). This is because the cost of Index Scan is 5 times that of Seq Scan

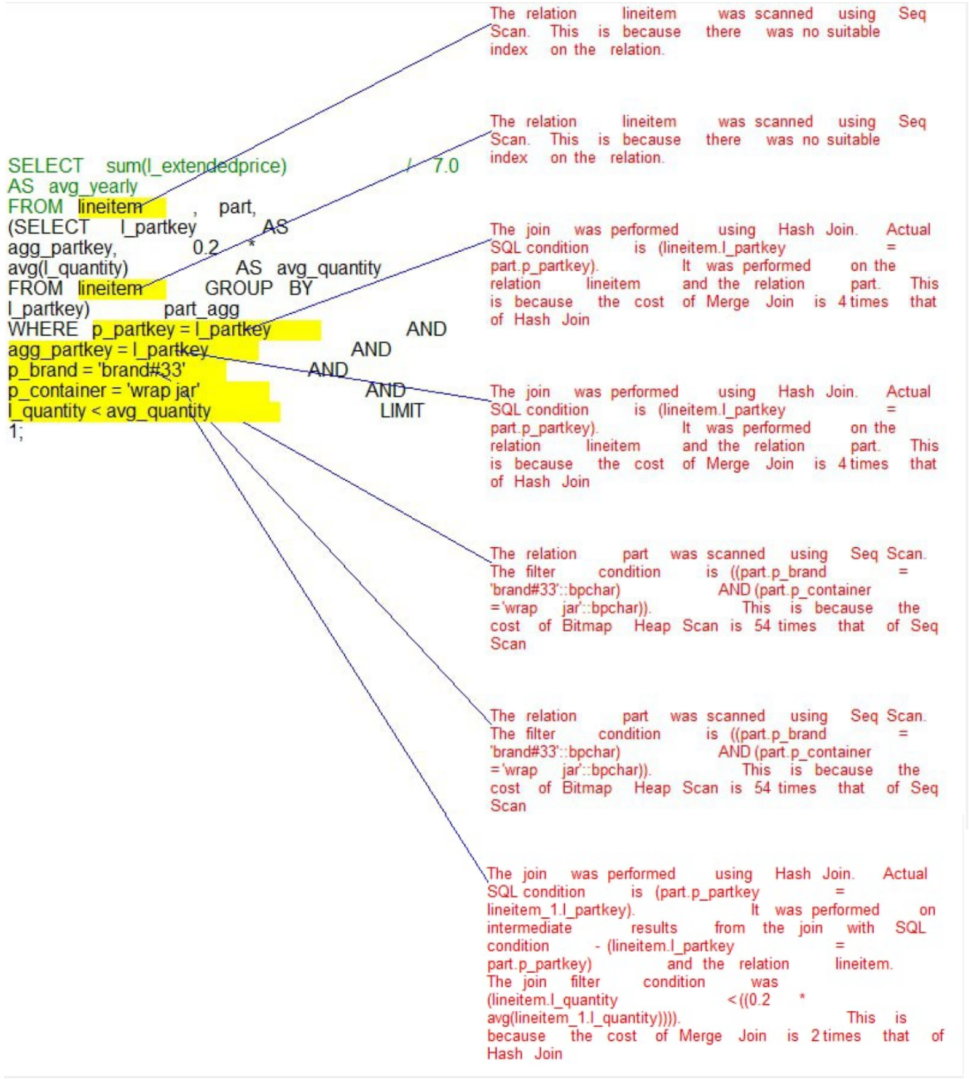
3. Result 4


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;
The relation lineitem was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation nation was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation customer was scanned using Index Scan. The index condition is (customer.c_custkey = orders.o_custkey). The filter condition is (customer.c_acctbal >'10'::numeric). This is because the cost of Bitmap Heap Scan is 9 times that of Index Scan
The join was performed using Index based Nested Loop Join. Actual SQL condition is (orders.o_orderkey = lineitem.l_orderkey). It was performed on intermediate results from the join with SQL condition - (lineitem.l_suppkey = supplier.s_suppkey) and the relation orders. The join filter condition was (orders.o_orderkey = lineitem.l_orderkey). This is because the cost of Hash Join is 2 times that of Index based Nested Loop Join
The join was performed using Hash Join. Actual SQL condition is (lineitem.l_suppkey = supplier.s_suppkey). It was performed on the relation lineitem and intermediate results from the join with SQL condition - (supplier.s_nationkey = nation.n_nationkey). This is because the cost of Merge Join is 4 times that of Hash Join
The join was performed using Nested Loop. Actual SQL condition is (supplier.s_nationkey = customer.c_nationkey). It was performed on intermediate results from the join with SQL condition - (orders.o_orderkey = lineitem.l_orderkey) and the relation customerThis is because the cost of Hash Join is 2 times that of Nested Loop
The join was performed using Hash Join. Actual SQL condition is (supplier.s_nationkey = nation.n_nationkey). It was performed on the relation supplier and intermediate results from the join with SQL condition - (nation.n_regionkey = region.r_regionkey). This is because the cost of Merge Join is 3 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (nation.n_regionkey = region.r_regionkey). It was performed on the relation nation and the relation region. This is because the cost of Merge Join is 2 times that of Hash Join
The relation region was scanned using Seq Scan. The filter condition is (region.r_name ='ASIA'::bpchar). This is because the cost of Bitmap Heap Scan is 3592 times that of Seq Scan
The relation orders was scanned using Index Scan. The index condition is None. The filter condition is ((orders.o_orderdate >= '1994-01-01'::date) AND (orders.o_orderdate < '1995-01-01'::date)). This is because the cost of Bitmap Heap Scan is 10 times that of Index Scan
The relation orders was scanned using Index Scan. The index condition is None. The filter condition is ((orders.o_orderdate >= '1994-01-01'::date) AND (orders.o_orderdate < '1995-01-01'::date)). This is because the cost of Bitmap Heap Scan is 10 times that of Index Scan
The relation customer was scanned using Index Scan. The index condition is (customer.c_custkey = orders.o_custkey). The filter condition is (customer.c_acctbal >'10'::numeric). This is because the cost of Bitmap Heap Scan is 9 times that of Index Scan
The relation supplier was scanned using Seq Scan. The filter condition is (supplier.s_acctbal >'20'::numeric). Surprising! The cost of Bitmap Heap Scan is 1 times that of Seq Scan

4. Result 5


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 >= '1995-01-01' 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;
The relation supplier was scanned using Seq Scan. This is because there was no suitable index on the relation.
The join was performed using Hash Join. Actual SQL condition is (lineitem.l_suppkey = supplier.s_suppkey). It was performed on the relation lineitem and intermediate results from the join with SQL condition - (supplier.s_nationkey = n1.n_nationkey). This is because the cost of Index based Nested Loop Join is 2 times that of Hash Join
The join was performed using Index based Nested Loop Join. Actual SQL condition is (orders.o_orderkey = lineitem.l_orderkey). It was performed on intermediate results from the join with SQL condition - (lineitem.l_suppkey = supplier.s_suppkey) and the relation orders. The join filter condition was (orders.o_orderkey = lineitem.l_orderkey). This is because the cost of Hash Join is 2 times that of Index based Nested Loop Join
The join was performed using Hash Join. Actual SQL condition is (orders.o_custkey = customer.c_custkey). It was performed on intermediate results from the join with SQL condition - (orders.o_orderkey = lineitem.l_orderkey) and intermediate results from the join with SQL condition - (lineitem.l_suppkey = supplier.s_suppkey). The join filter condition was (((n1.n_name = 'FRANCE'::bpchar) AND (n2.n_name = 'GERMANY'::bpchar)) OR((n1.n_name = 'GERMANY'::bpchar) AND (n2.n_name = 'FRANCE'::bpchar))). This is because the cost of Nested Loop is 2 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (supplier.s_nationkey = n1.n_nationkey). It was performed on the relation supplier and the relation nation. This is because the cost of Merge Join is 1003 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (customer.c_nationkey = n2.n_nationkey). It was performed on the relation customer and the relation nation. This is because the cost of Merge Join is 2 times that of Hash Join
The relation nation was scanned using Seq Scan. The filter condition is ((n1.n_name = 'FRANCE'::bpchar) OR(n1.n_name = 'GERMANY'::bpchar)). This is because the cost of Index Scan is 242 times that of Seq Scan
The relation nation was scanned using Seq Scan. The filter condition is ((n2.n_name = 'GERMANY'::bpchar) OR(n2.n_name = 'FRANCE'::bpchar)). This is because the cost of Index Scan is 3523 times that of Seq Scan
The relation nation was scanned using Seq Scan. The filter condition is ((n1.n_name = 'FRANCE'::bpchar) OR(n1.n_name = 'GERMANY'::bpchar)). This is because the cost of Index Scan is 242 times that of Seq Scan
The relation nation was scanned using Seq Scan. The filter condition is ((n2.n_name = 'GERMANY'::bpchar) OR(n2.n_name = 'FRANCE'::bpchar)). This is because the cost of Index Scan is 3523 times that of Seq Scan
The relation lineitem was scanned using Seq Scan. The filter condition is (lineitem.l_shipdate >= '1995-01-01'::date). This is because the cost of Bitmap Heap Scan is 3 times that of Seq Scan
The relation orders was scanned using Index Scan. The index condition is None. The filter condition is (orders.o_totalprice > '100'::numeric). This is because the cost of Seq Scan is 4 times that of Index Scan
The relation customer was scanned using Seq Scan. The filter condition is (customer.c_acctbal >'10'::numeric). This is because the cost of Index Scan is 10 times that of Seq Scan

5. Result 6


SELECT sum(l_extendedprice) / 7.0
AS avg_yearly
FROM lineitem , part,
(SELECT l_partkey AS
agg_partkey, 0.2 *
avg(l_quantity) AS avg_quantity
FROM lineitem GROUP BY
l_partkey) part_agg
WHERE p_partkey = l_partkey AND
agg_partkey = l_partkey AND
p_brand = 'brand#33' AND
p_container = 'wrap jar' AND
l_quantity < avg_quantity LIMIT
1;
The relation lineitem was scanned using Seq Scan. This is because there was no suitable index on the relation.
The relation lineitem was scanned using Seq Scan. This is because there was no suitable index on the relation.
The join was performed using Hash Join. Actual SQL condition is (lineitem.l_partkey = part.p_partkey). It was performed on the relation lineitem and the relation part. This is because the cost of Merge Join is 4 times that of Hash Join
The join was performed using Hash Join. Actual SQL condition is (lineitem.l_partkey = part.p_partkey). It was performed on the relation lineitem and the relation part. This is because the cost of Merge Join is 4 times that of Hash Join
The relation part was scanned using Seq Scan. The filter condition is ((part.p_brand = 'brand#33'::bpchar) AND (part.p_container = 'wrap jar'::bpchar)). This is because the cost of Bitmap Heap Scan is 54 times that of Seq Scan
The relation part was scanned using Seq Scan. The filter condition is ((part.p_brand = 'brand#33'::bpchar) AND (part.p_container = 'wrap jar'::bpchar)). This is because the cost of Bitmap Heap Scan is 54 times that of Seq Scan
The join was performed using Hash Join. Actual SQL condition is (part.p_partkey = lineitem_1.l_partkey). It was performed on intermediate results from the join with SQL condition - (lineitem.l_partkey = part.p_partkey) and the relation lineitem. The join filter condition was (lineitem.l_quantity < ((0.2 * avg(lineitem_1.l_quantity)))). This is because the cost of Merge Join is 2 times that of Hash Join