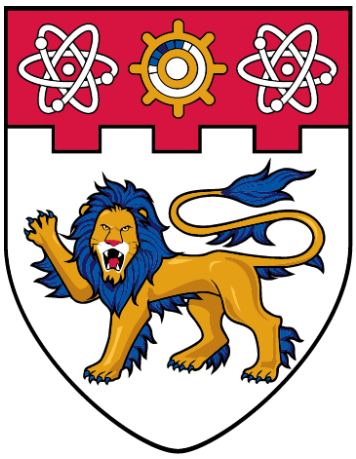


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE/CZ4031
Database Systems Principles
Project 2 Report

Group Members:

Name	Matriculation Number
Goh Hong Xiang Bryan	U1920609E
Lee Cheng Han	U1920206L
Chong Jing Hong	U1922300B
Terry Joel Ee Wen Jie	U1922131D

Table of contents

1. File Structure

The four python files below are as required:

- `project.py` is the main file that invokes all necessary procedures from the next three files, the file that handles the main flow and logic by calling the other three python files
- `interface.py` handles the main logic of how the GUI is displayed and invoked
- `preprocessing.py` handles the main login for reading inputs and connecting to the Postgres server as well as any preprocessing necessary to make our algorithm work
- `annotations.py` handles the main logic of how the query plan is displayed be it diagram format or text format

2. Libraries Used

The libraries used to make this GUI application are:

- `psycopg2` is a library that is used as a PostgresSQL database adapter for Python. More information can be found here [1].
- `PySimpleGUI` is a library that allows for simple and easy GUI development. More information can be found here [2].
- `Pmw` is a toolkit for building high-level compound widgets, or megawidgets, constructed using other widgets as component parts. More information can be found here [3].

In order to run the application, users must first ensure that the libraries stated above are installed on their systems. Users can do so by executing the following code in the command line:

```
pip install psycopg2
pip install PySimpleGUI
pip install Pmw
```

To run the project, navigate to the ‘Program’ folder and execute the following command in the command line:

```
python project.py
```

3. Algorithm

3.1. PostgreSQL EXPLAIN (ANALYSE, FORMAT JSON)

To generate the query execution plan for a query, we made use of PostgreSQL's own function. PostgreSQL devises a query plan on its own for each query it receives. When this function is executed, PostgreSQL's planner will generate an execution plan and run that plan. The plan that is executed will be output in json format, containing information such as what scans are used, what join algorithms are used etc. An example of a query execution plan output can be seen in Figure 1:

Figure 1: Query Plan by PostgreSQL

We are using a bottom-up approach so the first node to be executed will be the innermost node found in the query plan.

3.2. Operator Class

We defined a class called `Operator`. The attributes of the class includes:

- `x1, x2, y1, y2, center` - these attributes are for displaying the query execution plan as a table on the GUI, mainly the coordinates of where the table will be displayed on the GUI
- `operation` - this attribute stores the name of the operation
- `information` - this attribute stores information such as the execution time of the operation, which relation is being operated on etc.
- `children` - stores an array of child operations

The class also has a function `add_children(self, child)` to append children operators to the object.

3.3. build_plan(curr_op, json_plan)

This function parses the json output from PostgreSQL's `EXPLAIN (ANALYSE, FORMAT JSON)` output to return an array containing Operator objects corresponding to the json output.

3.4. get_current_operator_info(operator)

This function contains if-else cases for the `Node Type` key of the json output. It extracts relevant information for each type of operation. We based the if-else cases from this website [1] containing the list of possible node types.

3.5. draw(query_plan, query)

This is the main function that draws the query execution plan on the GUI. It takes the array output from the build_plan function as a parameter to draw the plan.

4. GUI

4.1. Instructions

To use the GUI, the user first has to connect to his local PostgreSQL server. To do so, simply enter the following information:

- Host name (localhost)
- Port (5432)
- Database name
- Username (postgres)
- Password

Then, the user has to input a query in the ‘Enter Query’ field. Finally, click on the submit button to generate the query execution plan. Figure 2 shows an example of the inputs.

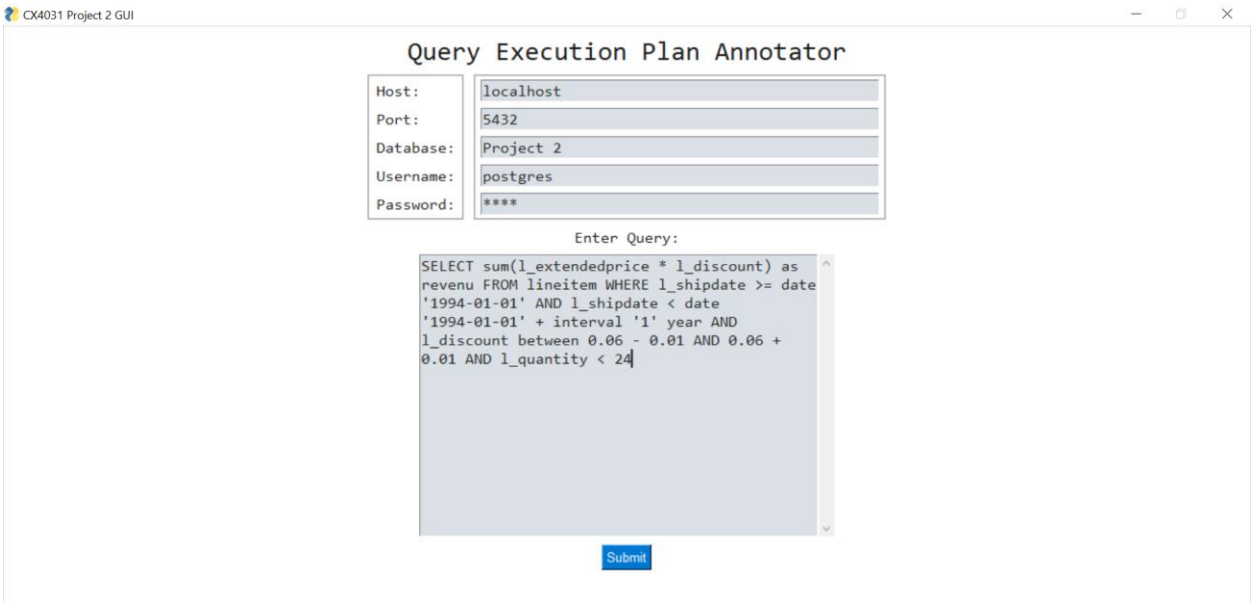


Figure 2: GUI input example

After submitting the query, a new window, containing 3 sections, will pop up. The top section contains the visual representation of the query execution plan. When users mouse-over the rectangles, a pop-up containing the annotations will appear. The bottom-left section contains the original query that the users submitted while the bottom-right query section contains the full json output. There is also a button at the top-left to close the GUI. Figure 3 illustrates the aforementioned.

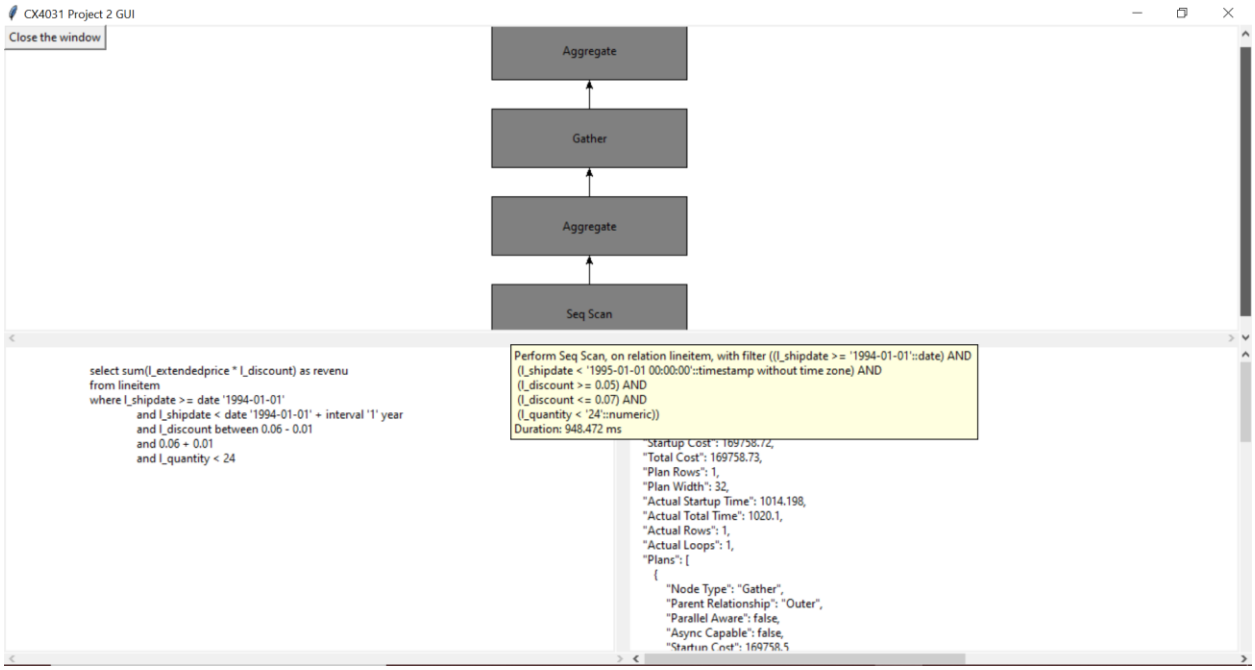


Figure 3: GUI output example

4.2. Tests

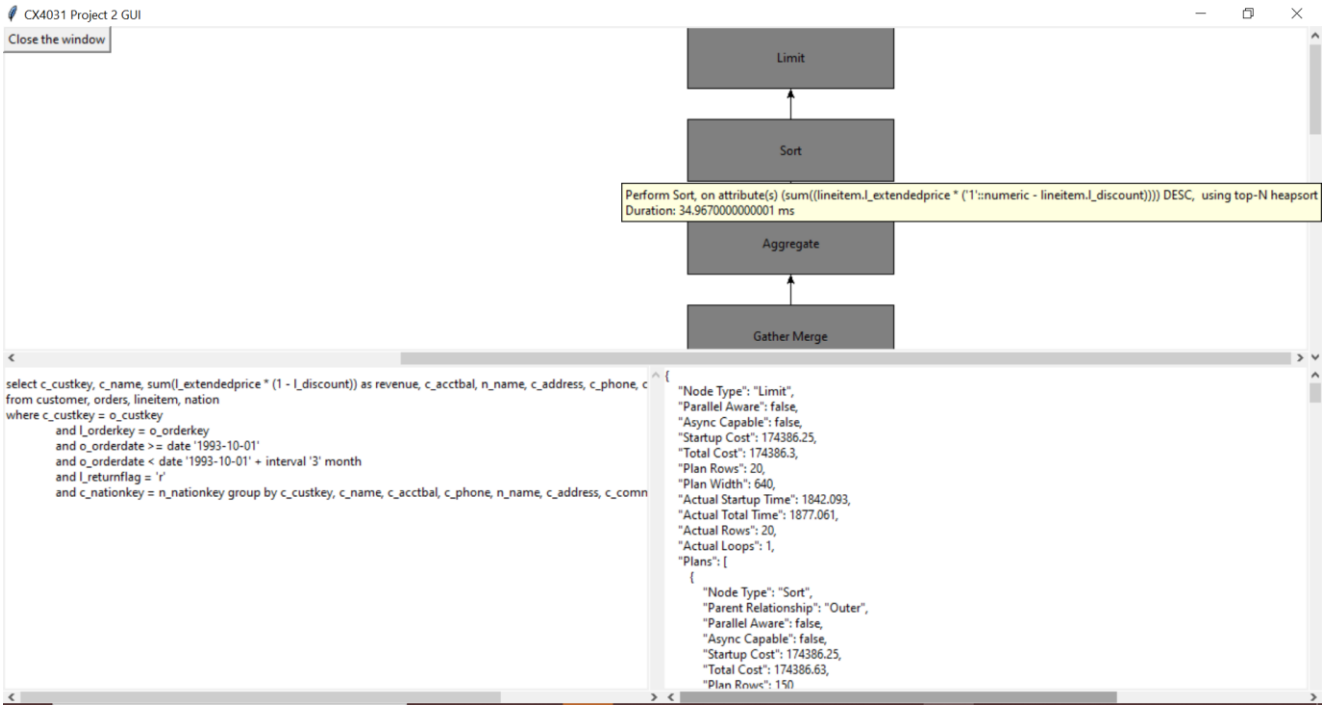
We conducted a few tests with queries found on this website to ensure that our GUI works for a range of different queries. These queries can also be found in our README file. We have picked queries 5 to 8 from the README to present in this report as they are more complex than the other queries.

4.2.1. Query 5

```
SELECT c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as
revenue, c_acctbal, n_name, c_address, c_phone, c_comment

FROM customer, orders, lineitem, nation

WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND
o_orderdate >= date '1993-10-01' AND o_orderdate < date '1993-10-01' +
interval '3' month AND l_returnflag = 'R' AND c_nationkey =
n_nationkey GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name,
c_address, c_comment ORDER BY revenue desc LIMIT 20
```

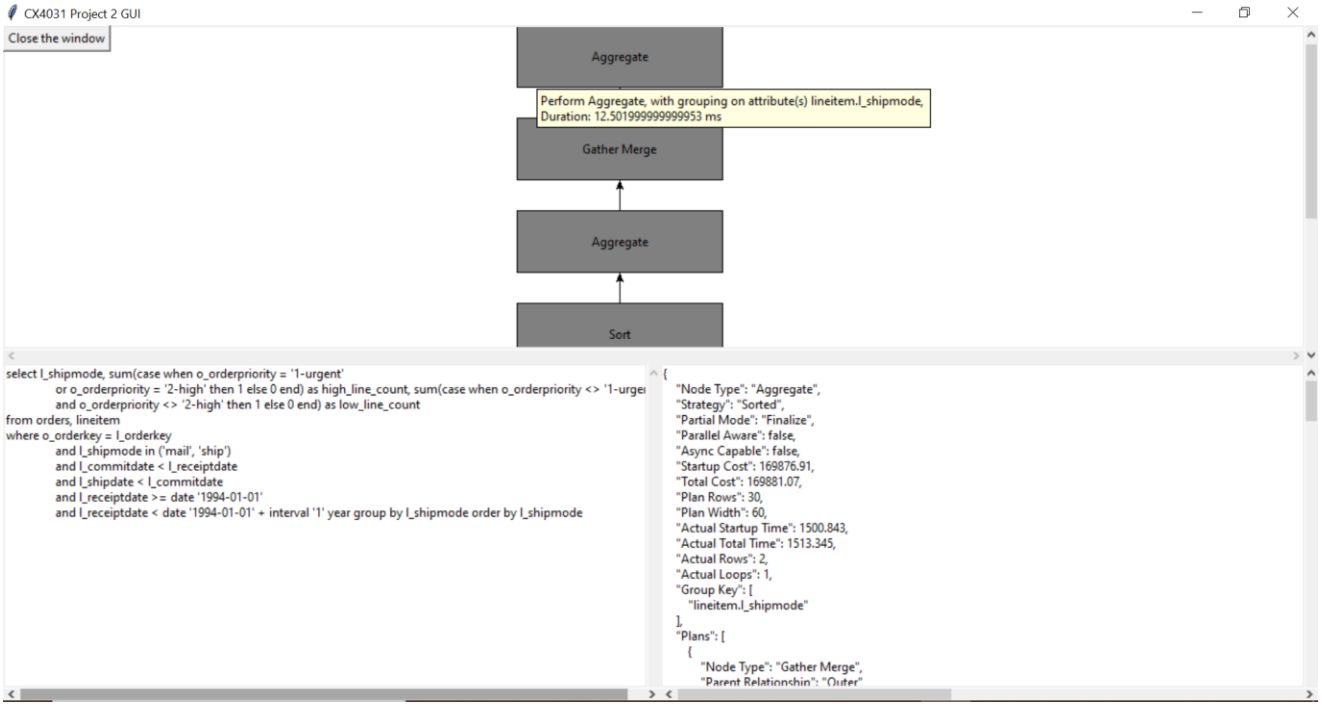


4.2.2. Query 6

```
SELECT l_shipmode, sum(case when o_orderpriority = '1-URGENT' OR
o_orderpriority = '2-HIGH' then 1 else 0 end) as high_line_count,
sum(case when o_orderpriority <> '1-URGENT' AND o_orderpriority <> '2-
HIGH' then 1 else 0 end) AS low_line_count

FROM orders, lineitem

WHERE o_orderkey = l_orderkey AND l_shipmode in ('MAIL', 'SHIP') AND
l_commitdate < l_receiptdate AND l_shipdate < l_commitdate AND
l_receiptdate >= date '1994-01-01' AND l_receiptdate < date '1994-01-
01' + interval '1' year GROUP BY l_shipmode ORDER BY l_shipmode
```

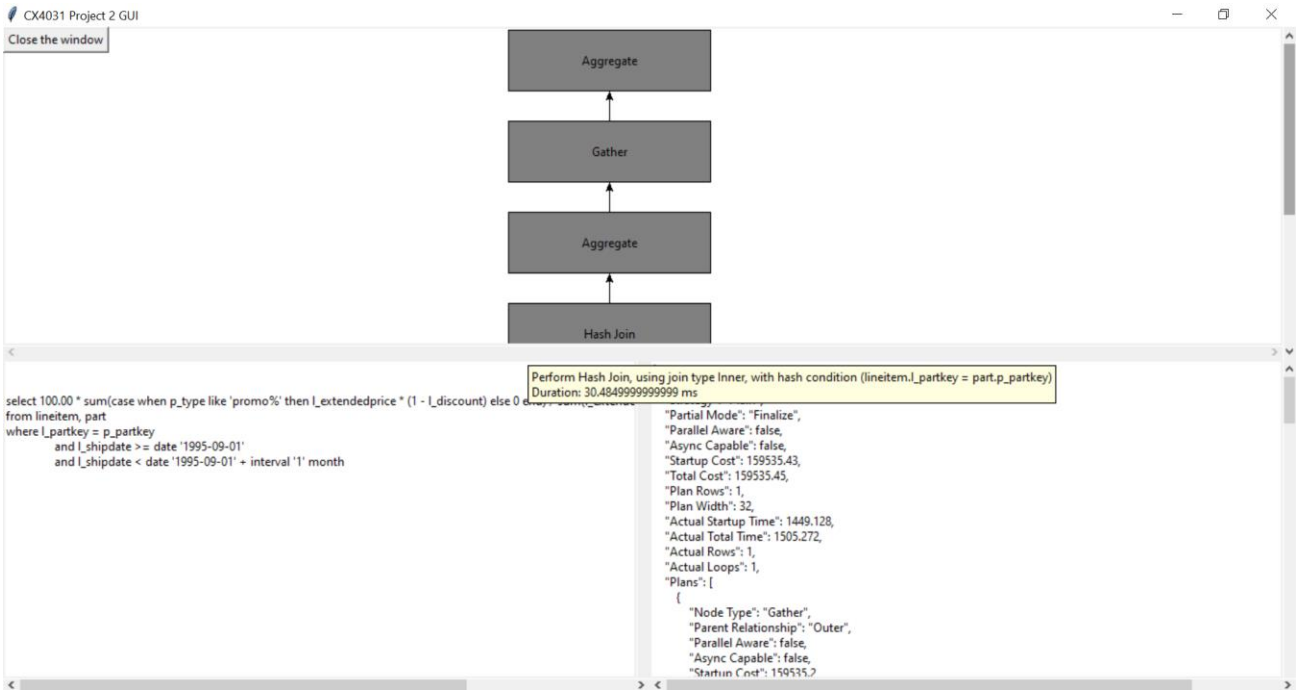


4.2.3. Query 7

```
SELECT 100.00 * sum(case when p_type like 'PROMO%' then
l_extendedprice * (1 - l_discount) else 0 end) / sum(l_extendedprice *
(1 - l_discount)) as promo_revenue

FROM lineitem, part

WHERE l_partkey = p_partkey AND l_shipdate >= date '1995-09-01' AND
l_shipdate < date '1995-09-01' + interval '1' month
```

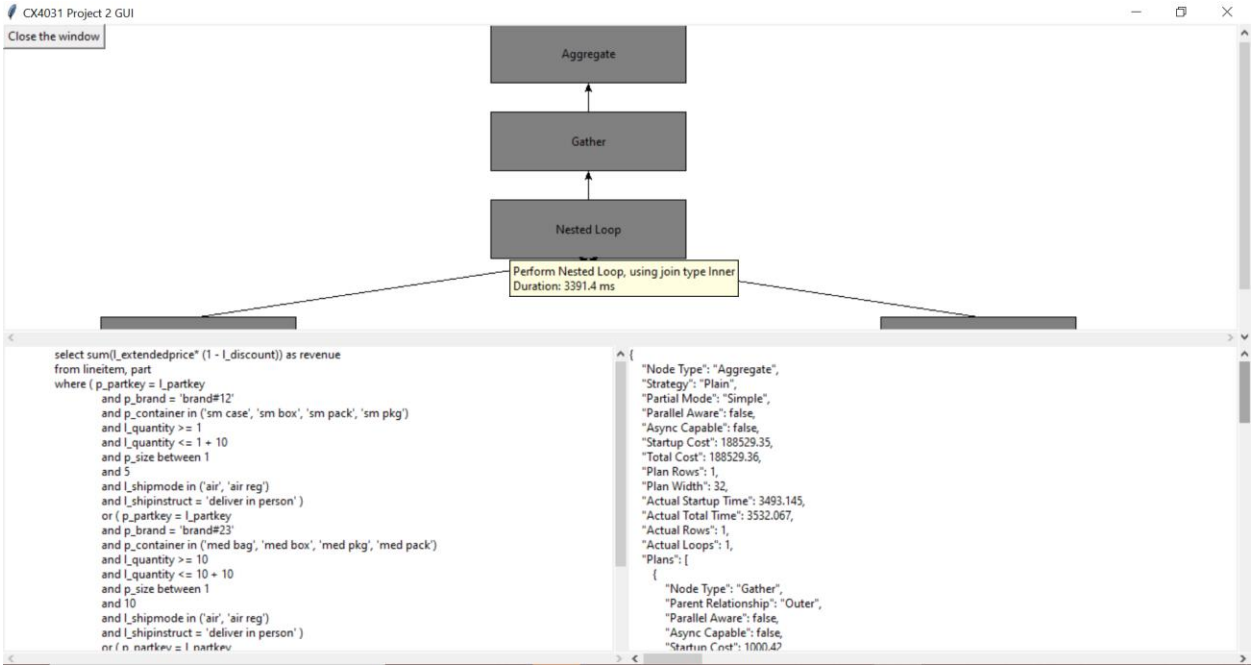


4.2.4. Query 8

```
SELECT sum(l_extendedprice* (1 - l_discount)) as revenue

FROM lineitem, part

WHERE ( p_partkey = l_partkey AND p_brand = 'Brand#12' AND p_container
in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG') AND l_quantity >= 1 AND
l_quantity <= 1 + 10 AND p_size between 1 AND 5 AND l_shipmode in
('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN PERSON' ) OR (
p_partkey = l_partkey AND p_brand = 'Brand#23' AND p_container in
('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK') AND l_quantity >= 10 AND
l_quantity <= 10 + 10 AND p_size between 1 AND 10 AND l_shipmode in
('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN PERSON' ) OR (
p_partkey = l_partkey AND p_brand = 'Brand#34' AND p_container in ('LG
CASE', 'LG BOX', 'LG PACK', 'LG PKG') AND l_quantity >= 20 AND
l_quantity <= 20 + 10 AND p_size between 1 AND 15 AND l_shipmode in
('AIR', 'AIR REG') AND l_shipinstruct = 'DELIVER IN PERSON' )
```



5. Limitations

Many open source apps do not support PostgreSQL.The installation must be homogeneous across all supported operating systems. It is an open source database application not owned by one particular organization. Therefore, it does not come with a warranty and has no liability or indemnity protection. This also causes compatibility issues with some issues due to the user-friendly interface.

It is also slower compared to other databases like MySQL. Reads were generally slower as a lot of times we have a query that is running slow and out of a sudden there is a performance degradation in the database environment. When finding a query, Postgres due to its relational database structure has to begin with the first row and then read through the entire table to find its relevant data. Hence, it performs slower when there is a large amount of data stored in the rows and columns of a table containing many fields of additional information to compare.

Furthermore, the GUI has only been tested on 8 queries. Hence, more extensive tests on queries with ranging complexities are required to further build upon the if-else cases mentioned in section 3.4.

6. Contributions

Name	Contributions
Goh Hong Xiang Bryan	Coding, report writing
Lee Cheng Han	Coding, report writing
Chong Jing Hong	Coding, report writing
Terry Joel Ee Wen Jie	Coding, report writing

7. References

[1] F. Di Gregorio, "psycopg2 2.9.1," 17 June 2021. [Online]. Available: https://pypi.org/project/psycopg2/.

[2] PySimpleGUI, "PySimpleGUI 4.55.0," 7 November 2021. [Online]. Available: https://pypi.org/project/PySimpleGUI/.

[3] Telstra Corporation Limited, Australia, "Pmw 2.0.1," 23 March 2015. [Online]. Available: https://pypi.org/project/Pmw/.

[4] pgMustard, "PostgrelSQL EXPLAIN," October 2021. [Online]. Available: https://www.pgmustard.com/docs/explain.