CZ4031 Project 2

<u>Group 37</u>
Goh Shan Ying (U1921497C)
Malcolm Tan (U2022160E)
Ng Chi Hui (U1922243C)
Shannen Lee (U2021991G)

1. Introduction

This report outlines our design and implementation of a system that automatically generates user-friendly explanations of changes to query execution plan (QEP) in a relational database management system (RDBMS). The program will help provide users with the changes in the SQL Query as well as the QEP using natural language and visual explanation to aid their understanding. This report aims to demonstrate the usefulness of this program in aiding end-users who may not be proficient in database technology to understand the changes in the execution of their queries during data exploration.

2. Dataset

The TPC-H database was utilized as the dataset for this project. This instance of the database consists of eight tables, namely *customer, lineitem, nation, orders, part, partsupp*, *region* and *supplier*

3. Project Methodology

The implementation of our project consists of:

1. Generate Query Execution Plan (QEP)
2. Generate the Graph Visualisation for each QEP
3. Generate the differences in natural language
    a. Identify the clauses that have changed in the two SQL query
    b. Identify the changes in the QEP Plan
4. Send all results to the Graphical Interface (UI)

Input	Output
Two Related Queries	1. Explain the steps taken to execute both queries using natural language. 2. Present visual representations of the execution plan of both queries in a tree structure. 3. Describe the differences in the execution plans of the two queries and provide a natural language explanation for the reasons behind these differences.

4. Project File

In this file, the main application is called, which is the Tkinter application in the interface.py file.

5. Interface File

5.1 Designing UI

We first designed our Graphic User Interface (GUI) using Figma. By designing the GUI first, we could identify and address design issues early in the process before any coding has begun. This can save time and effort by reducing the need for redesigns and changes later in the development cycle.

We created several drafts as seen in Figure 1, and then chose the finalised design after a round of discussion and feedback within our team, as seen in Figure 2.

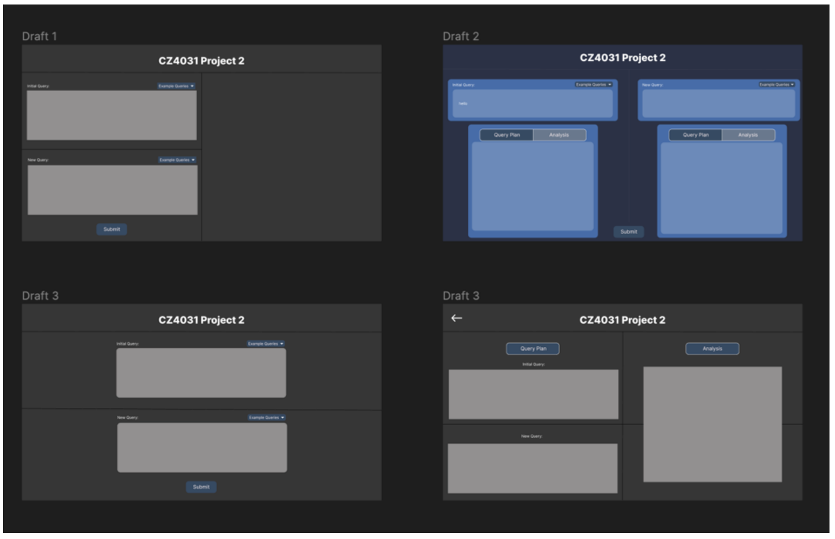


Figure 1. Drafts of the GUI design

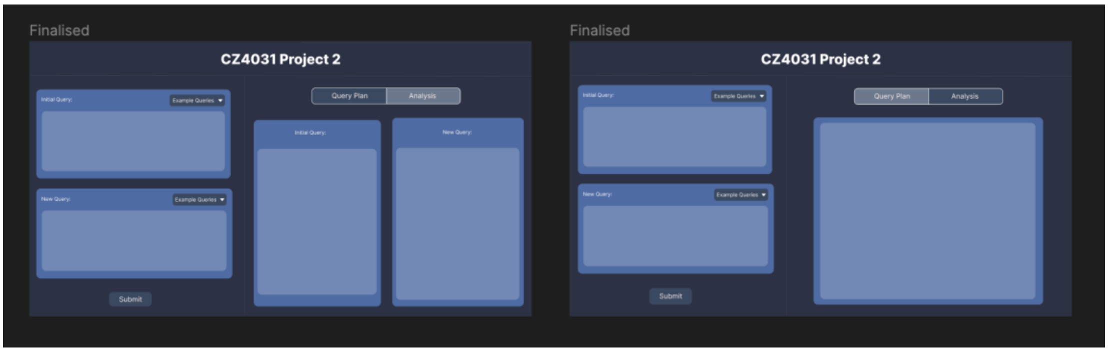


Figure 2. Finalised GUI design

We prioritised having the user input for SQL Queries to always be in the same window as the results, this would allow the user to easily enter more inputs and view the subsequent changes.

5.2 Creating UI

Our GUI is made using Tkinter and Ttkbootstrap. Tkinter is a very accessible python framework used to create simple GUI elements. It has great built-in functionalities like pack() or grid() that allowed for geometrical management, helping us built a user-friendly interface. Ttkbootstrap is a theme extension for Tkinter that provides modern styling aesthetics inspired by Bootstrap.

5.2.1 Login

We decided to incorporate a login feature for the connection to the Postgres database. Users will have to enter the following as seen in Figure 3.

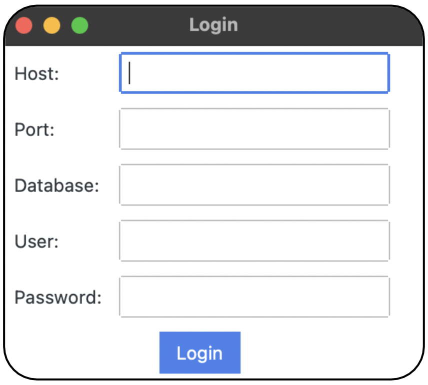


Figure 3. Image of the login window

The user will only be able to proceed after the connection is successful.

After logging in, the user would view the overall interface, as seen in Figure 4 below.

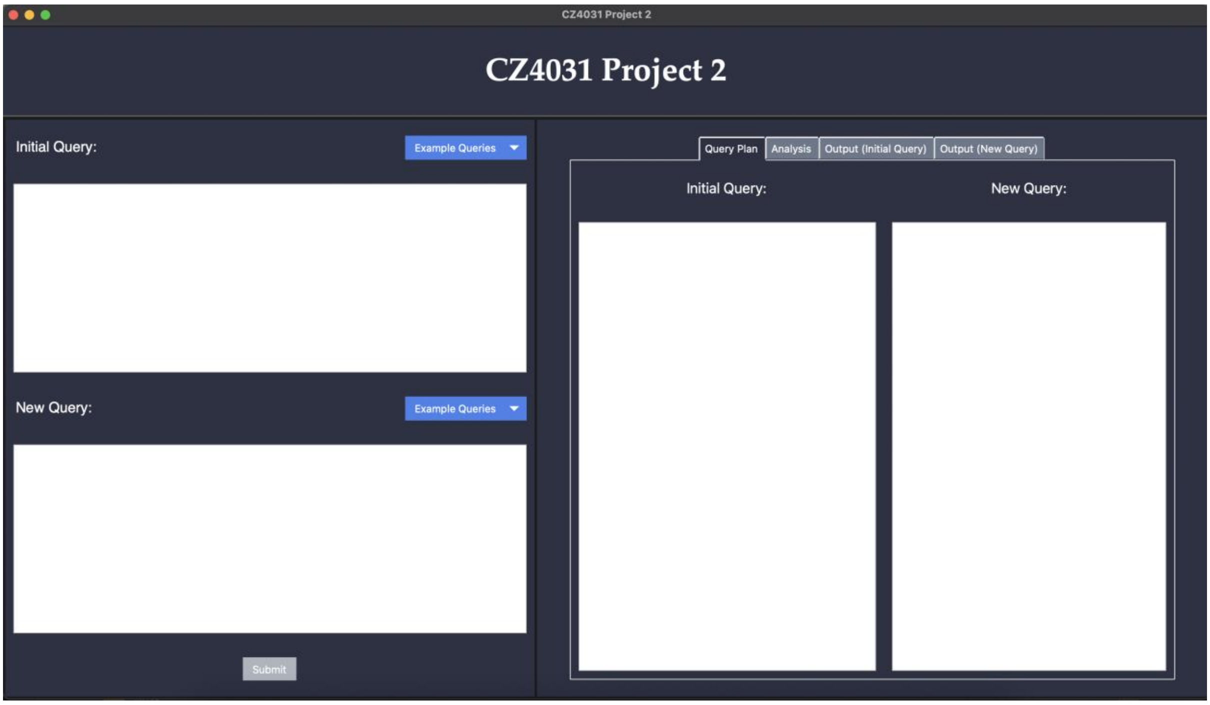


Figure 4. Overall interface of our application

5.2.2 Query Inputs

In the left window, the user can enter their desired queries for comparison. In the 'Initial Query' and 'New Query' text boxes, the user has the option to enter their own custom queries or choose from one of our 22 example queries to help any users who would like to test our systems comfortably, as seen in Figure 5 below.

Figure 5. Example Queries for user to choose from

5.2.3 Query Plans

After the user enters their queries, the ‘Query Plan’ tab will open and display the query plans in a visual format. The plans are placed in adjacent arrangement to allow user to easily compare the query plan diagrams, as seen in Figure 6 below.

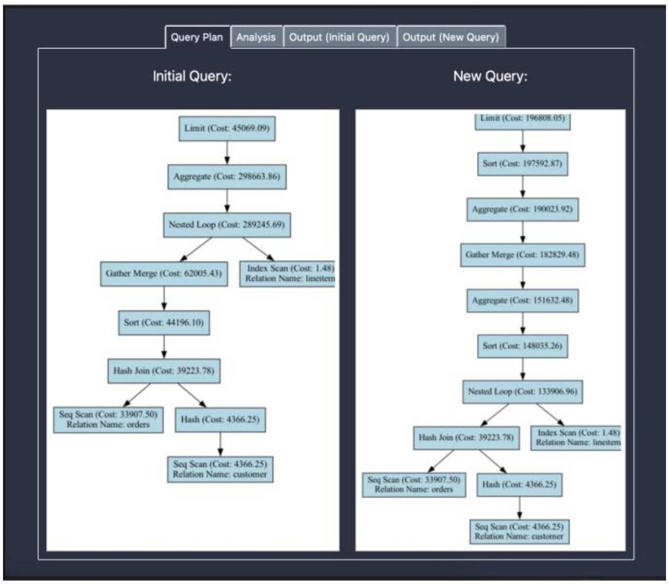


Figure 6. Example of Query Plan Diagrams

5.2.4 Analysis

If the user would like more detailed information on the differences between the queries, they can select the 'Analysis' tab for further explanation in written form. Our analysis will display the differences between the SQL lines, the join and relations used, and the total execution cost of each plan, as seen in Figure 7.

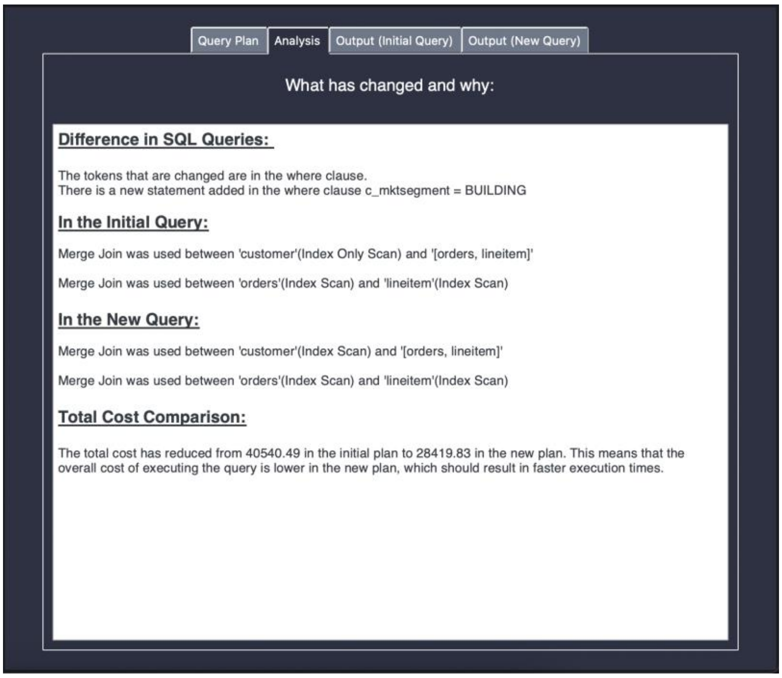


Figure 7. Example of analysis of differences in SQL queries

5.2.5 Table View

For further visualisation of the queries, we implemented a structured table format to display the output of the initial and new SQL queries. By selecting either the ‘Output (Initial Query)’ or the ‘Output (New Query)’ tabs, the user can generate the tables of their desired query by clicking the ‘Generate Table’ button, as seen in Figure 8 and 9.

Figure 8. ‘Generate Table’ button for user to click

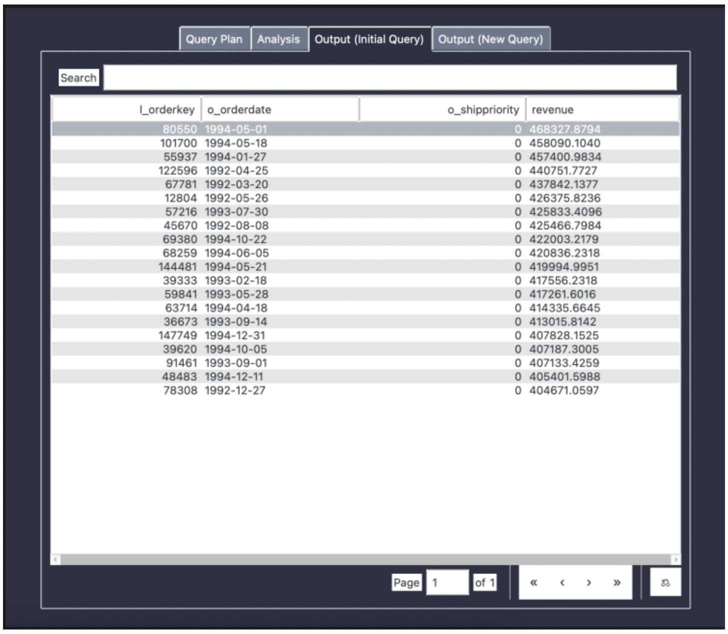


Figure 9. Example of a generated table

We allow the user to choose whether to generate the tables instead of auto-generating it upon the submission of the SQL queries because the generating of tables can take a substantial amount of time, especially when the number of output tuples are large.

6. Explain File

In this section, the program that generates and analyses query execution plans in a database will be explored in detail. It covers various components, including the Pre-processing class, responsible for connecting to the database, generating Query Execution Plans (QEPs), and validating user input; the Graph Generator section, which produces and saves visual representations of QEPs for ease of comparison and analysis; and the Query Processing section, which explains how the program parses queries into JZON-izable parse trees and generates query differences using the Deepdiff module.

Altogether, this section presents a complete overview of the program's functionality and features.

6.1. Pre-processing

The Pre-processing class is responsible for establishing a connection with the database, generating a Query Execution Plan (QEP) for the query, and validating the user's input. These steps are necessary to ensure that the query can be executed successfully and to prevent errors. The steps are clearly illustrated in Figure 10 below.

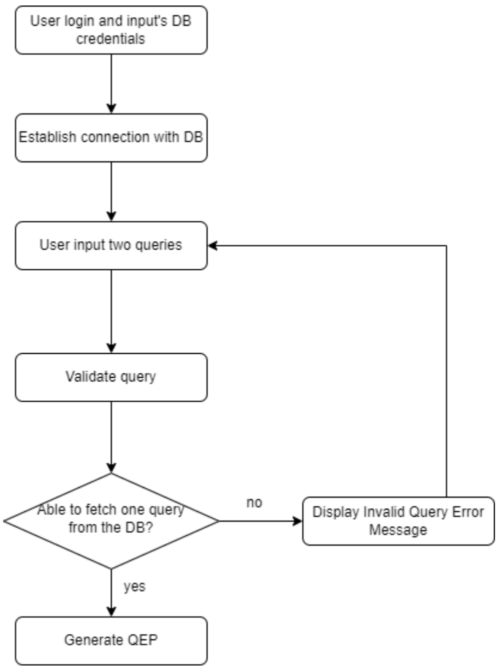


Figure 10. Flowchart to illustrate pre-processing steps

To establish a connection with the database, the Pre-processing class uses the DBConnection class to read the connection details from the login information entered and connect to the PostgreSQL server using the psycopg2 library. Once connected, the Pre-processing class can execute SQL queries on the database.

The Pre-processing class can generate a QEP for the input query using the get_query_plan() method. The QEP provides a detailed visual representation of the execution steps involved in the query. The QEP can be used to identify bottlenecks and inefficiencies in the execution process, which can help optimize the query's performance.

Prior to generating the QEP, the input is validated using the validate_query() method to ensure that it can be executed on the connected database. This validation includes fetching one result from the database to ensure that the query is properly formed and can be executed. If the validation fails, an error message is returned to the user. If the validation is successful, the get_query_plan() method executes the query using the EXPLAIN (FORMAT JSON) command, which directs PostgreSQL to generate a QEP in JSON format.

The QEP can used for several next steps, such as generating graphs and providing explanations to the user about the comparison of the execution process between the original and modified queries.

6.2. Graph Generator

This section is dedicated to the generation of a graph to aid the visualization of query execution. This visualization is beneficial in comprehending the flow of data and operations during query execution, thus facilitating comparison between different queries. The section highlights the "build_dot" function, which is a recursive approach for generating the graph in the DOT format, and the "generate_graph" function, which generates and saves the graph. (The DOT syntax is a plain text graph description language that is used to describe graphs and networks.)

6.2.1. Building Graphs with DOT Syntax

The purpose of building a graph in DOT syntax is to visually represent a QEP for better understanding and comparison.

The "build_dot" function is a recursive method that generates a graph in the DOT format. This function takes a query execution plan as input and outputs a DOT-formatted graph that represents the plan. The graph is generated by creating nodes for each operation in the plan and edges that connect the nodes to show the data flow between the operations. The edges in the graph represent the order in which the operations are executed.

The function starts by assigning a unique ID to each node in the graph, based on a hash of the node's properties. The node's label is then set to the node's type and cost. If the node represents a table or a relation, the node's label also includes the name of the table or relation. Finally, the function recursively calls itself for each child node in the plan to create a complete graph.

6.2.2. Generating Graphs from Query Execution Plans

After the DOT-formatted graph is generated by the "build_dot" function, the "generate_graph" method applies visual properties to the resulting graph, such as the shape and colour of the nodes and edges. Then, the method generates a Png image of the graph and passes it to the application for display, providing a convenient way to compare and understand different query execution plans.

Example Graph generated as seen in Figure 11 below.

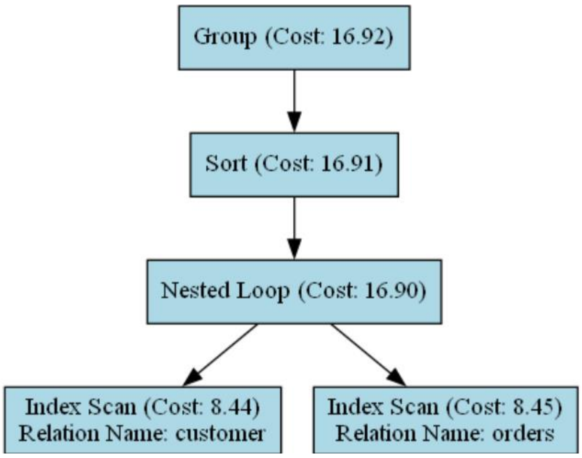


Figure 11. Example of Query Plan Graph

In summary, our program offers a user-friendly solution for generating and saving visual representations of query execution plans in a single image. This feature greatly simplifies the comparison and analysis of different plans, enabling users to easily comprehend the flow of the query execution process.

6.3. Query Processing

This section focuses on how we parse the two different query to compare and identify the differences between the two Query inputted by the user.

6.3.1. Parsing the Query

Inspired by NoSQL query writing and SQL expression syntax, we propose parsing queries into JZON-izable parse trees to simplify comparison. Complex queries with nested subqueries, multiple joins, and brackets can become difficult to compare. Thus, by converting the SQL query into a parse tree and then converting it to a JZON (JSON-like) format, it would be easier to compare queries because the structure would be standardized and uniform across different queries. This could be particularly useful for comparing queries that have nested subqueries, multiple joins, or complex conditions with multiple brackets. To do this, we made use of the mo-sql-parsing module and were easily able to parse the query into a Json format with its respective tokens.

A sample parsed query will look like this, in Figure 12 below:

```
query = {
    'select': [
        {'value': 'l_orderkey'},
        {'value': 'o_orderdate'},
        {'value': 'o_shippriority'},
        {'value': {'sum': {'mul': ['l_extendedprice', {'sub': [1, 'l_discount']}]}}, 'name':
'revenue'}
    ],
    'from': ['customer', 'orders', 'lineitem'],
    'where': {
        'and': [
            {'eq': ['customer.c_custkey', 'orders.o_orderkey']},
            {'eq': ['lineitem.l_orderkey', 'orders.o_orderkey']},
            {'lt': ['orders.o_orderdate', {'literal': '1995-03-15'}]}
        ]
    },
    'groupby': [
        {'value': 'l_orderkey'},
        {'value': 'o_orderdate'},
        {'value': 'o_shippriority'}
    ],
    'orderby': [
        {'value': 'revenue', 'sort': 'desc'},
        {'value': 'o_orderdate'}
    ],
    'limit': 20
}
```

Figure 12. Sample of a parsed query

6.3.2 Generating the Query Difference

By parsing each query into a JZON-izable format, we were able to easily identify the differences in key-value pairs and print out all the differences between the 2 SQL queries.

To do that, we made use of a module Deepdiff that will be able to return the deep difference between two python objects. Under the hood, DeepDiff compares the objects by iterating over their keys and values, comparing the types, and recursively iterating over any nested objects. It identifies the differences in a detailed way, including the exact path to the differences in the objects, the type of difference (added, removed, or changed), and the values before and after the change.

An example after running the two parsed tree with Deepdiff with return the following output as seen in Figure 13.

```
query_update = {
    'iterable_item_added': {
        "root['where']['and'][3]": {
            'eq': ['c_mktsegment', {'literal': 'BUILDING'}]
        }
    }
}
```

Figure 13. Example of the output

By parsing the results returned by the Deepdiff, we were able to extract out the parts that have been changed in the query and form back the SQL statements that have been changed which will be explained in the part below.

6.3.3 Translating the query difference into natural language

Once we receive the changes from the Deepdiff dictionary, we pass this information to a function called 'comparing_changes'. This function compares two SQL queries, the original and the modified one, to find out if there are any changes made.

6.3.3.1 Changes in Clauses

To do this, the function first checks if there are any differences between the two queries. It does this by comparing the two queries using a DeepDiff object. If the DeepDiff object indicates that there are no differences between the two queries, the function will tell the user "No changes". But if there are differences, the function will continue to identify which part of the query has been changed. But if there are differences, the function identifies which part of the query has been changed, like the SELECT, FROM, GROUP BY, LIMIT, or WHERE clauses.

To do this, the function looks for specific keywords in the DeepDiff object that indicate which clause has changed. For example, if the SELECT clause has been modified, the

DeepDiff object might contain the keyword "SELECT". Similarly, if the FROM clause has been modified, the DeepDiff object might contain the keyword "FROM".

Once it has identified the changed clauses, the function adds them to a set of "results". The function then combines these results into a single string that lists all the changed clauses. For example, if the SELECT and WHERE clauses have been modified, the function might return a string that says, "The tokens that are changed are in the SELECT, WHERE clause".

6.3.3.2 Changes made in WHERE clause

If there were changes made to the WHERE clause, the function further checks what specific changes were made, like a change in values or adding/removing conditions. It extracts this information and adds it to a string called 'diffString'.

To enhance readability, our program translates the Deepdiff object back to natural language. Once the program has identified the relevant changes and translated them, they are added to the 'diffString' variable. The translation is achieved through various functions, which are discussed in section 6.3.3.3.

Finally, the function checks whether any dictionary items were added or removed from the WHERE clause. If a dictionary item was added, the function identifies which operator was added and whether the addition was made to the first or second query. If a dictionary item was removed, the function identifies which operator was removed and whether the removal was made from the first or second query. In both cases, the function appends the relevant information to 'diffString'.

For instance, changes to the dictionary could involve adding OR clauses, changing an OR clause to an AND clause, or vice versa.

After all the relevant changes have been identified and appended to the 'diffString' variable, the function returns this variable. The 'diffString' variable contains a string that describes the differences between the two SQL queries. This string is then passed to the application, which displays the differences to the user.

6.3.3.3 Convert the DeepDiff object back to natural language

To convert the Deepdiff object back to natural language, our code implements several helper functions that work together to transform the parsed SQL query dictionaries into more readable descriptions of the differences. One such function is the 'cleaning_literal' function, which is used to extract the right-hand side literal or the last item of a dotted notation. This function is essential for generating accurate and meaningful natural language descriptions of the changes.

Additionally, our code uses several other functions such as 'convert_to_and_of_or_with_and_of', 'convert_or_clause', 'convert_and_clause', and 'convert_and_condition' to generate the natural language descriptions of the changes.

More details on the functions:

- 'convert_to_and_of_or_with_and_of': This function takes a list of items and generates a natural language description of the list, combining items with the word "and" and using "or" to separate sublists. For example, given the list ['A', 'B', ['C', 'D']], this function would return the string "(A and B) or (C and D)". This function is used to describe changes in the WHERE clause involving OR and AND operators.

- convert_or_clause: This function takes a dictionary representing an OR clause in the WHERE clause and generates a natural language description of the clause. For example, given the dictionary {'or': [{'eq': ['foo', 1]}, {'ne': ['bar', 2]}]}, this function would return the string "either foo is equal to 1 or bar is not equal to 2". This function is used to describe changes in OR clauses.

- convert_and_clause: This function takes a dictionary representing an AND clause in the WHERE clause and generates a natural language description of the clause. For example, given the dictionary {'and': [{'eq': ['foo', 1]}, {'ne': ['bar', 2]}]}, this function would return the string "foo is equal to 1 and bar is not equal to 2". This function is used to describe changes in AND clauses.

- convert_and_condition: This function takes a dictionary representing an AND condition in the WHERE clause and generates a natural language description of the condition. For example, given the dictionary {"and": [{"eq": {"foo": "1"}}, {"eq": {"foo": "5"}}]}, this function would return the string "foo = 1 AND foo = 5". This function is used to describe changes in individual conditions in the WHERE clause.

The combination of these helper functions enables our program to transform the Deepdiff object into easily understandable descriptions for the end user. This approach allows for a more detailed and accurate presentation of the differences between two SQL queries, providing users with an easier way to identify any changes made.

*** Please refer to the Appendix section to review the outputs generated by our program.**

The entire process is illustrated in Figure 14 below.

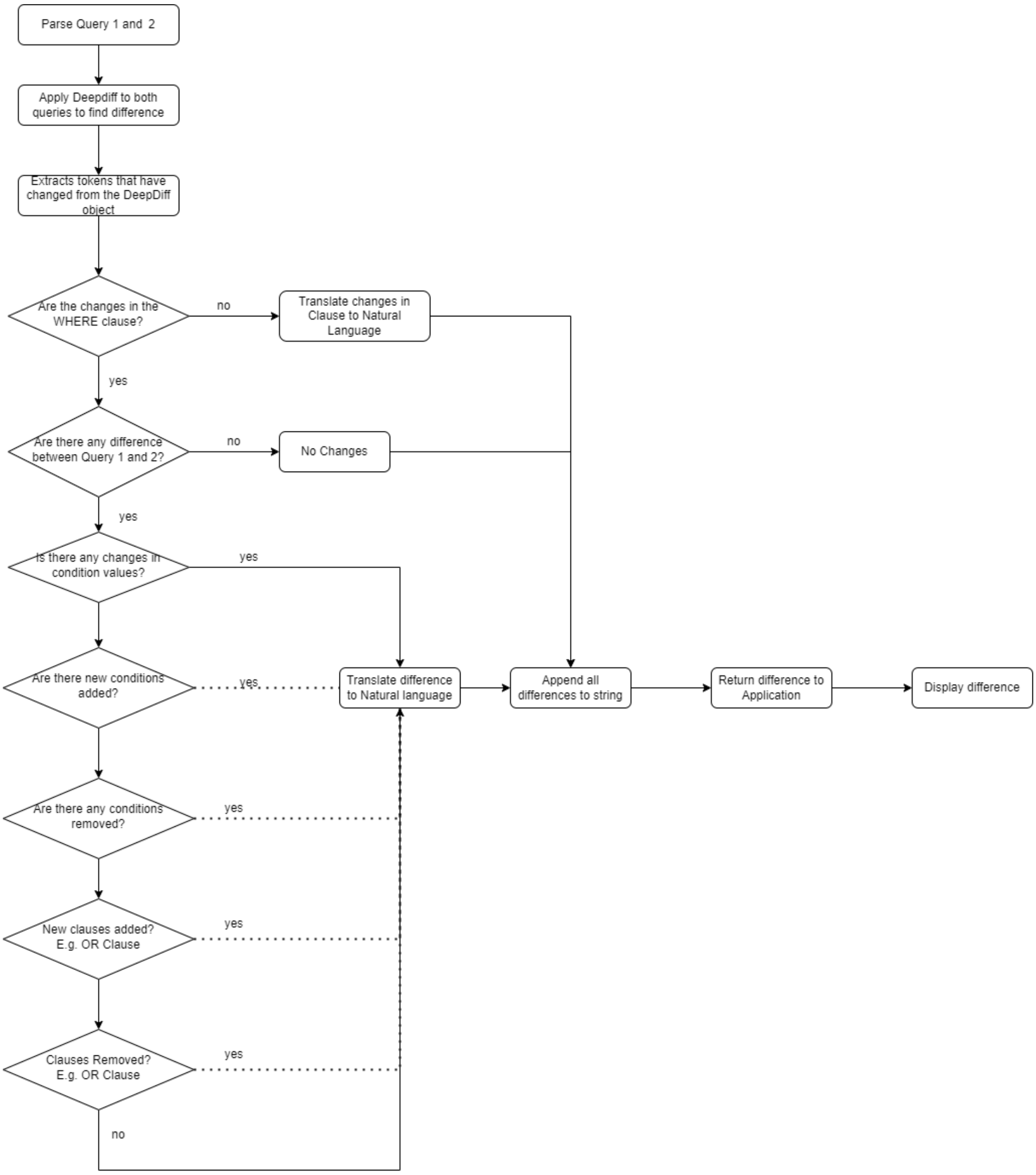


Figure 14. Flowchart of translation process to natural language

6.4. Query Plan Comparison

6.4.1 Join and Relations Comparison

With respect to our project requirements, our query plan comparison must generate user friendly descriptions of any developments made by the two plans. Therefore, we decided that the most important information while allowing generality would be to focus on any divergence made by nested loops or join operations. Additionally, we will supplement the data extracted with the relations and scan type used for those operations. A summary in the form of a flowchart is given in Figure 15 below.

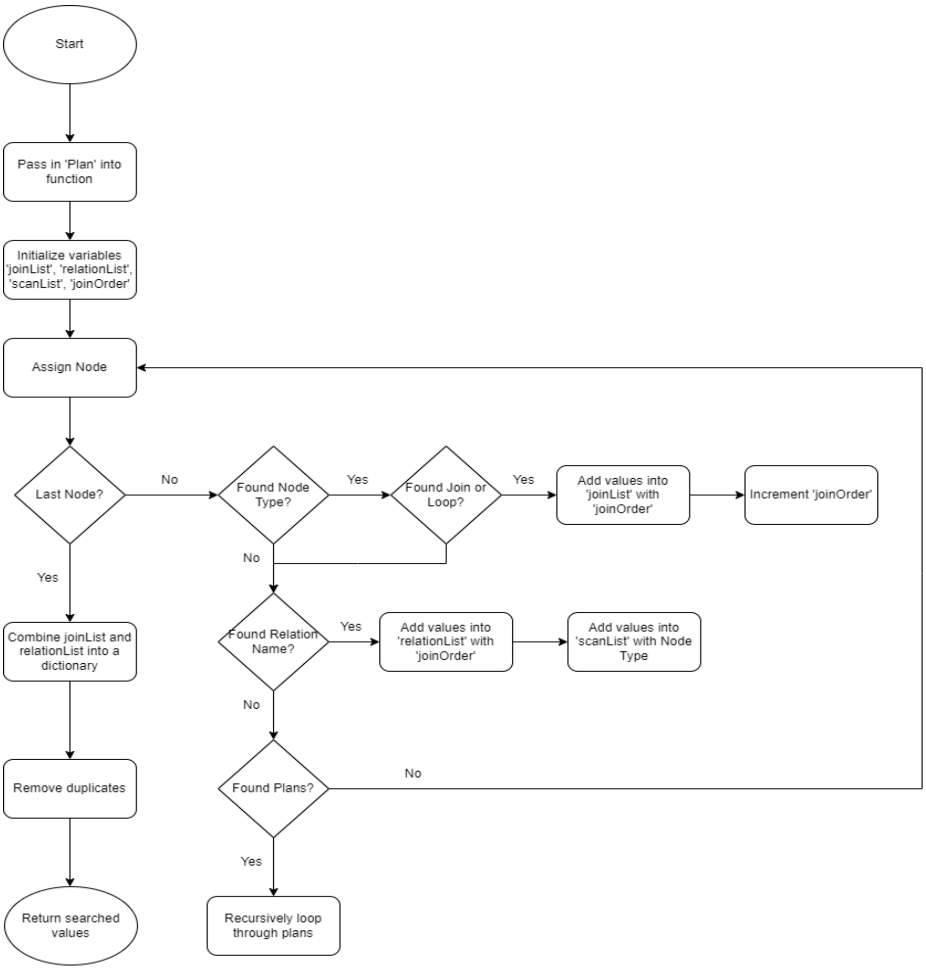


Figure 15. Flowchart of searching joins and relations

6.4.2 Cost Comparison

As an additional detail to our comparison, we have factored in the total execution cost for each plan.

In the context of a PostgreSQL execution plan, the total execution cost refers to the estimated cost of executing each node in the plan. This cost represents an estimate of the total amount of work that needs to be done to execute the query, taking into account factors such as I/O and CPU usage, memory usage, and network communication. A summary in the form of a flowchart is given in Figure 16 below.

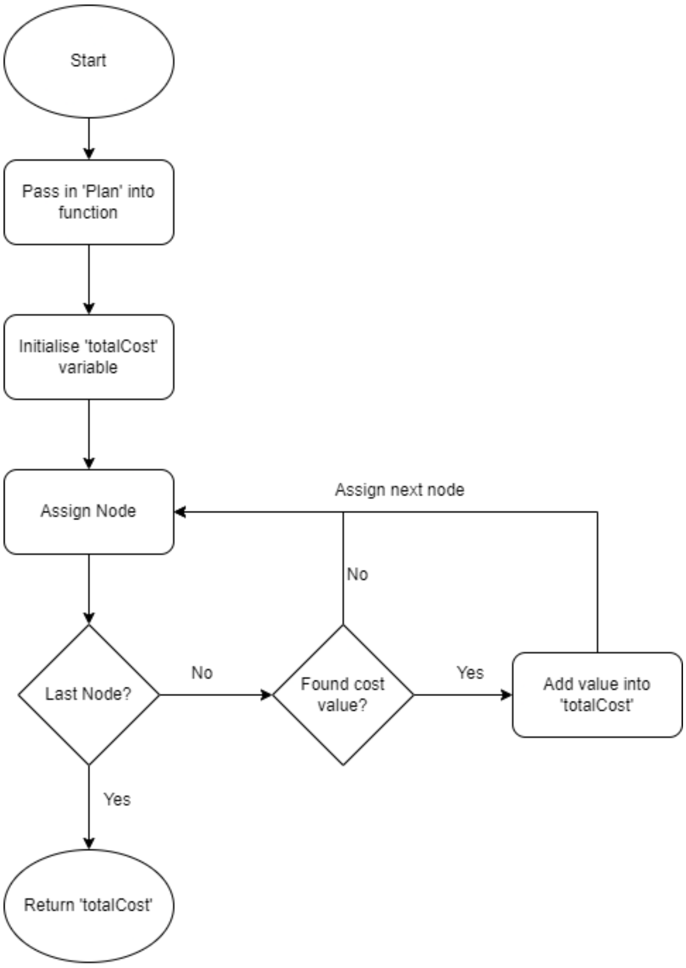


Figure 16. Flowchart of cost calculations

This would allow the user to identity the most efficient plan between their queries. We achieve this by iterating through each node and adding up the total cost found.

7. Limitation & Future Works

7.1. Constrains of QEP Comparison

In the course of this project, we have placed greater emphasis on detecting changes that occurred in the Query Execution Plans (QEPs) due to modifications made in the WHERE clause. However, any changes that occurred in other clauses, such as FROM, SELECT, and GROUP BY, INSERT, DELETE which could also result in differences in the QEP tree, were not identified. In light of this, it would be beneficial to extend this project's scope to include the identification of changes in other operations, such as AGGREGATE, HASH, APPEND, and SUBPPLAN, among others.

7.2. Constrains of SQL Parsing

Although we were successful in identifying differences in the SQL query (i,e changes in SELECT, FROM, WHERE, GROUP BY, HAVING, LIMIT) using Deepdiff, there were some difficulties in parsing the resulting object back to a modified SQL query. The issue arose due to the varied formats returned by Deepdiff when there were changes in multiple clauses, and the importance of the order of SQL. In this project, our focus was on modifications within the where clause, and identifying specific changes within it. While the program can still detect changes in other clauses such as from, group by, limit, select, or having, it cannot pinpoint specific changes.

In order to solve this problem, we believe the issues with parsing the Deepdiff object back to a modified SQL query can be addressed by implementing a custom parser that can handle the different formats returned by Deepdiff. This would enable the program to accurately identify changes in multiple clauses and order of SQL. Secondly, expanding the program's scope to identify specific changes in other clauses such as from, group by, limit, select, or having could enhance its usefulness. This could be achieved by developing separate algorithms to parse and compare each clause. By doing this, the program can provide more detailed information on the specific changes made to each clause.

Lastly, to improve the accuracy of the comparison, it may be useful to incorporate a natural language processing (NLP) component, which is a field of artificial intelligence and machine learning. By converting SQL queries into natural language statements, the program would have a better understanding of the queries and be able to identify any changes made to them more accurately.

7.3. Long runtime for complex queries (Postgres)

One of the most common limitations of running a program in PostgreSQL is the time it takes to execute a query, especially when dealing with complex queries that involve multiple joins and result in a large table. In such cases, PostgreSQL might take a long time to load and might even result in a timeout or crash. One way to mitigate this issue is by setting a limit on the SQL query.

It is recommended for all users to set a limit to their SQL queries, as it helps to prevent the system from overloading and crashing.

However, there are some cases where setting a limit might not be enough. For example, running a query that involves multiple joins and has an **order by** clause followed by a limit might still take a very long time to load and PostgreSQL timeout.

This is because the results will still be ordered first before the limit is applied. In such cases, it is recommended to optimize the query by using appropriate indexes or by breaking down the query into smaller, more manageable chunks.

To mitigate the limitations of running a program in PostgreSQL, users can take several steps. First, it is recommended to set a limit on the SQL queries to prevent overloading and crashing. Additionally, users should optimize their queries by using appropriate indexes and breaking them down into smaller, more manageable chunks.

8. Appendix

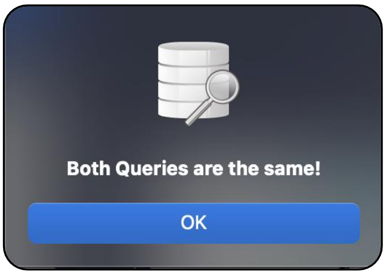
8.1 Experiments

The sample queries used in the experiments are also accessible in our interface via the 'Example Queries' drop box.

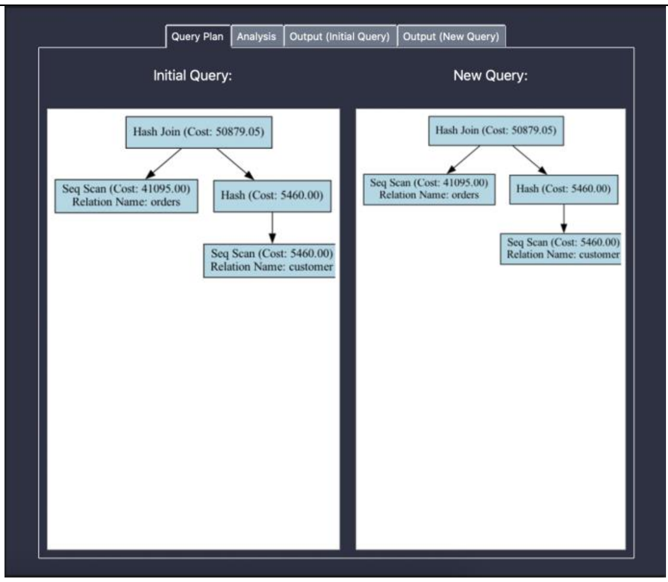
8.1.1 No Change in Queries

Initial Query **(Query 1)**	New Query **(Query 1)**
SELECT * FROM customer C, orders O WHERE C.c_mktsegment like 'BUILDING' and C.c_custkey = O.o_custkey	SELECT * FROM customer C, orders O WHERE C.c_mktsegment like 'BUILDING' and C.c_custkey = O.o_custkey

Warning Message



Query Plan Tab

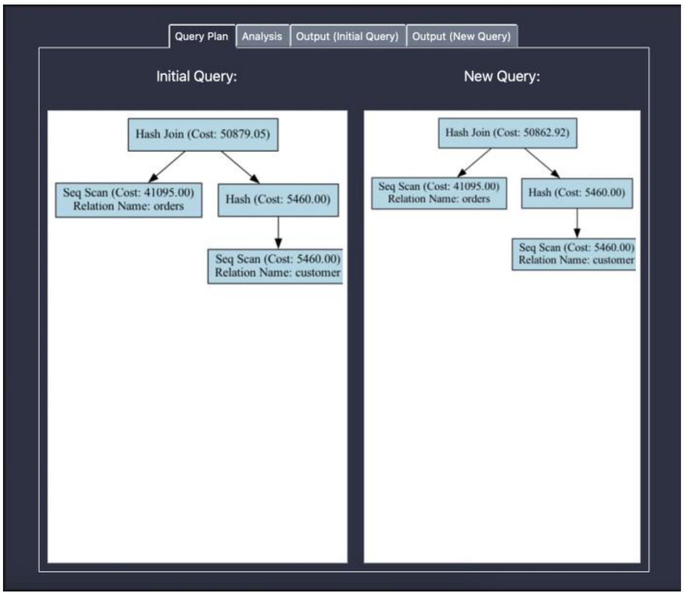


Analysis Tab

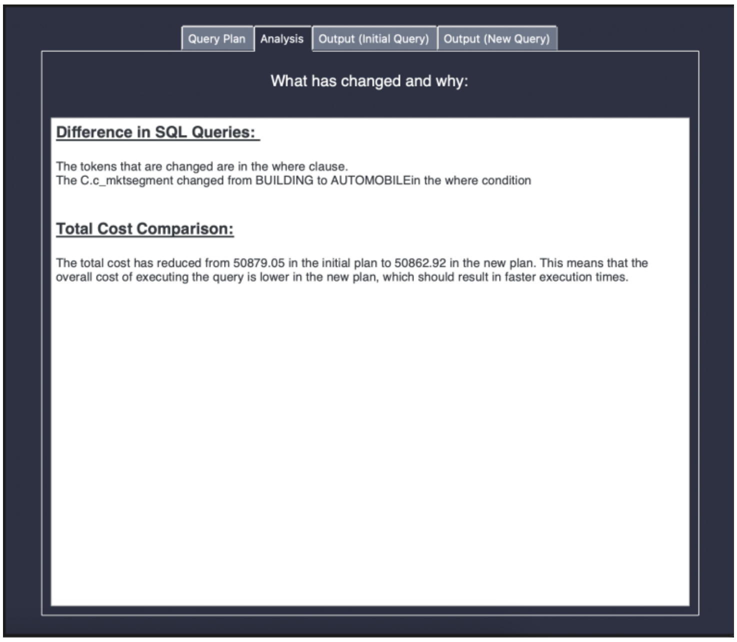

8.1.2. Value Changed

Initial Query **(Query 1)**	New Query **(Query 2)**
SELECT * from customer C, orders O WHERE C.c_mktsegment like 'BUILDING' AND C.c_custkey = O.o_custkey	SELECT * from customer C, orders O WHERE C.c_mktsegment like 'AUTOMOBILE' AND C.c_custkey = O.o_custkey

Query Plan Tab

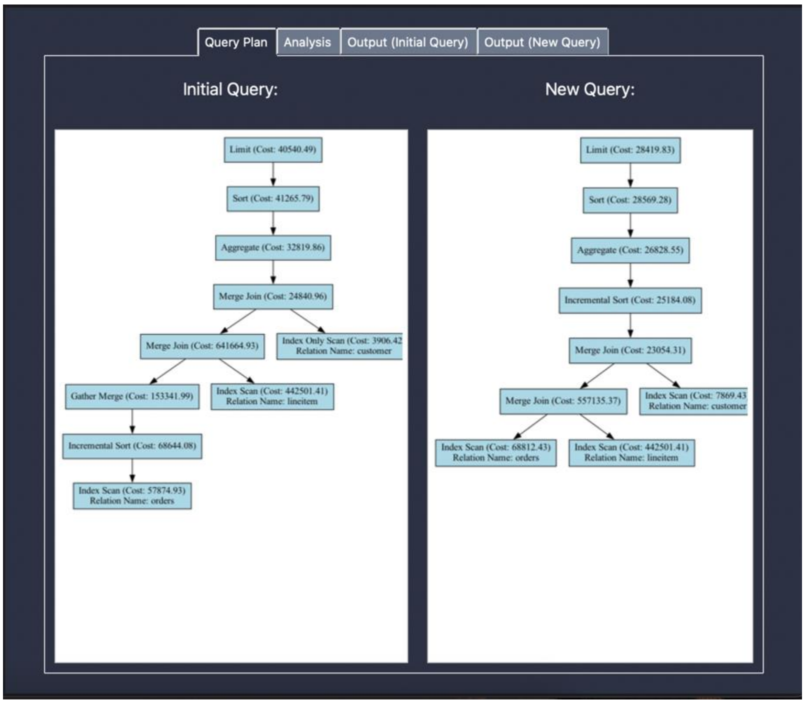


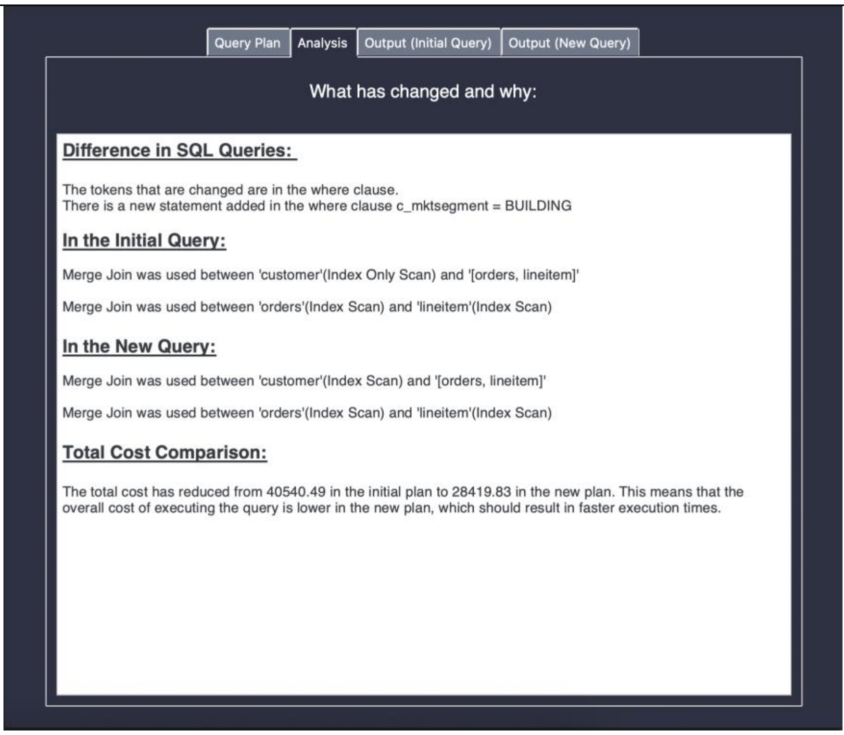
Analysis Tab



8.1.3 Addition of condition

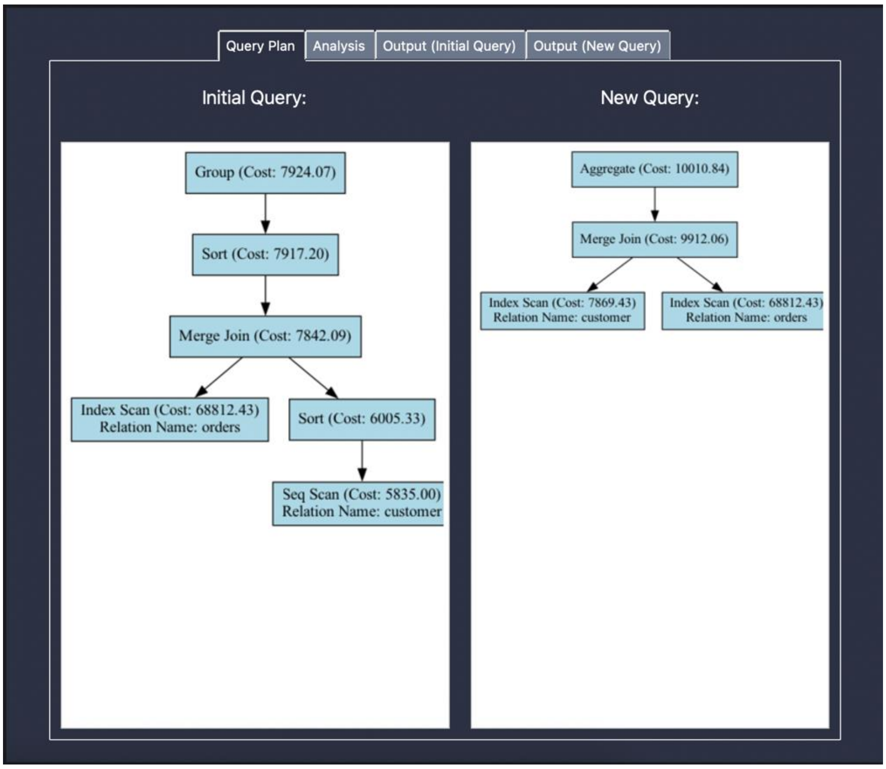
Initial Query **(Query 3)**	New Query **(Query 4)**
SELECT l_orderkey, o_orderdate, o_shippriority, sum((l_extendedprice) * (1-l_discount)) as revenue FROM customer, orders, lineitem WHERE customer.c_custkey = orders.o_orderkey AND lineitem.l_orderkey = orders.o_orderkey AND orders.o_orderdate < '1995-03-15' GROUP BY l_orderkey, o_orderdate, o_shippriority ORDER BY revenue desc, o_orderdate LIMIT 20	SELECT l_orderkey, o_orderdate, o_shippriority, sum((l_extendedprice) * (1-l_discount)) as revenue FROM customer, orders, lineitem WHERE customer.c_custkey = orders.o_orderkey AND lineitem.l_orderkey = orders.o_orderkey AND orders.o_orderdate < '1995-03-15' AND c_mktsegment = 'BUILDING' GROUP BY l_orderkey, o_orderdate, o_shippriority ORDER BY revenue desc, o_orderdate LIMIT 20

Query Plan Tab


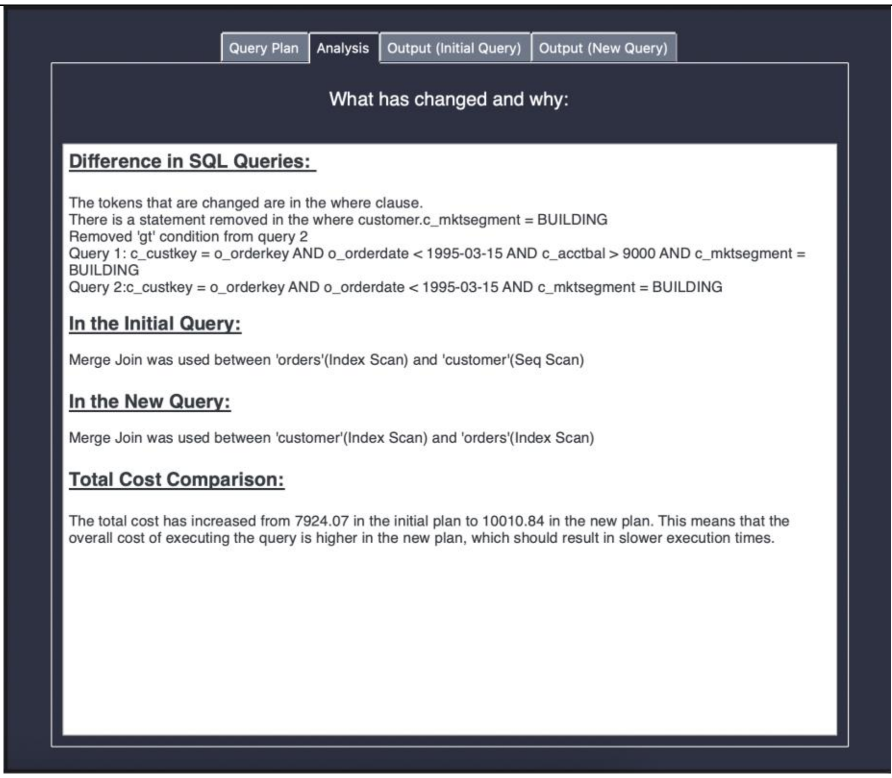
Analysis Tab


8.1.4 Removal of Condition

Initial Query **(Query 5)**	New Query **(Query 6)**
SELECT o_orderdate, o_shippriority FROM customer, orders WHERE customer.c_custkey = orders.o_orderkey AND orders.o_orderdate < '1995-03-15' AND customer.c_acctbal > '9000' AND customer.c_mktsegment = 'BUILDING' GROUP by o_orderdate, o_shippriority	SELECT o_orderdate, o_shippriority FROM customer, orders WHERE customer.c_custkey = orders.o_orderkey AND orders.o_orderdate < '1995-03-15' AND customer.c_mktsegment = 'BUILDING' GROUP by o_orderdate, o_shippriority

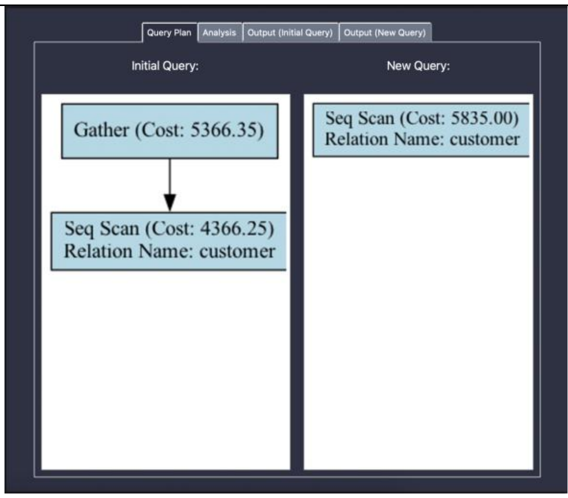
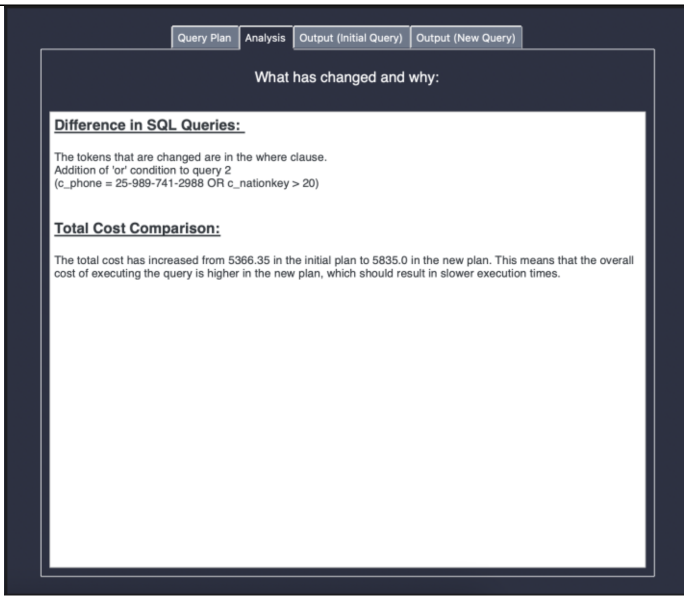
Query Plan Tab


Analysis Tab



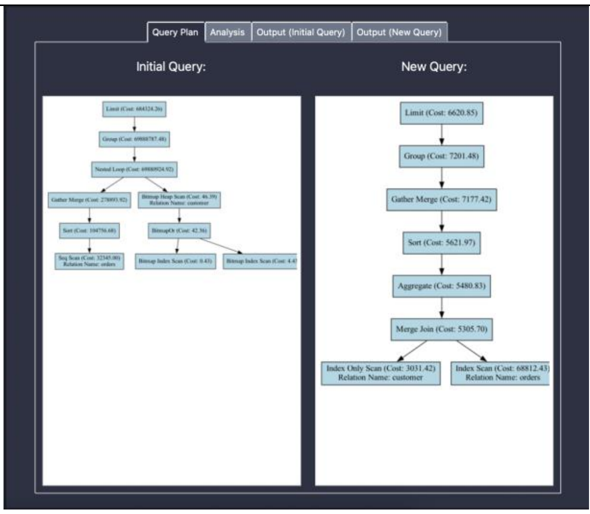
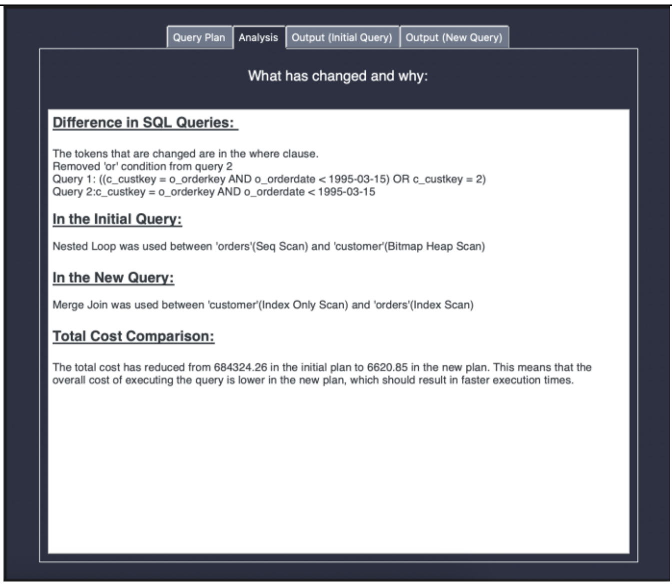
8.1.5 Addition of OR Clause

Initial Query **(Query 7)**	New Query **(Query 8)**
SELECT * FROM customer WHERE customer.c_phone = '25-989-741-2988'	SELECT * FROM customer WHERE customer.c_phone = '25-989-741-2988' OR customer.c_nationkey > '20'

Query Plan Tab

Analysis Tab


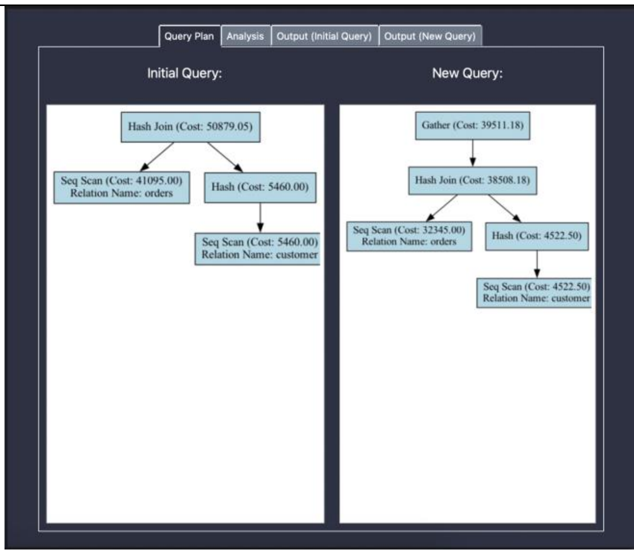
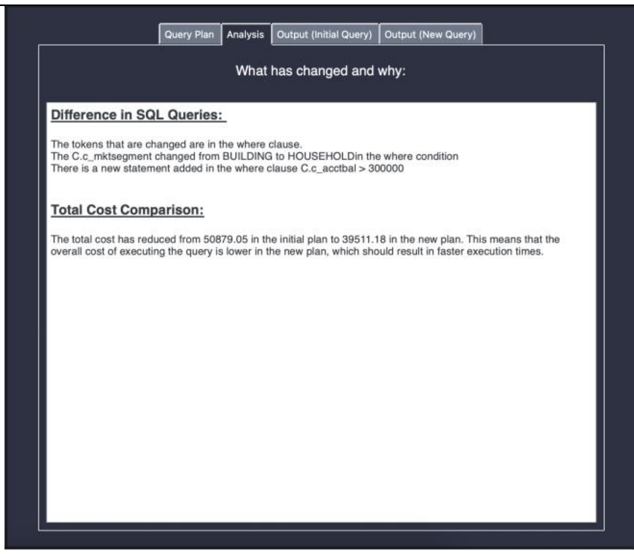
8.1.6 Removal of OR Clause

Initial Query **(Query 9)**	New Query **(Query 10)**
SELECT o_orderdate, o_shippriority FROM customer, orders WHERE customer.c_custkey = orders.o_orderkey AND orders.o_orderdate < '1995-03-15' OR customer.c_custkey= '2' GROUP by o_orderdate, o_shippriority LIMIT 20	SELECT o_orderdate, o_shippriority FROM customer, orders WHERE customer.c_custkey = orders.o_orderkey AND orders.o_orderdate < '1995-03-15' GROUP by o_orderdate, o_shippriority LIMIT 20

Query Plan Tab

Analysis Tab


8.1.7 Multiple changes to WHERE clause

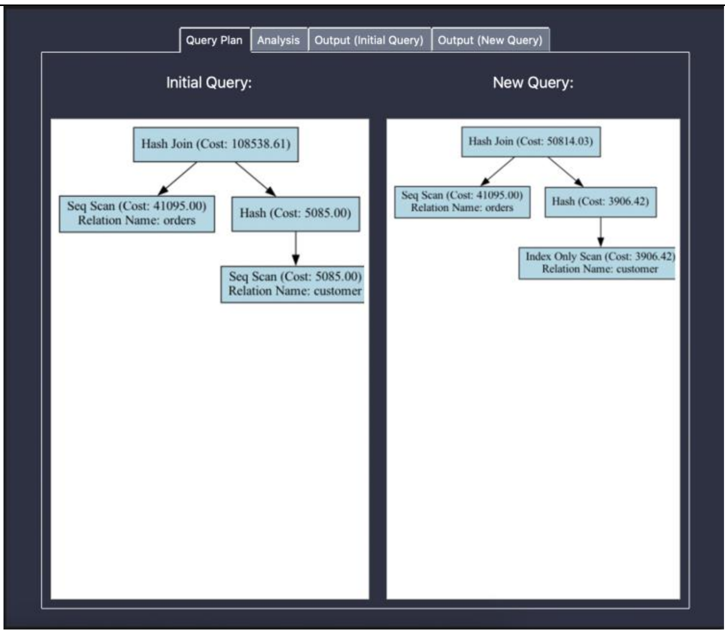
Initial Query **(Query 11)**	New Query **(Query 12)**
SELECT * FROM customer C, orders O WHERE C.c_mktsegment like 'BUILDING' AND C.c_custkey = O.o_custkey	SELECT * FROM customer C, orders O WHERE C.c_mktsegment like 'HOUSEHOLD' AND C.c_custkey = O.o_custkey AND C.c_acctbal > '300000'

Query Plan Tab

Analysis Tab


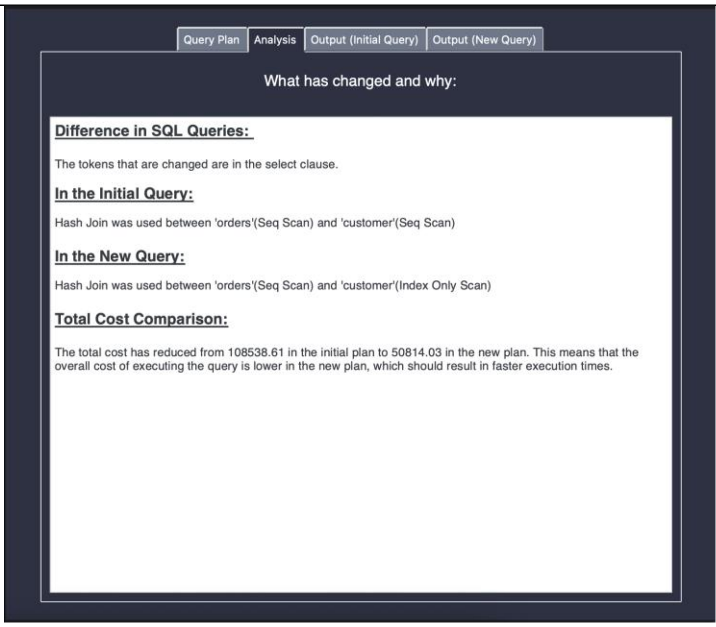
8.1.8 Changed the SELECT clause

Initial Query **(Query 13)**	New Query **(Query 14)**
SELECT * FROM customer C, orders O WHERE C.c_custkey = O.o_custkey	SELECT C.c_custkey FROM customer C, orders O WHERE C.c_custkey = O.o_custkey

Query Plan Tab

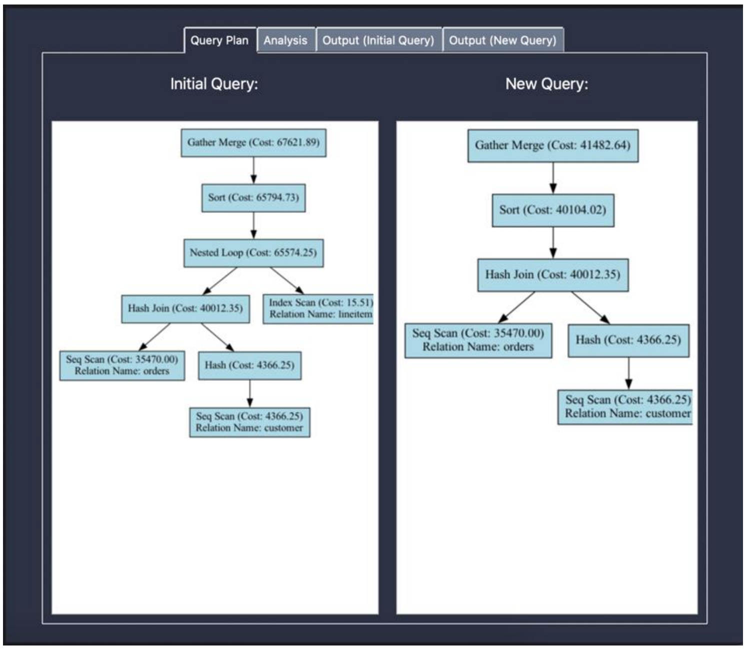


Analysis Tab

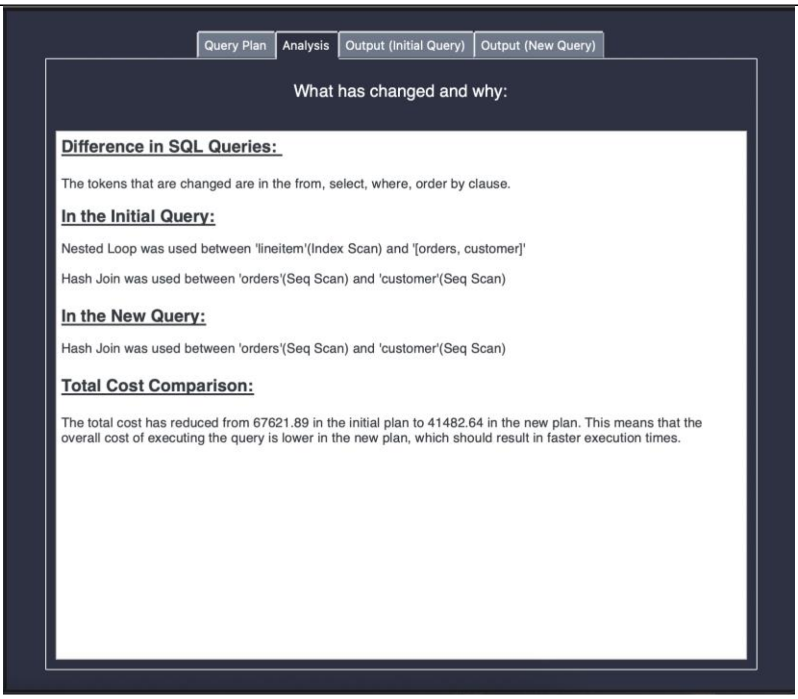


8.1.9 Changed the FROM clause

Initial Query **(Query 15)**	New Query **(Query 16)**
SELECT orders.o_orderkey, customer.c_custkey, lineitem.l_partkey, lineitem.l_quantity, lineitem.l_extendedprice FROM orders, customer, lineitem WHERE orders.o_custkey = customer.c_custkey AND orders.o_orderkey = lineitem.l_orderkey AND orders.o_orderdate BETWEEN '1994-01-01' AND '1994-01-31' AND lineitem.l_discount BETWEEN 0.05 AND 0.10 AND customer.c_mktsegment = 'AUTOMOBILE' ORDER BY orders.o_orderkey, customer.c_custkey, lineitem.l_partkey;	SELECT orders.o_orderkey, customer.c_custkey FROM orders, customer WHERE orders.o_custkey = customer.c_custkey AND orders.o_orderdate BETWEEN '1994-01-01' AND '1994-01-31' AND customer.c_mktsegment = 'AUTOMOBILE' ORDER BY orders.o_orderkey, customer.c_custkey;

Query Plan Tab


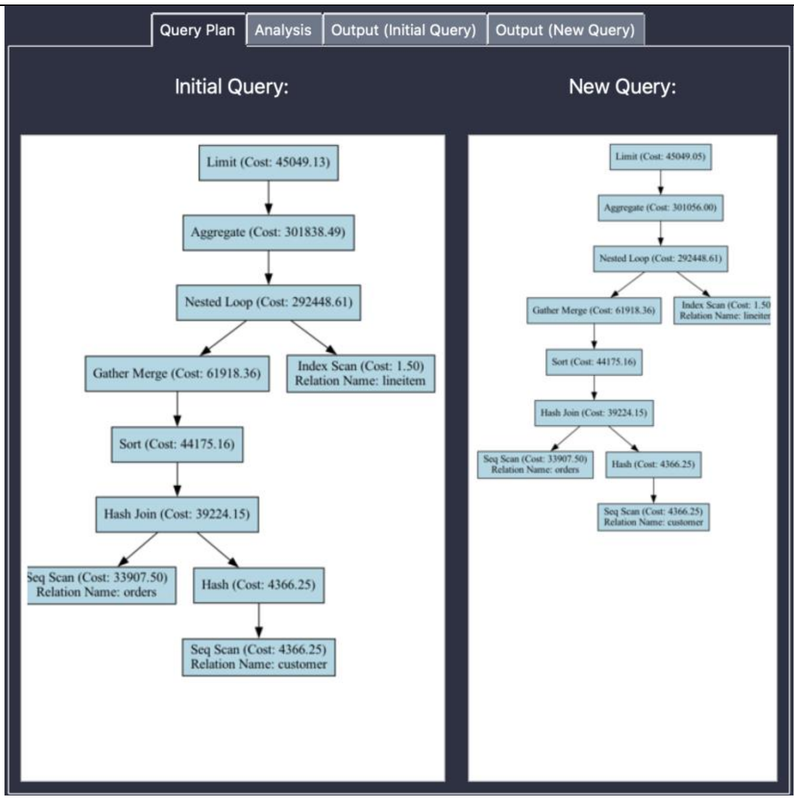
Analysis Tab



8.1.10 Changed the GROUP BY clause

Initial Query **(Query 17)**	New Query **(Query 18)**
SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) AS revenue, o_orderdate, o_shippriority FROM customer, orders, lineitem WHERE c_mktsegment = 'HOUSEHOLD' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < '1995-03-15' AND l_shipdate > '1995-03-15' GROUP BY l_orderkey, o_orderdate, o_shippriority HAVING SUM(l_extendedprice*(1-l_discount)) > 10000 ORDER BY o_orderdate LIMIT 10;	SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) AS revenue, o_orderdate FROM customer, orders, lineitem WHERE c_mktsegment = 'HOUSEHOLD' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < '1995-03-15' AND l_shipdate > '1995-03-15' GROUP BY l_orderkey, o_orderdate HAVING SUM(l_extendedprice*(1-l_discount)) > 10000 ORDER BY o_orderdate LIMIT 10;

Query Plan Tab

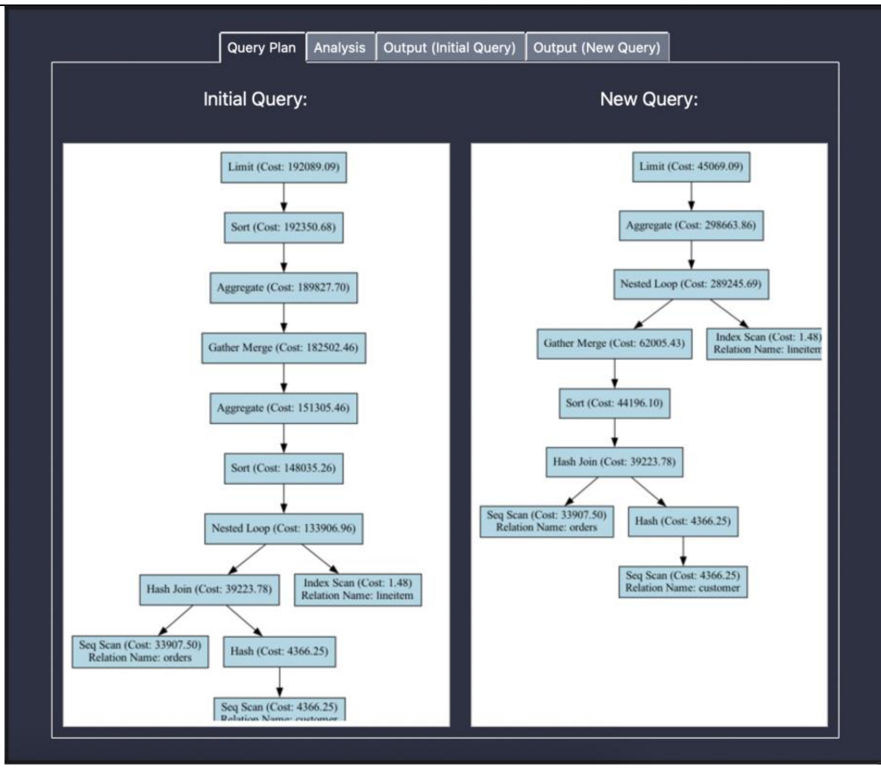


Analysis Tab

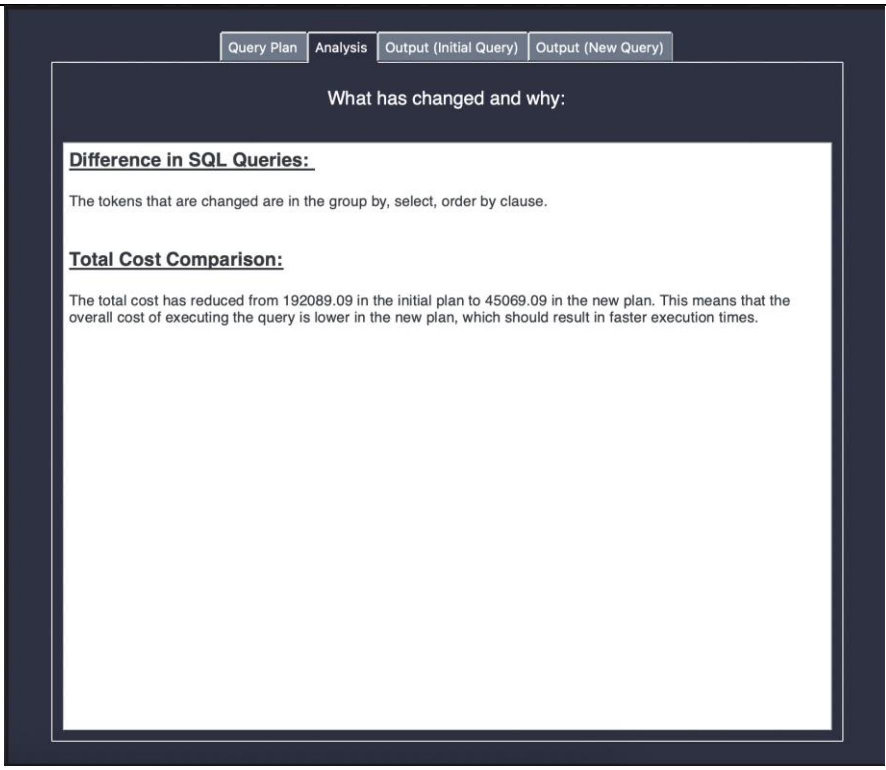
8.1.11 Changed the ORDER BY clause

Initial Query **(Query 19)**	New Query **(Query 20)**
SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) AS revenue, o_orderdate FROM customer, orders, lineitem WHERE c_mktsegment = 'HOUSEHOLD' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < '1995-03-15' AND l_shipdate > '1995-03-15' GROUP BY l_orderkey, o_orderdate HAVING SUM(l_extendedprice*(1-l_discount)) > 10000 ORDER BY revenue DESC, o_orderdate LIMIT 10;	SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) AS revenue, o_orderdate, o_shippriority FROM customer, orders, lineitem WHERE c_mktsegment = 'HOUSEHOLD' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < '1995-03-15' AND l_shipdate > '1995-03-15' GROUP BY l_orderkey, o_orderdate, o_shippriority HAVING SUM(l_extendedprice*(1-l_discount)) > 10000 ORDER BY o_orderdate LIMIT 10;

Query Plan Tab

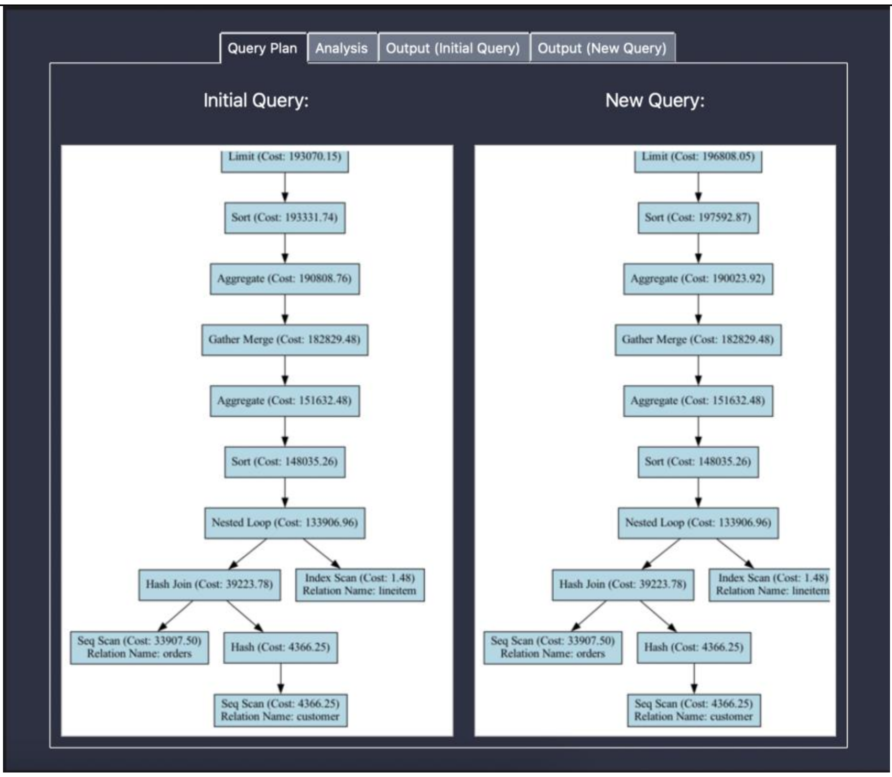


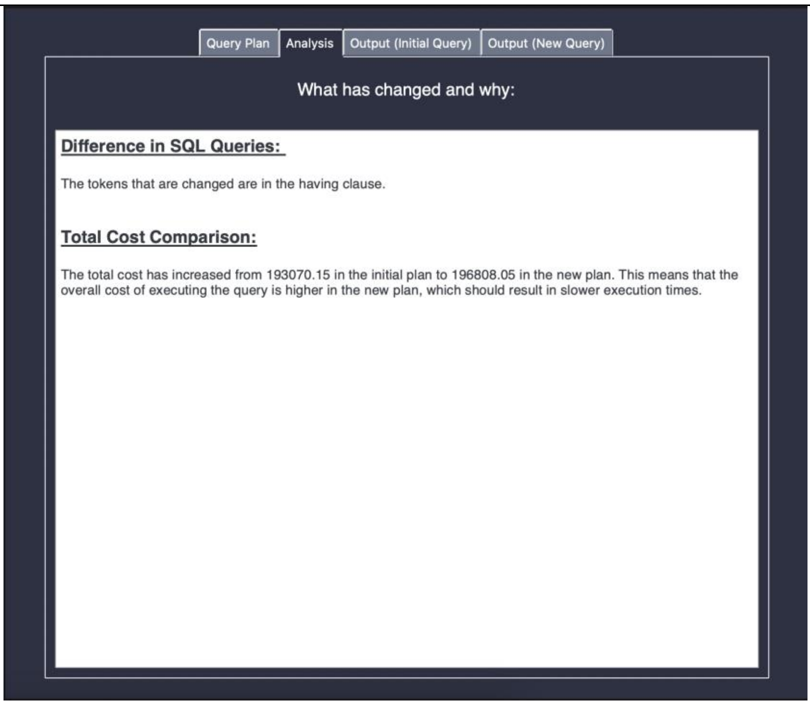
Analysis Tab



8.1.12 Changed the HAVING clause

Initial Query **(Query 21)**	New Query **(Query 22)**
SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) AS revenue, o_orderdate, o_shippriority FROM customer, orders, lineitem WHERE c_mktsegment = 'HOUSEHOLD' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < '1995-03-15' AND l_shipdate > '1995-03-15' GROUP BY l_orderkey, o_orderdate, o_shippriority HAVING SUM(l_extendedprice*(1-l_discount)) > 10000 ORDER BY revenue DESC, o_orderdate LIMIT 10;	SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) AS revenue, o_orderdate, o_shippriority FROM customer, orders, lineitem WHERE c_mktsegment = 'HOUSEHOLD' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < '1995-03-15' AND l_shipdate > '1995-03-15' GROUP BY l_orderkey, o_orderdate, o_shippriority ORDER BY revenue DESC, o_orderdate LIMIT 10;

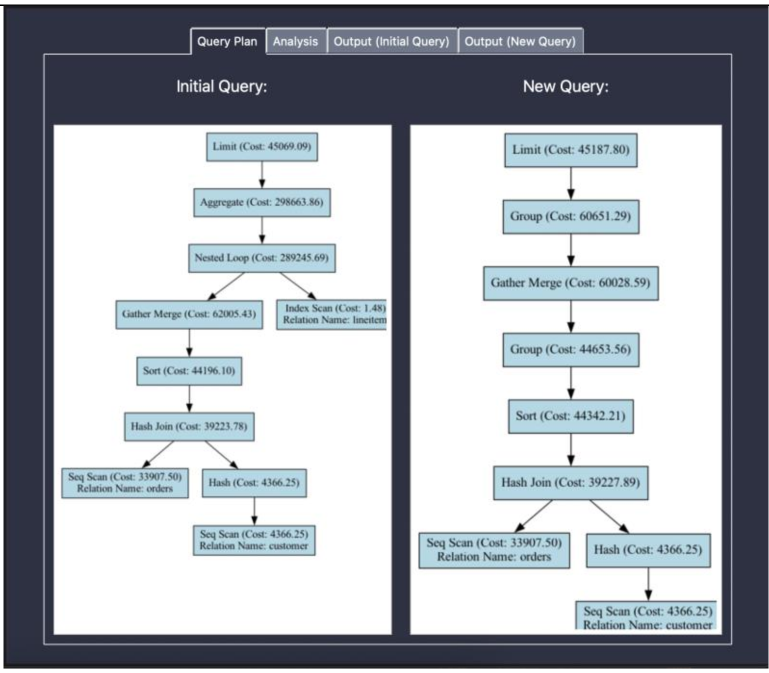
Query Plan Tab


Analysis Tab


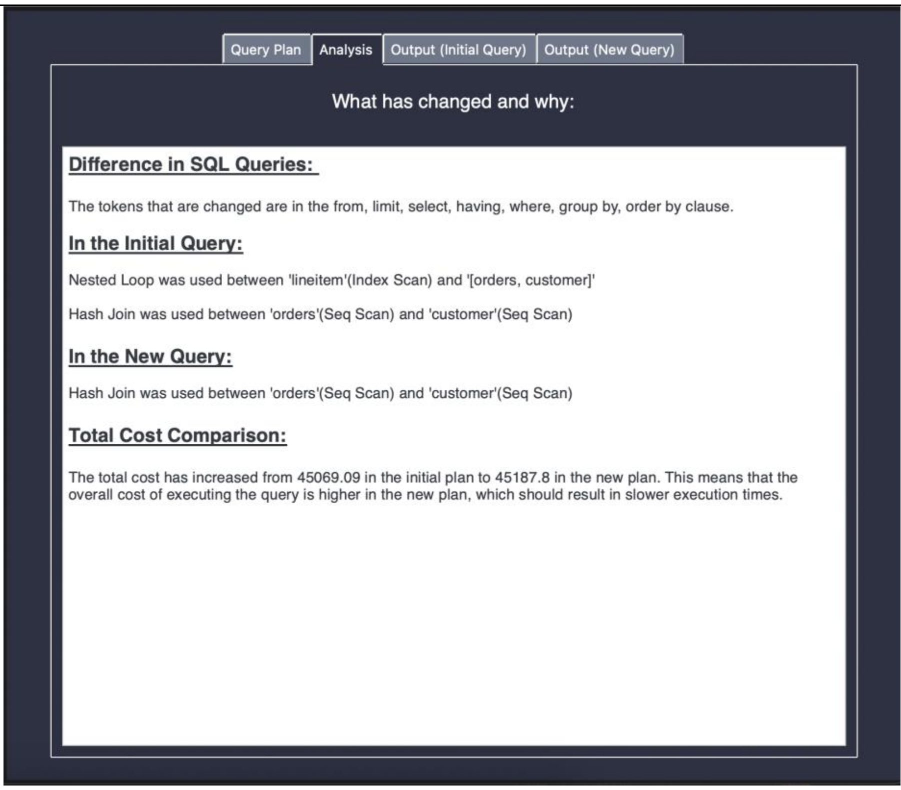
8.1.13 Combination of changes in multiple clauses

Initial Query **(Query 23)**	New Query **(Query 24)**
SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) AS revenue, o_orderdate, o_shippriority FROM customer, orders, lineitem WHERE c_mktsegment = 'HOUSEHOLD' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < '1995-03-15' AND l_shipdate > '1995-03-15' GROUP BY l_orderkey, o_orderdate, o_shippriority HAVING SUM(l_extendedprice*(1-l_discount)) > 10000 ORDER BY o_orderdate LIMIT 10;	SELECT o_orderdate, c_custkey FROM customer, orders WHERE c_mktsegment = 'BUILDING' AND c_custkey = o_custkey AND o_orderdate < '1995-03-15' GROUP BY o_orderdate, c_custkey ORDER BY c_custkey LIMIT 12;

Query Plan



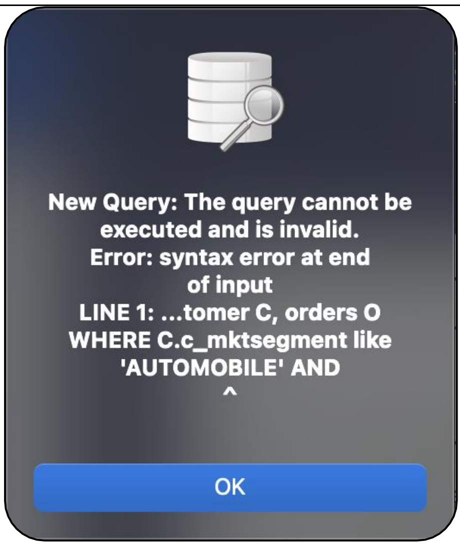
Analysis Tab



8.1.14 Invalid query

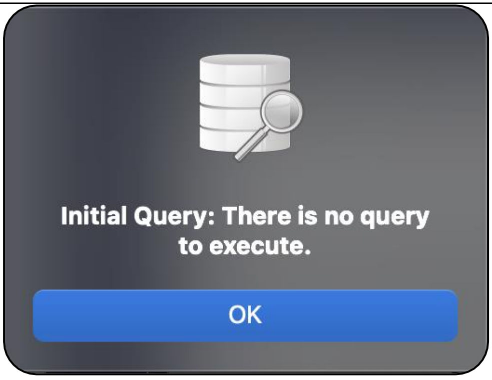
Initial Query	New Query **(Query 1)**
SELECT * FROM customer C, orders O WHERE C.c_mktsegment like 'AUTOMOBILE' AND	SELECT * FROM customer C, orders O WHERE C.c_mktsegment like 'BUILDING' and C.c_custkey = O.o_custkey

Warning Message



8.1.15 Empty query

Initial Query	New Query **(Query 1)**
	SELECT * FROM customer C, orders O WHERE C.c_mktsegment like 'BUILDING' and C.c_custkey = O.o_custkey

Warning Message


9. Source Code Installation Guide:

Here are the steps to run the project from the zip file:

1) Unzip the folder and open the codebase (CZ4031 Project 2 Code) to an IDE of your choice.

2) Install the requirements in requirements.txt file by running the command below:

    pip install -r requirements.txt

3) Download Graphviz from the following link:

    https://graphviz.org/download/

    *If you are using Windows,

    You will need to add the Graphviz bin folder to your PATH environment variable so that your system can find the Graphviz executables.

    Here's how you can do this:

    1. Open the Start menu and search for "Environment Variables". Click on the "Edit the system environment variables" option that appears.

    2. In the System Properties window that appears, click on the "Environment Variables" button.

    3. In the Environment Variables window, scroll down to the "System Variables" section and find the "Path" variable. Click on the "Edit" button.

    4. In the Edit Environment Variable window, click on the "New" button and enter the path to the Graphviz bin folder. This is typically something like

"C:\Program Files (x86)\Graphviz2.38\bin" (depending on your Graphviz version and installation location).

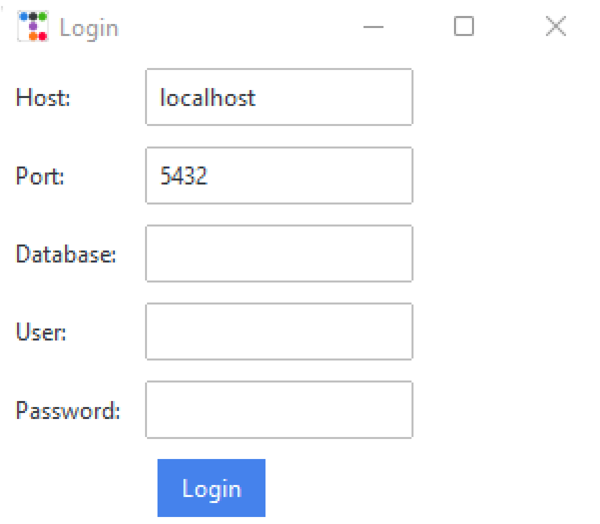
5. Click "OK" on all windows to close them and save your changes.

6. Once you have added the Graphviz bin folder to your PATH environment variable, restart your windows PC. The changes will be made, and you should be able to run Graphviz from the command line or from within your Python code.

4) Run the project.py file:

python project.py

5) Prior to querying, ensure that you log in and establish a connection to the database through this pop-up window.



6) After you've logged in successfully, go ahead and use our program!