

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

CE/CZ4031 Project 2

Submitted By:

Jess Tan Jing Yi (U1822039E)

Desmond Yeo Kok Leong (U1821023C)

Rachael Neoh Li Yii (U1822797A)

School of Computer Science and Engineering

1. Introduction

In this project, our team have developed a Python application that will assist users in understanding and visualizing the performance of a query optimizer by explaining the selected query as well as alternative queries.

The Python application includes an interactive user interface developed using PyQt5 for users to input a query. It also implements an algorithm to retrieve the query execution plan(s) based on the user's input query. Thereafter, it will return the user an explanation that describes the reason for the selected query execution plan.

In accordance with the requirements, PostgreSQL is used as the selected DataBase Management System (DBMS). The dataset and queries used are from the TPC-H database provided for this project.

2. Picasso Diagram

2.1 Modified TPC-H benchmark query

Figure 1 and 2 illustrates the modified query that is used to generate the PICASSO diagrams in this report, except for Figure 9 and 10 which will be elaborated later in Section 2.4.

In Figure 1, the selectivity variation is on the *LINEITEM* relation through the *l_extendedprice :varies* predicate.

```
Select
  c_custkey,
  c_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue
from
  customer,
  orders,
  lineitem
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate >= '1993-10-01'
  and o_orderdate < '1994-01-01'
  and l_extendedprice :varies
group by
  c_name,
  c_custkey
order by
  revenue desc
```

Figure 1 Example Query Template 1

2.2 Plan Diagram

A Plan Diagram is built by PICASSO when the user inputs a query in the PICASSO query template format. The Plan Diagram indicates which Query Execution Plan (QEP) will be executed based on the efficiency of the QEP at various values of the PICASSO Selectivity Predicate (PSP). Hence, it can represent alternative plans that will be selected based on the PSP attribute's value where the most efficient QEP will be selected. A PSP attribute is determined by the user and is specified when the keyword `:varies` is included behind the attribute in the WHERE clause of the input query.

Figure 2 illustrates the Plan Diagram for the modified query illustrated in Figure 1 earlier. As the modified query only has one PSP attribute, the Plan Diagram only includes one PSP attribute, `l_extendedprice`. Each color represents a QEP which we will refer to by using their colors, with the pattern of Plan <Colour>.

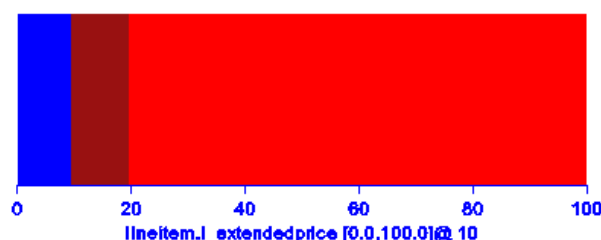


Figure 2 Plan Diagram with `l_extendedprice :varies`

The Plan Diagram illustrates in Figure 2 showcases 3 plans – Plan_Blue, Plan_Maroon, and Plan_Red. In the modified query, the PSP attribute `l_extendedprice` determines which QEP will be executed based on the attribute's values. Table 1 illustrates which QEP will be executed based on the value of the PSP attribute.

Value of <code>l_extendedprice</code>	Plan to be Executed
0 ~ 9	Plan_Blue
10 ~ 19	Plan_Maroon
20 ~ 100	Plan_Red

Table 1: Table of Plan to be Executed Based on `l_extendedprice` values

Plan_Blue will be executed when the value of `l_extendedprice` is between 0 to 9 while Plan_Maroon will be executed when the value is between 10 to 19. Plan_Red will be executed when the value is from 20 to 100. These boundary values are estimated as we are unable to expand the Plan Diagram for precise values.

Figure 3 illustrates the Plan Diagram for a query that includes 2 PSP attribute.

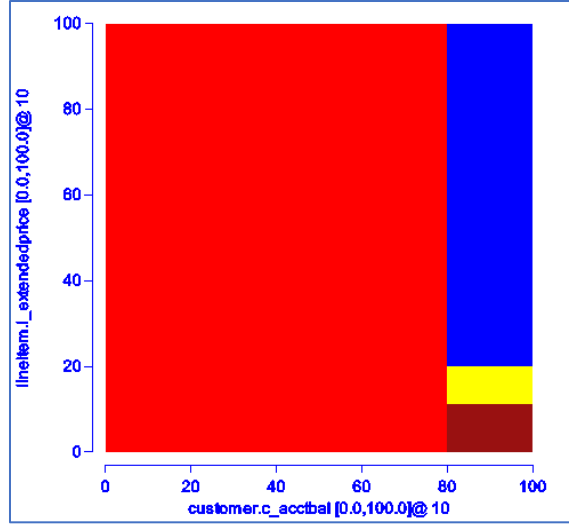


Figure 3 Plan Diagram with 2 PSP Attributes

Table 2 illustrates which QEP will be executed based on the estimated value of both PSP attributes.

Value of <i>c_acctbal</i>	Value of <i>l_extendedprice</i>	Plan to be Executed
0 ~ 80	Any value	Plan_Red
80 ~ 100	0 ~ 10	Plan_Maroon
80 ~ 100	11 ~ 19	Plan_Yellow
80 ~ 100	20 ~ 100	Plan_Blue

Table 2: Table of Plan to be Executed Based on *l_extendedprice* values – 2 PSP Attributes

2.3 Cost Diagrams – Compilation/Execution

A Cost Diagram is the visualization of cost for a given query's execution plan, given the PSP attribute of the given query. Hence, for the given query, PICASSO will generate the Estimated Cost Diagram and the Execution Cost Diagram. The Estimated Cost Diagram is also known as the Compilation Cost Diagram as the cost of execution the query is estimated.

Figure 4 illustrates the estimated cost of the query while Figure 5 illustrates the actual cost of the query.

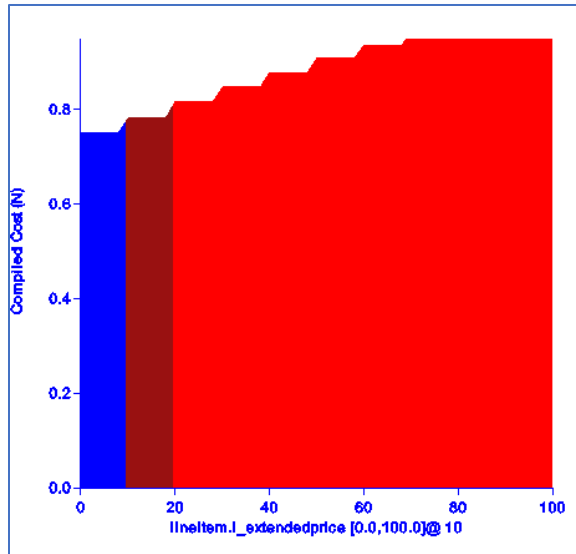


Figure 4 Estimated Cost Diagram

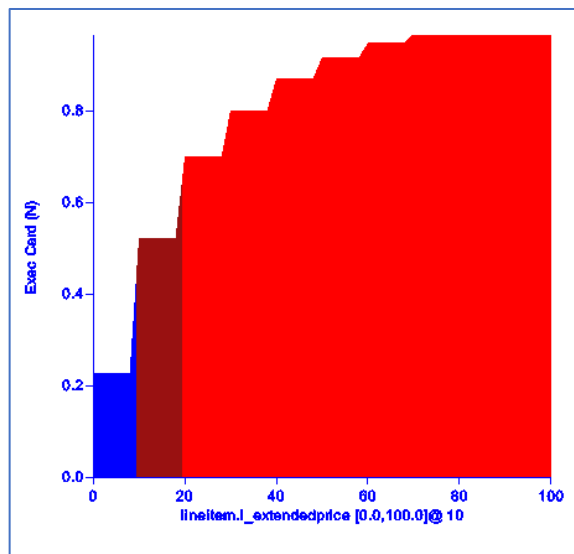


Figure 5 Execution Cost Diagram

While PostgreSQL does not do a perfect job at estimating the cost of executing the query, it provides a strong estimate of the execution cost as it is largely reflective of the actual cost of execution. This can be observed by the fact that both cost diagrams share a similar shape and values.

Figure 6 illustrates another Execution Cost Diagram of the same query.

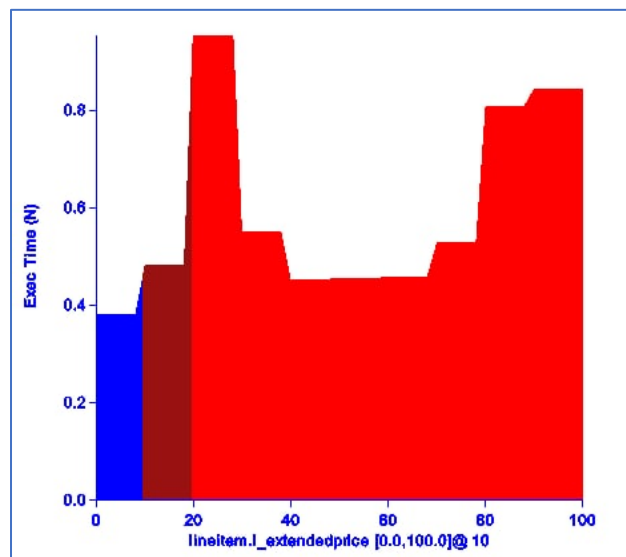


Figure 6 Volatile Execution Cost Diagram

We can observe a stark difference between the new Execution Cost Diagram and the previous Estimated Cost Diagram. This is because the execution cost is highly volatile and subjected to the availability of resources when the query execution is performed.

The key difference that caused the differences between Figure 5 and Figure 6 is the ongoing number of background task and its resource requirements. The query execution that produced Figure 5 was performed when there were no other background tasks. Figure 6 was produced when there were many other resource-intensive programs running in the background, hence limiting the number of resources available to PostgreSQL server which can in turn increases and decreases the execution time in an arbitrary pattern.

2.4 Cardinality Diagrams – Compilation/Execution

The Cardinality Diagram is a visualization of the resultant cardinality after the query execution. When the estimated cardinality diagram is similar to the execution cardinality diagram means that the DBMS can estimate the cardinality well.

Figure 7 illustrates the estimated cardinality and cost of the query while Figure 8 illustrates the actual cardinality and cost of the query.

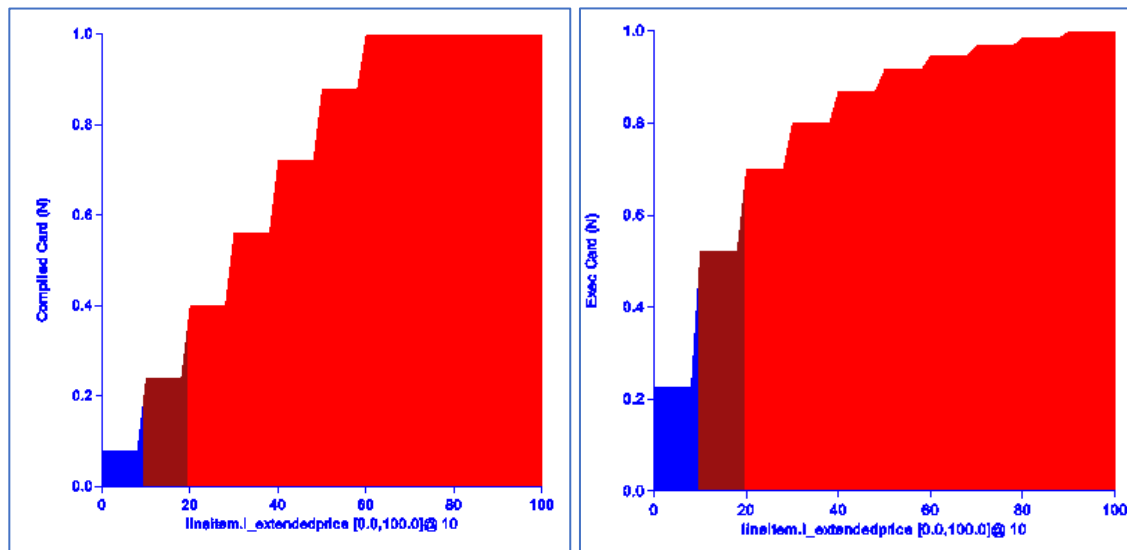


Figure 7 Estimated Cardinality Diagram

Figure 8 Execution Cardinality Diagram

As the DBMS will select a plan based on the cardinality of the table, if the estimated cardinality is close to the execution cardinality, the query processor will be able to accurately select the best QEP for each range.

The difference in cardinality plays a large role in the selection of QEP. Using an example, if the estimated cardinality of the operation is high, index scan might perform better than sequential scan. However, if the estimated cardinality of operation is low sequential scan might be much faster.

In Figure 7 and Figure 8, we find that the Estimated Cardinality Diagram accurately reflects the Execution Cardinality Diagram to a considerable amount.

However, a key point regarding Cardinality Diagrams is that while the Estimated Cardinality Diagram seems to accurately the Execution Cardinality Diagram, this is an **edge case** as it is rarely able to provide such accurate representations.

Figure 9 and Figure 10 illustrates an example of the inaccuracy of Cardinality Diagrams that will occur in usual cases.

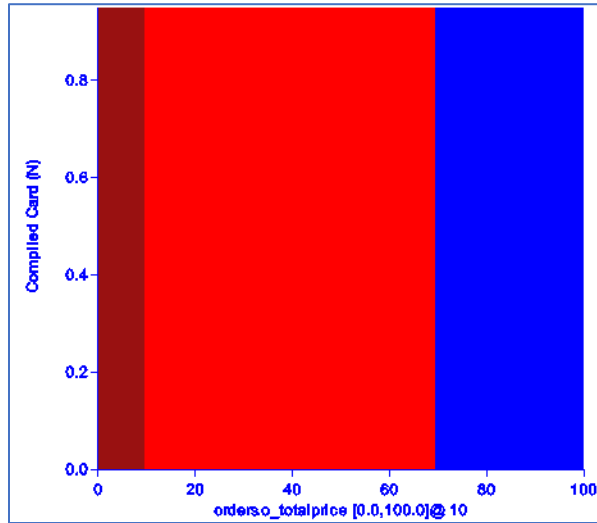


Figure 9 Estimated Cardinality Diagram

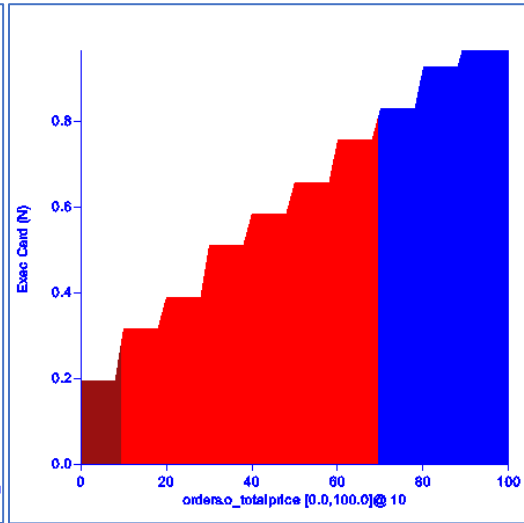


Figure 10 Execution Cardinality Diagram

Figure 11 illustrates the query used to develop the Cardinality Diagrams in Figure 9 and 10 where the selectivity variation is on the *ORDER* relation through the *o_totalprice* :varies .

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

Figure 11 Example Query Template 2

The Estimated Cardinality Diagram in Figure 9 is extremely different from the Execution Cardinality Diagram in Figure 10. This supports the a forementioned key point that the accuracy depicted in Figure 7 and Figure 8 is merely an edge case.

2.5 Description and Analysis of Performance of Query Processor

Query processors are one of the largest major components in a relational or electronic database where data is stored in tables of rows and columns. It accepts and executes SQL commands according to the chosen QEP, interacting with the Database Server to return the expected results. It is also responsible for parsing and optimizing the input query before executing these queries by using specific data access methods and database operator implementations.

As the chosen query does not need to be the best and it is sufficient to be good enough, the performance of query processor can be analyzed using PICASSO cost and cardinality diagram. Hence, we will refer to such QEP as ‘best’ as they are not actually the best but are good enough.

Comparing the execution and compilation for both the cost and cardinality diagram allows us to observe the optimizer’s modeling quality as the query processor will determine what kind of operation to performance by estimating the cost and/or cardinality of the query. If the estimation is accurate, the query processor will be able to select the ‘best’ QEP for the given query.

When analyzing for the performance of query processor, cardinality diagram can provide results that are more consistent than cost diagram as it is not affected by the system's resource utilization

In conclusion, while the query processor can provide us with various diagrams, they may be flawed and inaccurate. PICASSO also does not provide a final decision of which QEP to execute. Furthermore, DBMS are not only able to provide alternative QEP like PICASSO, but they can also select the ‘best’ QEP. Hence, the algorithm to implement our Python application does not include the use of PICASSO to retrieve the selected plan and other alternative plans.

3. Algorithm

3.1 Description

The implemented algorithm requires the user to input a query that is in the format of a SQL query template into the Python application. It will first perform validation on user inputs. If the query’s syntax is correct, the application will proceed to retrieve the QEP in JSON format from PostgreSQL by performing the EXPLAIN operation.

Our algorithm will try to retrieve all the available alternative plans. This is done by updating the database configurations parameters at each attempt to retrieve the QEP. These configuration parameters provide a crude method of influencing the query plans chosen by the query optimizer and will force the optimizer to choose a different plan.

We begin by creating a list of configurations to be applied to the query and send this query with the different configuration to PostgreSQL.

3.2 Generation of Different Configurations

The following is a list of available Planner Method Configurations:

- enable_bitmapscan
- enable_hashagg
- enable_hashjoin
- enable_indexscan
- enable_mergejoin
- enable_nestloop
- enable_seqscan
- enable_sort

The application will generate all possible combinations of these configurations by enabling or disabling a combination of each configuration parameter.

3.3 Generate Explanation

After the application have retrieved the selected plan and the available alternative plans, it will generate the explanation explaining why PostgreSQL chose the selected QEP. Most of the current query optimizers are cost-based which means that the DBMS will choose the QEP with the less cost.

Figure 12 illustrates a screenshot of the cost of each QEP retrieved from the DBMS, as well as the explanation for choosing the selected QEP.

P1	P2	P3	P4	P5	P6	P7
240,993.02	10,200,194,173.59	1,193,880.76	365,983.88	558,715.3	534,933.77	342,016.45

Plan 1 is selected because it has the least cost amongst other plans.
Low selectivity of predicate (o_totalprice > '0'::numeric) will make seq scan on **orders** faster.
Low selectivity of predicate (l_extendedprice > '0'::numeric) will make seq scan on **lineitem** faster.

Hash join has better performance when doing equality join **where** hash condition is (orders.o_orderkey = lineitem.l_orderkey)

Figure 12 Sample Result of Cost and Explanation from Application

In this case, P1 will be the selected plan as it has the lowest cost. The application also explains why P1 is the selected plan, focusing on the types of scans and joins used to explain why the selected query is better than the alternative queries.

For example, performing sequential scan on a predicate with low selectivity can result in a lower cost while performing index scan on a predicate with high selectivity can result in a lower cost. An example of explaining why certain join operations are used instead of others is how hash join is used in certain queries as it is more efficient as compared to another join operator such as nested loop.

4. Query Visualizer

Our team has designed and implemented a friendly Graphical User Interface (GUI), Query Visualizer, to enhance the user's understanding of the QEP and user experience by displaying the QEP in an interactive operator tree diagram. Explanation of why the selected plan was selected is included in the GUI, below the user's input.

Figure 13 illustrates a screenshot of the GUI when it is launched. Figure 14 illustrates a screenshot of the GUI if the user clicks on 'Query Plan' button before inserting a query. Figure 15 illustrates a screenshot of the GUI after the user has inserted a query and clicked the 'Query Plan' button. Figure 16 illustrates the settings that the user may change in order to use the Python application.

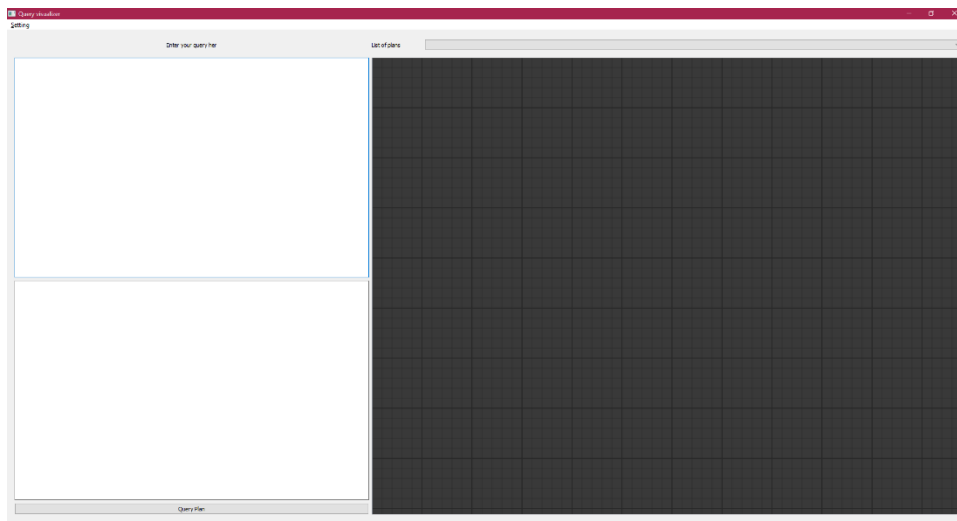


Figure 13 GUI when Python Application is Launched - No User Input

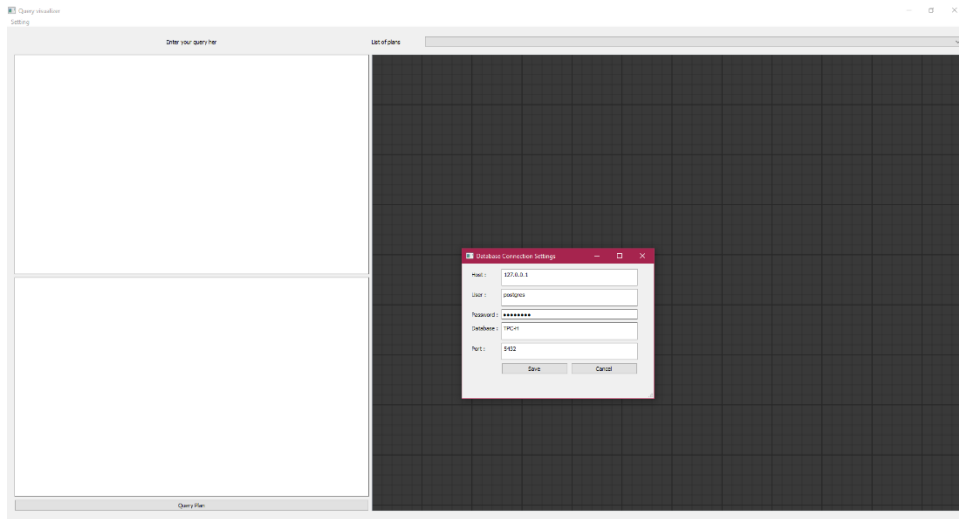
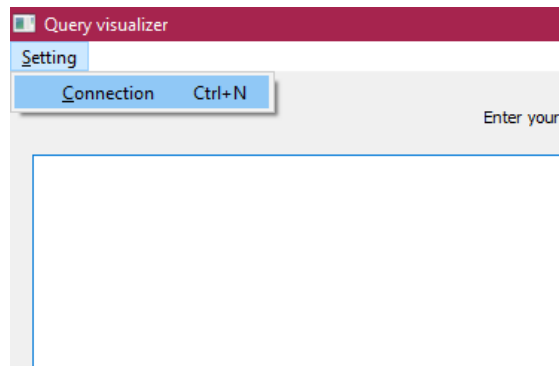


Figure 16 Query Visualizer Database Settings

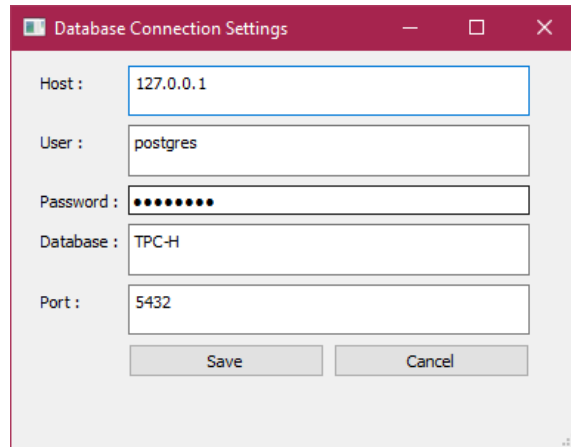
5. User Guide

This section presents an example of using the GUI of the Python application, Query Visualizer. It is important to note that the user must update the Python application with its PostgreSQL user credentials before querying the user's input. This can be done through the GUI.

1. Zipped the submission folder.
2. Run the *dsp.exe* executable file located at "Executable program\main\" to launch the Python application.
3. Set up the Python application by updating it with PostgreSQL's user credentials.
 - a. Under **SETTINGS** menu, select **CONNECTION (CTRL + N)**.

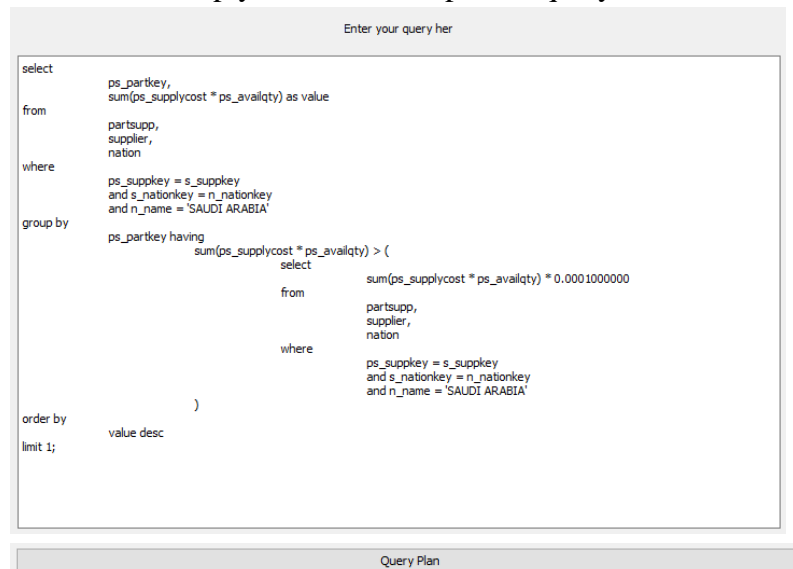


- b. Update the user's credential as necessary and save.



A screenshot of a 'Database Connection Settings' dialog box. It has a title bar with a green icon and standard window controls. The dialog contains five input fields: 'Host' with '127.0.0.1', 'User' with 'postgres', 'Password' with masked characters, 'Database' with 'TPC-H', and 'Port' with '5432'. At the bottom are 'Save' and 'Cancel' buttons.

- c. You are now ready to retrieve QEPs and the explanation for the selected QEP.
4. Enter a query that follows the SQL query template format, then click **QUERY PLAN**.
- a. Click on the empty textbox and input the query.



A screenshot of a query editor interface. At the top is a label 'Enter your query her'. Below it is a large text area containing a complex SQL query. At the bottom is a button labeled 'Query Plan'.

```
select
    ps_partkey,
    sum(ps_supplycost * ps_availqty) as value
from
    partsupp,
    supplier,
    nation
where
    ps_supplykey = s_supplykey
    and s_nationkey = n_nationkey
    and n_name = 'SAUDI ARABIA'
group by
    ps_partkey having
        sum(ps_supplycost * ps_availqty) > (
            select
                sum(ps_supplycost * ps_availqty) * 0.0001000000
            from
                partsupp,
                supplier,
                nation
            where
                ps_supplykey = s_supplykey
                and s_nationkey = n_nationkey
                and n_name = 'SAUDI ARABIA'
        )
order by
    value desc
limit 1;
```

5. Query Visualizer will be updated with an interactive operator tree and an explanation for why this plan is selected.

The screenshot shows the Query Visualizer interface. On the left, there is a text area for the SQL query:

```
select
  ps_partkey,
  sum(ps_supplycost * ps_availqty) as value
from
  partsupp,
  supplier,
  nation
where
  ps_supply = s_supplykey
  and s_nationkey = n_nationkey
  and n_name = 'SAUDI ARABIA'
group by
  ps_partkey having
    sum(ps_supplycost * ps_availqty) > (
      select
        sum(ps_supplycost * ps_availqty) * 0.000100000000
      from
        partsupp,
        supplier,
        nation
      where
        ps_supply = s_supplykey
        and s_nationkey = n_nationkey
        and n_name = 'SAUDI ARABIA'
    )
order by
  value desc
limit 1;
```

Below the query is a table with 8 columns (P1 to P8) and 1 row of data:

P1	P2	P3	P4	P5	P6	P7	P8
52,423.03	9,655,686.5	126,481.61	127,294.8	2,626,494.58	10,009,226,796.44	2,507,079.94	10,000,094,226.04

Below the table is a text box explaining why Plan 1 is selected:

Plan 1 is selected because it has the least cost amongst other plans.
 Low selectivity of predicate (n_name = 'SAUDI ARABIA')::bpchar will make seq scan on **nation** faster.
 Low selectivity of predicate (n_name = 'SAUDI ARABIA')::bpchar will make seq scan on **nation** faster.

Hash join has better performance when doing equality join **where** hash condition is (partsupp.ps_supplykey = supplier.s_supplykey)
 Hash join has better performance when doing equality join **where** hash condition is (supplier.s_nationkey = nation.n_nationkey)
 Hash join has better performance when doing equality join **where** hash condition is (partsupp.ps_supplykey = supplier.s_supplykey)
 Hash join has better performance when doing equality join **where** hash condition is (supplier.s_nationkey = nation.n_nationkey)

On the right, there is a diagram of the query plan showing various operators like Seq Scan, Hash Join, and Sort.

6. To view alternative plans, click on the drop-down menu **LIST OF PLANS**
 - a. Click on any alternative plan to view the updated interactive operator tree.

The screenshot shows the Query Visualizer interface with the 'LIST OF PLANS' dropdown menu open. The menu lists 'Selected Plan' and 'Alternative Plan1' through 'Alternative Plan9'. 'Alternative Plan7' is selected.

Below the dropdown menu, there is a diagram of the query plan for 'Alternative Plan7'. The diagram shows a different sequence of operators compared to the 'Selected Plan', including Seq Scan, Hash Join, and Sort.