NANYANG TECHNOLOGICAL UNIVERSITY

CZ4031 Database System Principles

Project 2

Group 44

Bob Lin An - U1921583E

Gareth Thong Jun Hong - U2021083G

Jin Han - U1922733A

Louis Wirja - U2023825E

Tam Kai Feng - U2022118J

Project Contributions

Name	Contributions
Bob Lin An	- interface.py - annotations.py - Report
Jin Han	- Report - annotations.py
Gareth Thong Jun Hong	- interface.py - annotations.py - Report
Tam Kai Feng	- Report
Louis Wirja	- Report

**project.py* and *preprocessing.py* are not included in contributions because they are very short.

Source code and detailed README.md file can also be found here:
https://github.com/bobbyrayyy/Query-Plan-Explainer

1. Introduction

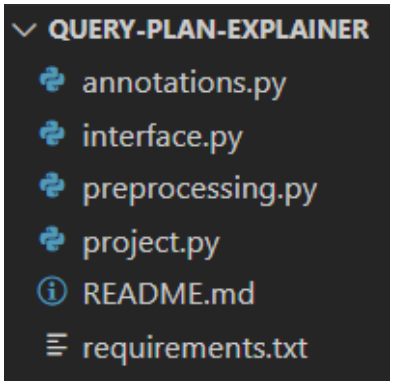
Project Overview
A Structured Query Language (SQL) is a standard language for accessing and manipulating databases. When a query is submitted to a database management system (DBMS) such as PostgreSQL, the query optimizer will evaluate some of the different, correct possible plans for executing the query and returns what it considers the best option.

In this project, we aim to:

- Show for the QEP and AQP, how different components of the query are executed.
- Identify what kind of operations are chosen for each step.
- Explain why certain operations in the QEP are chosen over the alternative operation in AQP, and annotate the original user query.
- Automatically perform the above three points.
- Ensure a good user experience and intuitive user interface.

TPC-H Dataset
The dataset we will be using is the TPC-H Benchmark H (TPC-H) dataset. TPC-H is a decision support, transaction processing and database benchmark. The TPC-H dataset is representative of a real world database, where joins can take a significant amount of time if working on relations with millions of rows. Therefore, a query optimizer plays a large part in optimizing the query to reduce the inputs to joins.

Source Code Structure

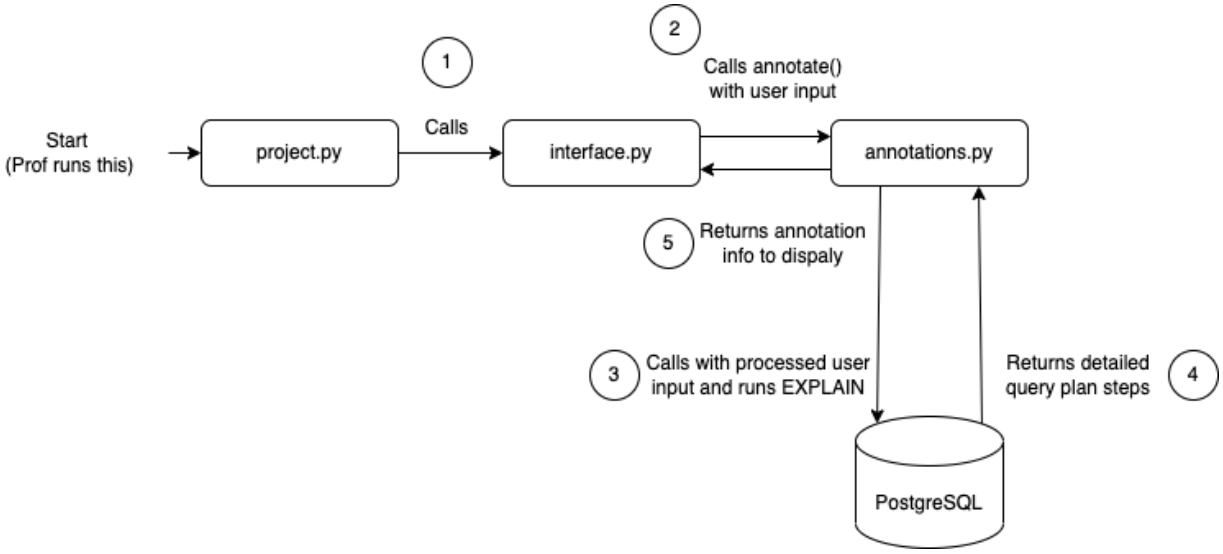


requirements.txt contains the libraries that need to be installed for this project.

The code is split into four program files:

- **interface.py** - contains the code for the GUI
- **annotations.py** - contains code for generating the annotations and reading inputs to make the algorithm work
- **project.py** - the main file that invokes all the necessary procedures from the two files

- **preprocessing.py** - an intentional unused file and this design choice will be explained in a later subsection.



2. Setup Guide

To run the program, make sure that all files are in your directory. Please ‘cd’ into the correct directory.

Before running the program, run the command below to install the required libraries.

```
pip install -r requirement.txt
```

If the above command does not work, please manually install each dependency with the following commands:

```
pip install psycopg2-binary==2.9.4
pip install PySimpleGUI==4.60.4
pip install json5==0.9.6
```

User information for the PostgreSQL database are set in the annotations.py file as follows:

- DB_NAME = "TPC-H"
- DB_USER = "postgres"
- DB_HOST = "localhost"
- DB_PORT = "5432"

3. Running the Program

To run the program, run the command:

```
python project.py
```

The user will be asked to enter your PostGres password in the terminal, which will be set as the DB_PASS for the duration of the session. Note that if the user were to enter the wrong password, the GUI window will still pop up, but queries will fail to execute when entered into the GUI.

Detailed explanation on the GUI can be found in Section 2.1.

4. Program Files

4.1 interface.py

The GUI is implemented using the PySimpleGUI library. PySimpleGUI is a library based on Tkinter that is easy to use with simple yet highly customizable features of GUI for Python. This section aims to explain in detail the components of the GUI and user flow.

The GUI window that pops up when the user has started the program is shown in Figure 1.1.

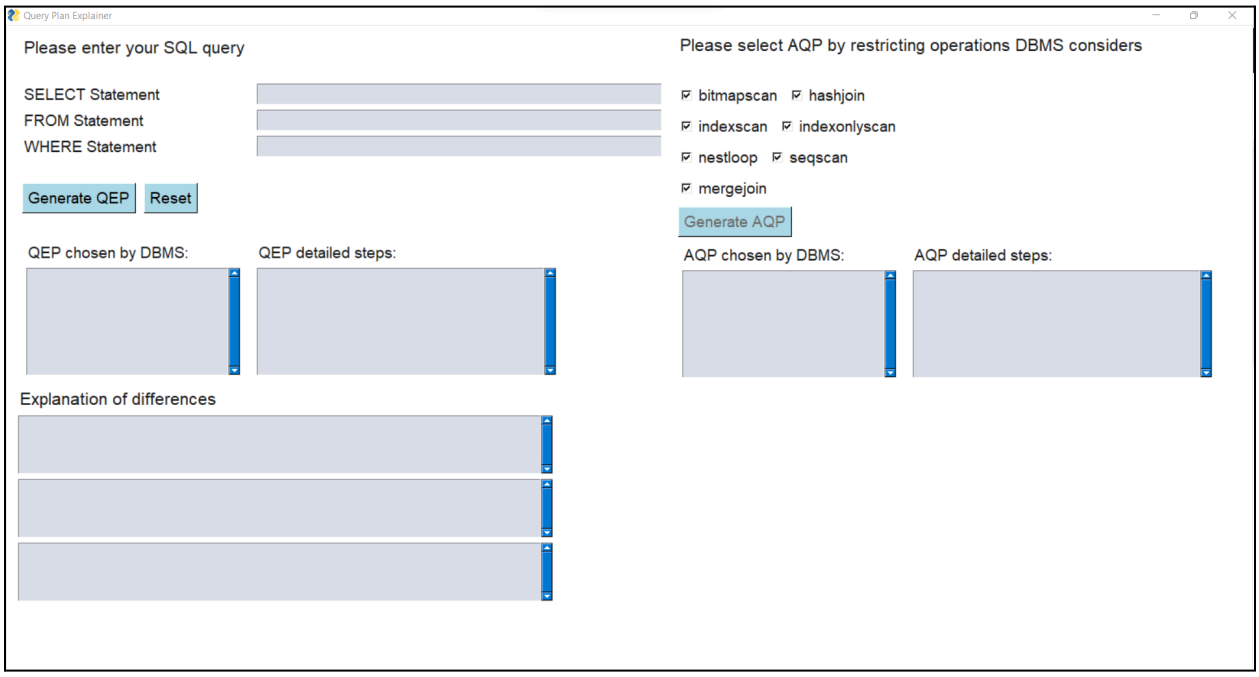


Figure 1.1: GUI window upon running the program

4.1.1 Generating QEP (Figures 1.2.1, 1.2.2 and 1.3)

The user can generate the QEP of a SQL query by inputting the query into separate statements under **Please enter your query**. If the query is valid, pressing the **'Generate QEP'** button generates the QEP of the query. An explanation on the QEP and its query plan tree are generated as well.

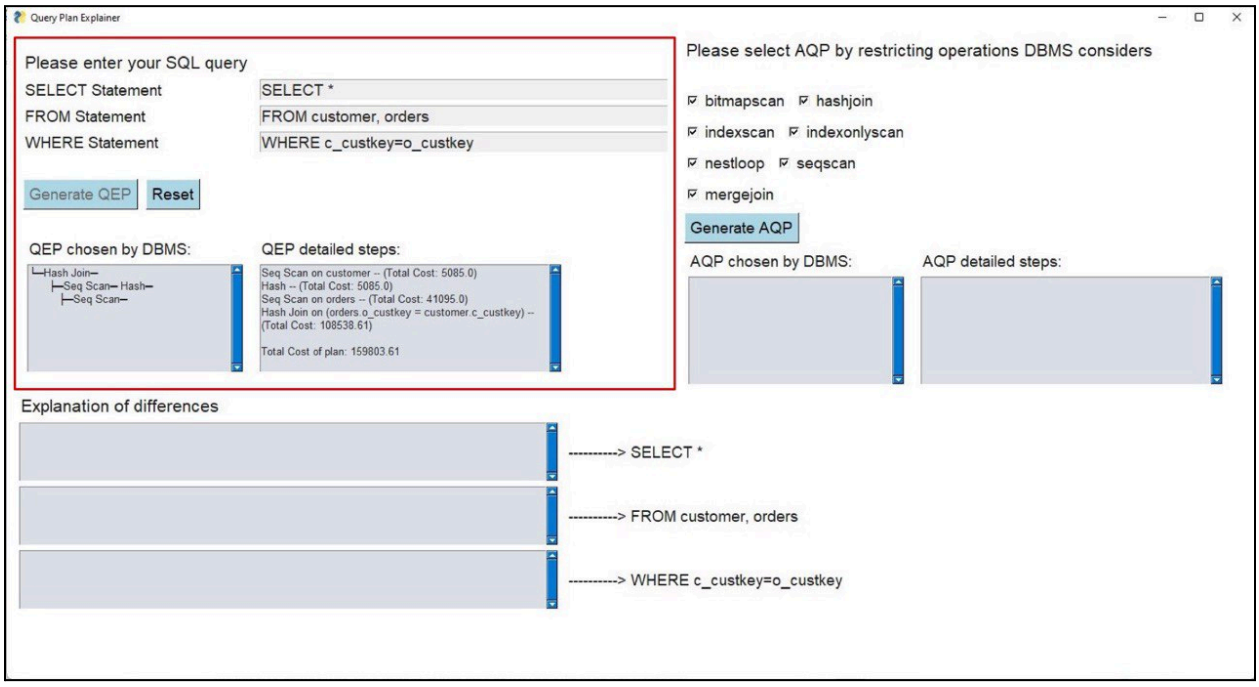


*Figure 1.2.1: QEP of the SQL query SELECT * FROM customer, orders WHERE c_custkey=o_custkey has been generated*

- **Please enter your SQL query:** The SQL query should be entered into the text fields over here. The SQL query in question is first to be separated into its SELECT, FROM and WHERE clauses, and then keyed into their respective text boxes.
- **Generate QEP:** If a valid SQL query is entered, pressing the 'Generate QEP' button will generate the query execution plan used by PostgreSQL. An example of the QEP generation of a SQL query is shown in Figure 1.2.2 below, which is a cropped version of the segment outlined by the red box in Figure 1.2.1.

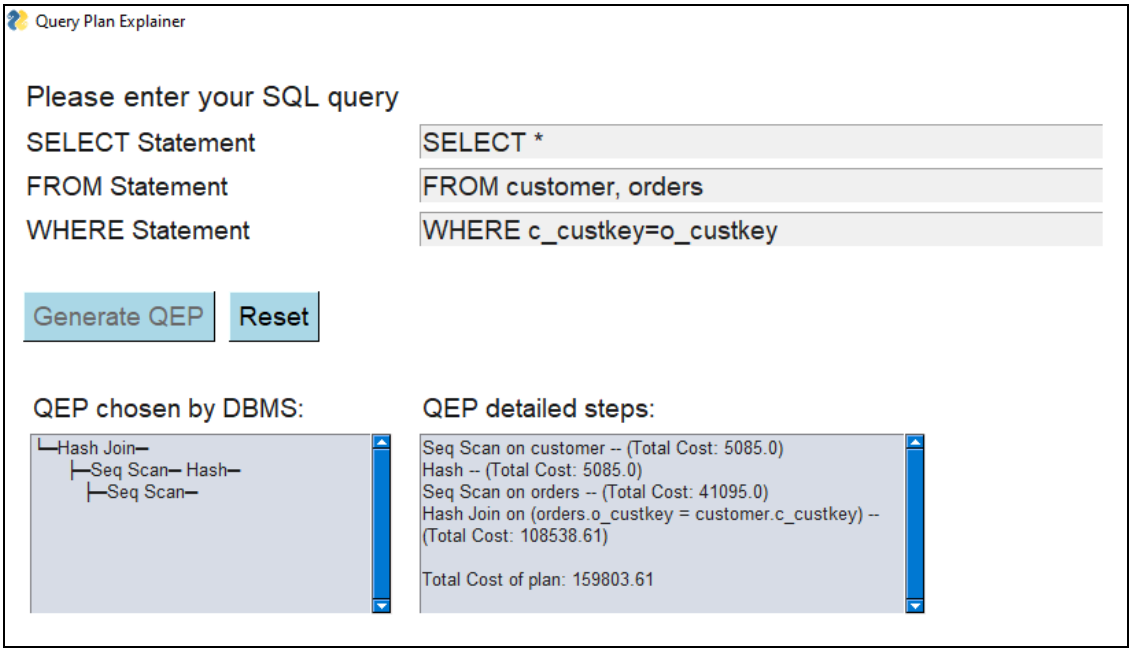


*Figure 1.2.2: QEP of the SQL query SELECT * FROM customer, orders WHERE c_custkey=o_custkey (zoomed in)*

- **QEP chosen by DBMS:** A tree denoting the sequence of operations performed on the relations is generated. The most indented operation at the bottom is performed first, while the least indented operation at the top of the tree is performed last.
- **QEP detailed steps:** Each operation is broken down into further details, such as the cost of the operation, the relation(s) on which the operation was performed, and so on.

A try-except block is also used in the code to catch all possible errors. In the event of an error caused by an invalid query, an error message will show up to inform the user. As shown in Figure 1.3, when we try to click the **'Generate QEP'** button with an empty query input, an error message will appear.

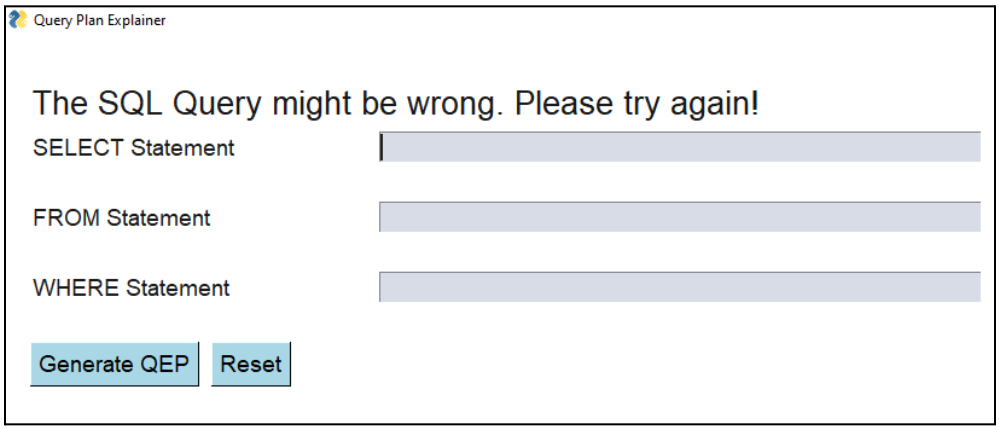


Figure 1.3: Error message for invalid query

4.1.2 Generating AQPs (Figure 1.4.1 and 1.4.2)

Since there are 11 different operations that PostGresSQL allows us to enable/disable in the EXPLAIN command, there are 2^{11} possible AQPs for each user query. Instead of generating all possible AQPs (which is too expensive and not useful to the use case), we have elected to allow the user to toggle between AQPs to compare with the QEP, to provide a robust solution.

The user selects the options that will toggle between enabling/disabling certain restrictions on operations that the DBMS considers when generating the AQP in PostgreSQL. These selections can be done under **Please select AQP by restricting operations DBMS considers** (Figure 1.4.1). An AQP can be generated by pressing the **'Generate AQP'** button (Figure 1.4.1) after selecting and deselecting a specific combination of AQP options.

When the '**Generate AQP'** button (Figure 1.4.1) is clicked, PostgreSQL will consider the best alternative plan based on these restrictions. An explanation of the AQP and the query plan tree are generated.

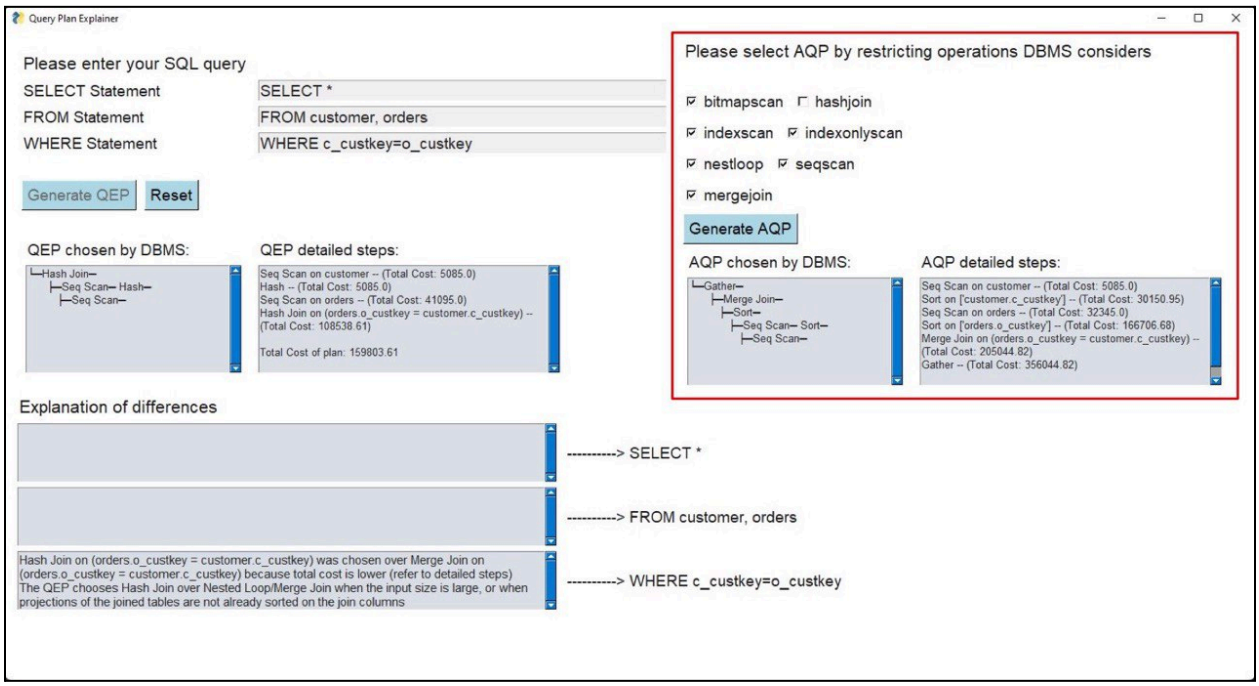


*Figure 1.4.1: AQP of the SQL query SELECT * FROM customer, orders WHERE c_custkey=o_custkey after deselecting 'hashjoin'*

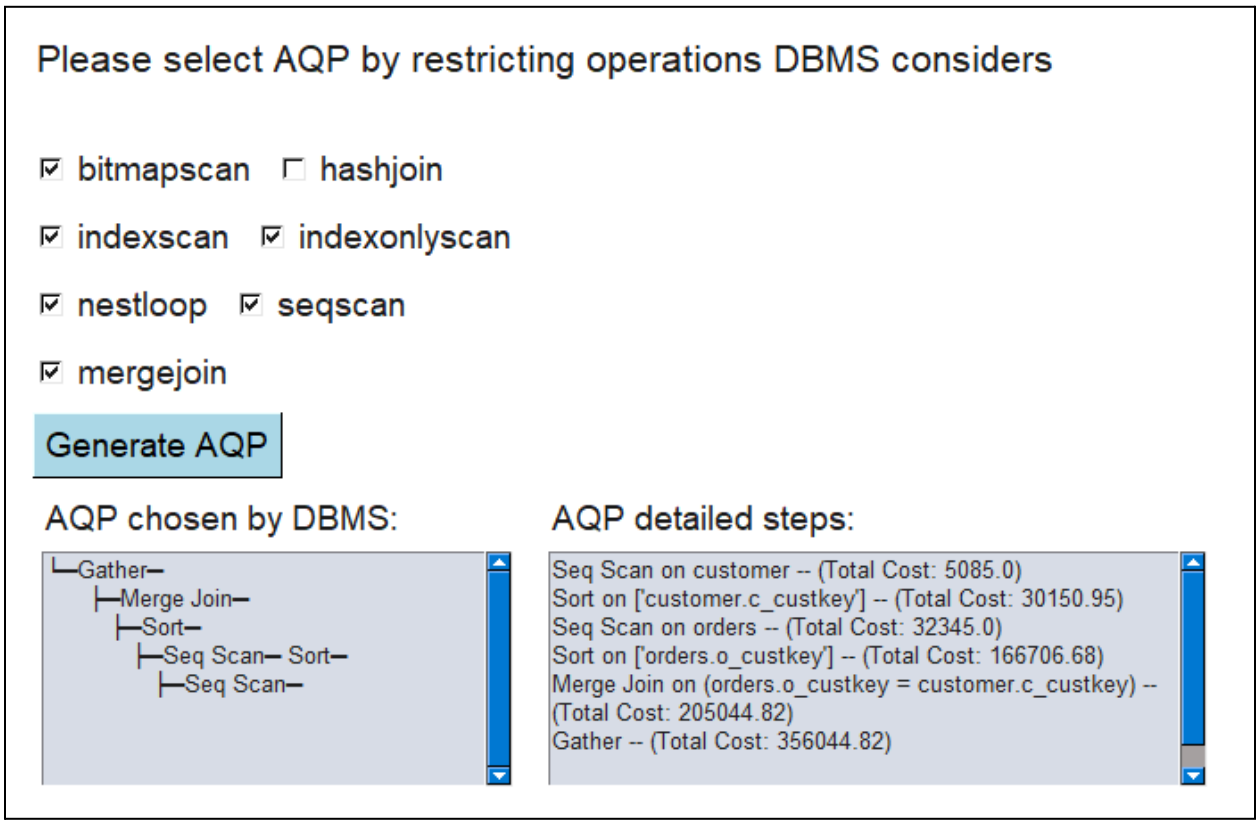


*Figure 1.4.2: AQP of the SQL query SELECT * FROM customer, orders WHERE c_custkey=o_custkey from deselecting 'hashjoin' (zoomed in)*

4.1.3 Going through an example of AQP selection

Consider a user selecting some restrictions for the AQP. In Figure 1.4.2, user deselects *hashjoin*. This information is stored temporarily by *interface.py*. When the user clicks 'Generate AQP', this selection information is passed to annotations.py and then PostgreSQL through the aqp_explain(select_text, from_text, where_text, AQP_CONFIGS_2) function.

The command **`SET enable_hashjoin TO off;`** informs PostgreSQL to not use hash joins when generating the AQP. Notice that the QEP uses hash join as the joining operator, as shown in Figure 1.2.2. Notice how the chosen AQP uses merge join instead to join the two tables *customer* and *orders*.

Explanation on the differences between the QEP and the AQP is then annotated on the SQL query on the bottom of the GUI under **Explanation of differences** (Figure 1.5.2).

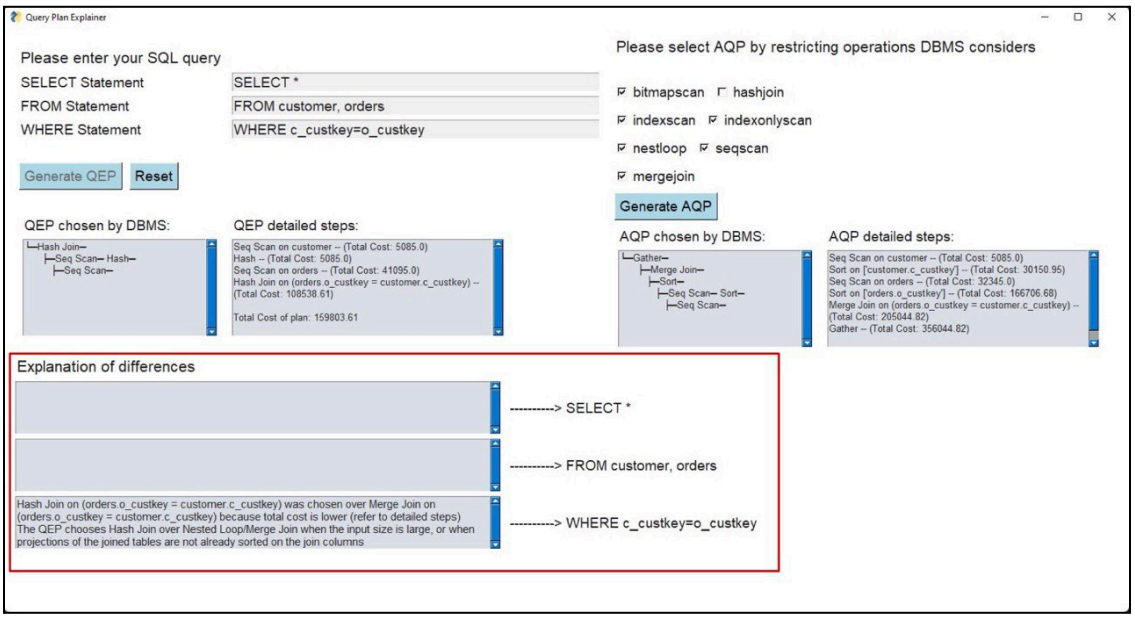


*Figure 1.5.1: Difference between QEP and AQP for the SQL query SELECT * FROM customer, orders WHERE c_custkey=o_custkey'*

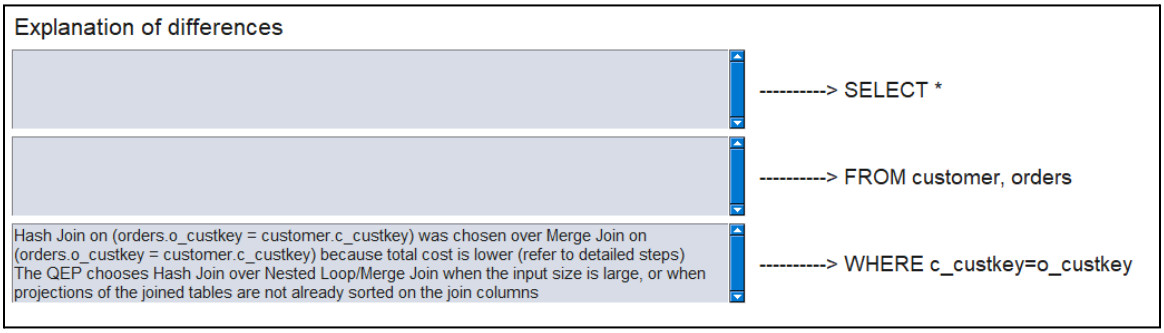


*Figure 1.5.2: Difference between QEP and AQP for the SQL query SELECT * FROM customer, orders WHERE c_custkey=o_custkey' (zoomed in)*

In Figure 1.5.2, the annotation explains why hash join is used in the QEP over merge join: hash join is the most optimal method in joining the two tables of this database due to lower cost.

The user will have to click the '**Reset'** button (Figure 1.5.1) to clear all text fields. This will allow the user to enter a new SQL query.

To illustrate with another example, refer to Figure 1.6:

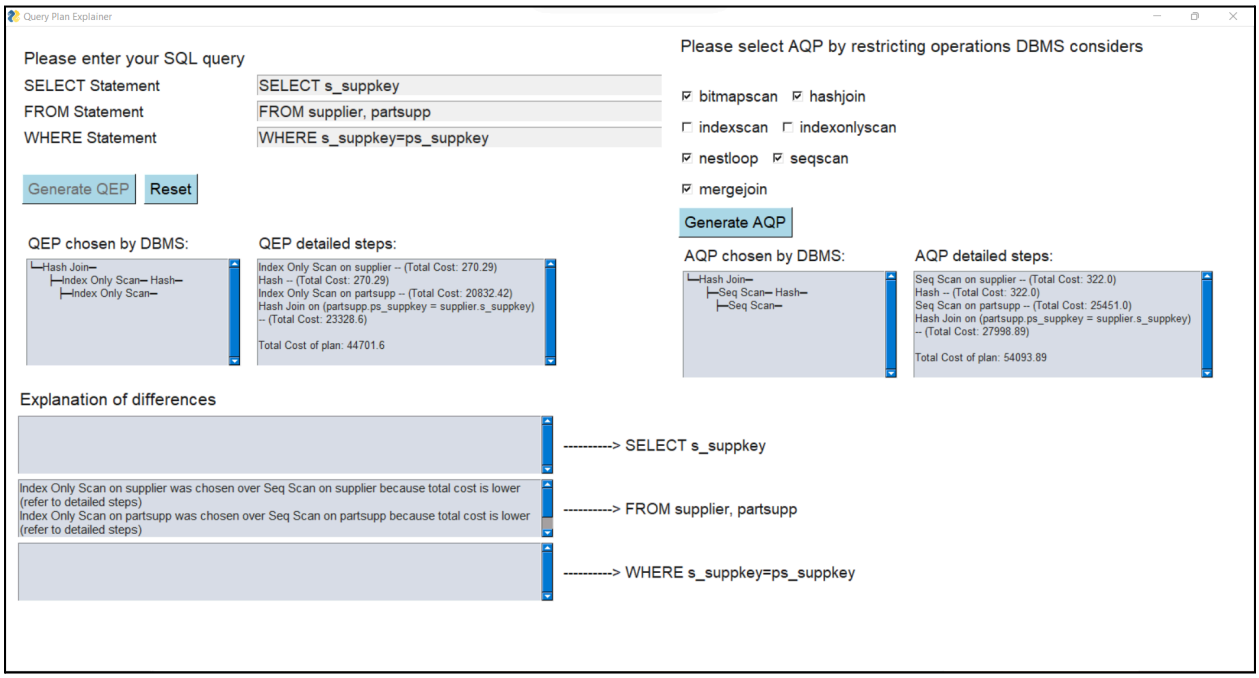


Figure 1.6: QEP and AQP for the SQL query SELECT s_suppkey FROM supplier, partsupp WHERE s_suppkey=ps_suppkey

After clicking the 'Reset' button, the user has generated another QEP and AQP for a new SQL query, with the appropriate explanations and annotations. Note that *indexscan* and *indexonlyscan* have been disabled for generation of the AQP.

These commands were passed to PostgreSQL before generation of the AQP in Figure 1.6:

```
SET enable_indexscan TO off;
SET enable_indexonlyscan TO off;
```

Note that the following commands are always passed to PostgreSQL through the explain(select_text, from_text, where_text) function before generating the QEP:

```
SET enable_bitmapscan TO on;
SET enable_hashagg TO on;
SET enable_hashjoin TO on;
SET enable_indexscan TO on;
```

```
SET enable_indexonlyscan TO on;
SET enable_material TO on;
SET enable_mergejoin TO on;
SET enable_nestloop TO on;
SET enable_seqscan TO on;
SET enable_sort TO on;
SET enable_tidscan TO on;
```

The process of obtaining the QEP and AQPs will be explained in Section 2.2, which elaborates on *annotations.py.*

4.2 annotations.py

This is where the program actually connects to the DMBS, extracts information from the results returned from the DMBS, and also generates explanations for the differences between QEP and AQP for annotation.

The library *psycopg2* allows Python to pass queries to the PostgreSQL database and receive the result of those queries.

When the 'Generate QEP' button is pressed, the program will read inputs from the SELECT, FROM and WHERE textboxes and pass it to the function *explain(select_text, from_text, where_text)*. *annotations.py* will return the explanation, total cost and tree of the QEP generated to interface.py.

When the 'Generate AQP' button is pressed, the program will read inputs from the SELECT, FROM and WHERE textboxes and the set of chosen operators, and pass them to the *aqp_explain(select_text, from_text, where_text, AQP_CONFIGS_2)* function. Similarly, *annotations.py* will return the explanation, total cost and tree of the AQP generated to *interface.py.*

Explanation key functions and algorithms

As explained in Section 4.1, settings like *enable_bitmapscan* are first set appropriately depending on whether the user wishes to generate a QEP or AQP.

The following command is then passed to PostgreSQL to generate the query plan:

```
EXPLAIN (ANALYSE, COSTS true, FORMAT json) <query>
```

PostgreSQL will then generate the query plan, and return the output in a JSON format. The components of the command are:

- **EXPLAIN:** This is for displaying the execution plan that the PostgreSQL planner generates for the supplied query. The execution plan shows how the tables referenced by the statement will be scanned and how operations are carried out. The most critical part of the plan is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement.
- **ANALYSE:** The actual run time statistics are added to the output, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned.
- **FORMAT json:** To return the output in JSON format.

Using this command, PostgreSQL returns the QEP. To get AQPs, the user-configured parameters that enable/disable certain operations are formatted as commands to execute on

the DBMS, telling the DBMS what operations to exclude. Only then is the EXPLAIN command run.

One example is the inclusion of hash joins in the generation of query plans. If the setting to include hash joins is turned off, the optimizer will not include that operation and find the next best query plan that does not involve any hash joins.

Information is first extracted from the JSON output returned by PostgreSQL through a series of iterations and recursions that unpack the data. The functions involved in this process are *strip_unneeded_data* and *process_plan*.

strip_unneeded_data removes unneeded data from the JSON output and *process_plan* function recursively unpacks the data and extracts key information like Node Type, Total Cost, and which relation(s) the operation is performed on. We also build our plan execution tree during this process. Finally, the operations information, total cost, and tree structure are returned to *interface.py.*

In *interface.py,* the returned operations information for both the QEP and AQP are passed into the find_difference function. This function finds the specific differences between the QEP and AQP, generates an explanation for why DBMS chose the operation in the QEP over the one in AQP, and formats them for display on the GUI interface.

We define unique operations as operations that appear only within the QEP and not within the AQP, or operations that appear only with the AQP and not the QEP. For each unique operation that appears in the QEP, we then search through the unique AQP operations to identify the AQP operation that replaced the QEP operation due to the AQP generation settings which were set by the user in the GUI (as explained in interface.py). After identifying the replacement operation, the *find_difference* function then generates an explanation for why the DBMS chose that particular QEP operation as opposed to the replacement AQP operation, and tags it to the appropriate line of the user SQL query for annotation.

In the *explanation* function, more theoretical explanations for why certain QEP operations were chosen over AQP operations are generated. Then, they are tagged to the appropriate line of the user SQL query for annotation.

With reference to the PostgreSQL documentation [9], it is noted that certain operations (i.e. Seq Scan and Nested Loop) cannot be suppressed entirely for certain queries but will only discourage the planner from using one if other methods are available.

For a better understanding of the algorithms, please refer to the code itself, which has clear and detailed comments.

4.3 preprocessing.py

We intentionally left the preprocessing.py file empty because all the preprocessing of user input is done by the GUI's separate input text boxes. We feel that a designed-in solution to handle user inputs elevates the user experience and forces users to enter input in the correct format.

The user's SQL query is passed directly to the *explain* or *aqp_explain* functions in *annotations.py*. *preprocessing.py* file is not used as there is no transformation of the user input.

4.4 project.py

This is the main file that the user will need to run. It will first prompt the user for the password to their local PostgreSQL database using the function *set_password* from *interface.py*. Then, it will invoke the *display* function from *interface.py,* which will display the GUI for the user to enter their queries and to display the results.

5. Limitations

5.1 Plain GUI

PySimpleGUI is a great library for creating a bare-bones GUI on Python. The GUI we have created is simple and fulfills all functionality that we believe will serve the user best. That said, given more time and Python frontend development expertise, our group definitely thinks a more advanced and visually-appealing GUI library can be chosen instead to create a more modern and aesthetic GUI. Dynamic window sizing according to machine specifications and dynamic margins were a feature that was sorely missed during our development of the GUI on PySimpleGUI.

5.2 Handling of long and complex nested queries

Our group has managed to come up with accurate and robust annotations and explanations for differences between the QEP and the user-chosen AQP. For long and complex queries, the explanation of differences and content of annotations are still accurate and robust. However, the current algorithm might mislabel some annotations, and is unable to pinpoint the exact word/phrase in the user query that the generated annotation should point to. This is partially due to the limitations of PySimpleGUI tool. As previously mentioned, with more time and expertise, our group would explore libraries such as schemdraw to generate arrows and lines that can point our annotations to specific words in the user query.

6. Conclusion

As mentioned, the goals of the project we set out to accomplish were:

- Show for the QEP and AQP, how different components of the query are executed.
- Identify what kind of operations are chosen for each step.
- Explain why certain operations in the QEP are chosen over the alternative operation in AQP, and annotate the original user query.
- Automatically perform the above three points.
- Ensure a good user experience and intuitive user interface.

We designed and implemented a program that automatically retrieves the QEP and the user-selected AQPs of an input SQL query from PostgreSQL.

A detailed step-by-step plan is generated for both QEP and AQP, detailing the operation, the relations affected, and the cost. Along with this, a query plan tree is generated for better visualization.

We were also able to generate explanations on why the DBMS chose certain operators in the QEP) over its alternatives in the AQPs, and annotate the user's query on the GUI.

Finally, we designed and implemented a user-friendly and clean GUI that allows users to choose different AQPs to compare against the DBMS QEP, by toggling on/off certain operators that DBMS can use. Information is displayed in a clear and structured way. User flow is intuitive and simple.

7. References

1. “Learn to code for free,” *Programiz*. [Online]. Available: https://www.programiz.com/. [Accessed: 02-Nov-2022].
2. “Python guis for humans,” *PySimpleGUI*. [Online]. Available: https://www.pysimplegui.org/en/latest/. [Accessed: 09-Nov-2022].
3. “PostgreSQL 15.1 documentation,” *PostgreSQL Documentation*, 10-Nov-2022. [Online]. Available: https://www.postgresql.org/docs/current/index.html. [Accessed: 29-Oct-2022].
4. Real Python, “Pysimplegui: The simple way to create a GUI with python,” *Real Python*, 01-Apr-2022. [Online]. Available: https://realpython.com/pysimplegui-python/. [Accessed: 12-Nov-2022].
5. “Introduction to pysimplegui,” *GeeksforGeeks*, 10-May-2020. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-pysimplegui/. [Accessed: 02-Nov-2022].
6. *Creating 10 Apps in Python [with PySimpleGui]*. YouTube, 2022. [Accessed: 10-Nov-2022].
7. A. Zahir, “Creating user interfaces in python using pysimplegui,” *CodeProject*, 27-Oct-2021. [Online]. Available: https://www.codeproject.com/Articles/5315989/Creating-User-Interfaces-in-Python-using-PySimpleG. [Accessed: 01-Nov-2022].
8. Real Python, “OrderedDict vs DICT in python: The right tool for the job,” *Real Python*, 13-Aug-2022. [Online]. Available: https://realpython.com/python-ordereddict/. [Accessed: 29-Oct-2022].
9. “PostgreSQL 18.7 documentation”, PostgreSQL Documentation, 10-Nov-2022. [Online]. Available:https://www.postgresql.org/docs/9.2/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS
10. *TPC Download current specs/source*. [Online]. Available: https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp. [Accessed: 13-Nov-2022].