NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE

AND ENGINEERING

CZ4031 Group 31 Project 2 Report

CHIA SONGCHENG	U2021738F
CHIN KIN MUN	U2020779H
LIU LIWEN	N2202357F
LUO MINJUAN	N2202356J

Table of Contents

1. Introduction

The report is going to explain details of the software including formal descriptions of the key algorithms with examples base on the submission.
Hence the content should be:

1. project.py
2. interface.py
3. explain.py

2. project.py

The software provided is a Python-based application that allows users to execute and visualize query execution plans for a database. It is composed of three main Python files. The program uses the PostgreSQL database and connects to a specified database by providing the necessary login credentials. The main program file, project.py, is responsible for invoking all the necessary procedures from the other two files.

2.1. Database:

The Database contains a class that initializes the connection to the database and has methods for executing SQL queries and returning the results.

The class constructor takes the name of the database as an argument and initializes the necessary login credentials such as the hostname, username, password, and port id. The psycopg2 library is used to establish a connection to the database.

The method get_query_results(query) take an SQL query as an argument, executes the query using the established connection and returns the results.

The method print_explanation_results(queryresult) print out the explanation of the query execution plan while print_query_results(queryresult) print out the query results. This is used for debugging.

2.2. main function:

The main function in the project.py file will invoke all the necessary procedures from the other three files. It initializes a database object and a user interface object and connects to the database.

When the user inputs an SQL query and presses the execute button, the program executes the query using the database object and displays the results in the table widget of the user interface.

3. interface.py

The file interface.py contains the code for the GUI.

The interface.py file provides a graphical user interface for executing SQL queries. The file uses the PyQt5 library for building the GUI. The file consists of a class called MyWindow that is a subclass of QMainWindow. The following are the key components of the GUI:

The class constructor initializes the user interface window and creates the necessary widgets including text boxes, labels, and tables to display the query results. It also defines the necessary functions to execute the SQL query, retrieve the query results, and display them in the table widget.

1. Textboxes for inputting SQL queries: The GUI has two textboxes for inputting SQL queries. These textboxes are named queryTextbox1 and queryTextbox2.

2. Textbox for inputting the database name: The GUI has a textbox for inputting the name of the database. This textbox is named dbNameTextbox.

3. Labels for displaying output: The GUI has three labels for displaying output. These labels are named queryOutput1, queryOutput2, and queryExplain.
    a. The queryOutput1 label displays the output of the first SQL query
    b. The queryOutput2 label displays the output of the second SQL query
    c. The queryExplain label displays the user-friendly explanation of the changes to the two inputted query execution plans that take place during data exploration.

4. Button for executing SQL queries: The GUI has a button named submitButton for executing the SQL queries.

5. Error message box: The GUI has an error message box for displaying error messages.

The program uses the following algorithms:

1. onClick() method: This method is executed when the submitButton is clicked. The method first checks if the database name entered by the user is the same as the current database name. If not, it establishes a connection with the new database. Next, it calls the process () method from the project.py file to execute the SQL queries and obtain the output. It then calls the explain () method from the explain.py file to obtain the execution plan of the first SQL query. Finally, it displays the output and execution plan in the appropriate labels.

2. ScrollableLabel class: This is a custom class that extends the QScrollArea class. The ScrollableLabel class is used for displaying the output and execution plan in the GUI. The class provides a setText() method that sets the text to be displayed in the label.

Conclusion:
The interface.py file provides a GUI for executing SQL queries. The file uses the PyQt5 library for building the GUI. The program uses the onClick() method and the ScrollableLabel class for executing the SQL queries and displaying the output and execution plan in the GUI.

4.explain.py

The explain.py contains the code for generating the explanation.

The program uses the PostgreSQL EXPLAIN command to get the execution plans for the two queries, and then compares the node types and depth of the plans to identify any changes. The program outputs a report describing the differences between the two plans. This output is shown in Appendix B. This result is written in the text box named “Explain for Output Query:”.

```
def explain(self, database, query1, query2):
    print("working on explain")
```

Figure 1: explain function

The program defines a class called ‘explain’ and a method called ‘explain’ which takes three arguments: ‘database’, ‘query1’, and ‘query2’. The database argument is an object representing the database to execute the queries on, while ‘query1’ and ‘query2’ are strings representing the two SQL queries to compare.

4.1. Retrieval of query plan

The "explain" method first retrieves the query execution plans for the two SQL queries using the "get_query_results" method of the database object. It then uses a helper function called "get_node_types" to extract the node types and execution times from the raw explanation of the query execution plan. The "get_node_types" function recursively traverses the query execution plan and extracts the node types and execution times of each operation.

```
def get_node_types(raw_explanation):
    node_types_time = []
    node_types = []
    for Plans in raw_explanation[0][0]:
        node_types_time, node_types=get_node_helper(Plans['Plan'])
    return node_types_time, node_types
```

Figure 2: Function for get_node_types

This function takes raw explanation generated by PostgreSQL explain command as input and gets information about which operations are performed and the execution time for each operation. It uses a get_node_helper function to recursively get information about all operations and return two lists: the operations with their execution time, and the name of operations that are used to compare the difference between two queries.

```
def get_node_helper(raw_explanation):
```

Figure 3: Function for get_node_helper

The get_node_helper function is a recursive function that recursively goes through each depth of the query execution plan tree to get information about operations and adds them into lists. The get_node_helper function is called by the get_node_types function.

4.2. Check depth of query plans

Figure 4: Function for check_depth

The "explain" method then compares the node types of the two-query execution plans to identify any differences. It uses another helper function called "check_depth" to calculate the depth of each plan and checks if they are the same. If the depth is different, the method adds an explanation stating the difference.

4.3. Differences between two query plans

Next, the "explain" method compares the node types of the two-query execution plans to identify any differences in the operations performed. It uses a nested loop to compare the node types of each plan and identify any nodes that are present in one plan but not the other.

4.4. Checking for parallel execution and limit

The program then checks if a "Gather" node was added or removed and adds an explanation to indicate whether parallel execution was enabled or disabled. It also checks for limit by checking for a “Limit” node in the query plan.

4.5. Checking for similar operations

The "explain" method identifies any nodes that were present in both query execution plans but had a different depth in each plan. It generates an explanation to highlight these nodes, indicating that the operations were the same but were executed at a different depth.

It stores these nodes in a list called ‘same_operation_diff_depth’, removes them from the ‘nodes_in_query1’ and ‘nodes_in_query2’ lists, and then proceeds to compare the remaining nodes in the same way as before.

4.6. Check for any other unique operations

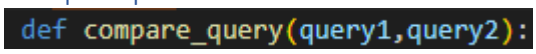

Figure 5: Function for compare_query

The compare_query function checks for any other unique operations in both queries so as to map the differences in the query plan. It removes similar parts from the queries and gives an array of three arrays. Each of the three arrays contains another two arrays of strings to show the differences in the select clause, where clause and the from clause. It then adds the explanation for the unique operations done in one query compared to the other based on the differences in the queries.

Overall, the explain method aims to provide a detailed and informative comparison of the query execution plans of query1 and query2. The method achieves this by recursively traversing the execution plan trees of the two queries, extracting relevant information, and identifying differences and similarities between the two.

5. Features and limitations

5.1. Features

1. Able to identify whether parallel execution is used for the query.
2. Able to identify if there is a limit to the number of tuples in the query plan.
3. Able to show the complexity of the query plan:
    a. Able to show the depth of the query plan.
    b. Able to show the different number of operations in the query plan.
4. Gives reasons why the query plans are different.
5. Looks up at the query and check for differences to explain the differences to the query plan.

5.2. Limitation

1. Only up to two query plans are generated and get compared.
2. The query execution system will work slow if the query output contains many results.
3. Unable to map each difference in the query plan to a difference in a query.
4. If the query run into an error, the program must be terminated and start again to work.

Appendix A: Installation Guide

Steps to install the python environment:

1. Install python from https://www.python.org/downloads/

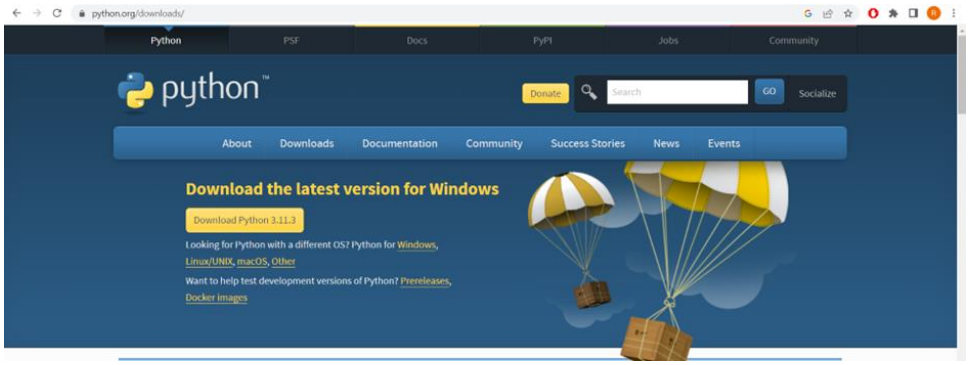

Figure 6: Figure of website to install python

2. Set up virtual environment:
    a. Open Python terminal
    b. Enter command:
        Pip install virtualenv
        python -m venv venv
        Which would create a virtual environment with venv as its name.
    c. To activate the virtual environment enter the following in the terminal: *.\venv\Scripts\activate*
    d. Libraries installation

    After the virtual environment is activated, install the following libraries by entering the following:
    Pip install psycopg2==2.9.1
    Pip install PyQt5==5.15.9

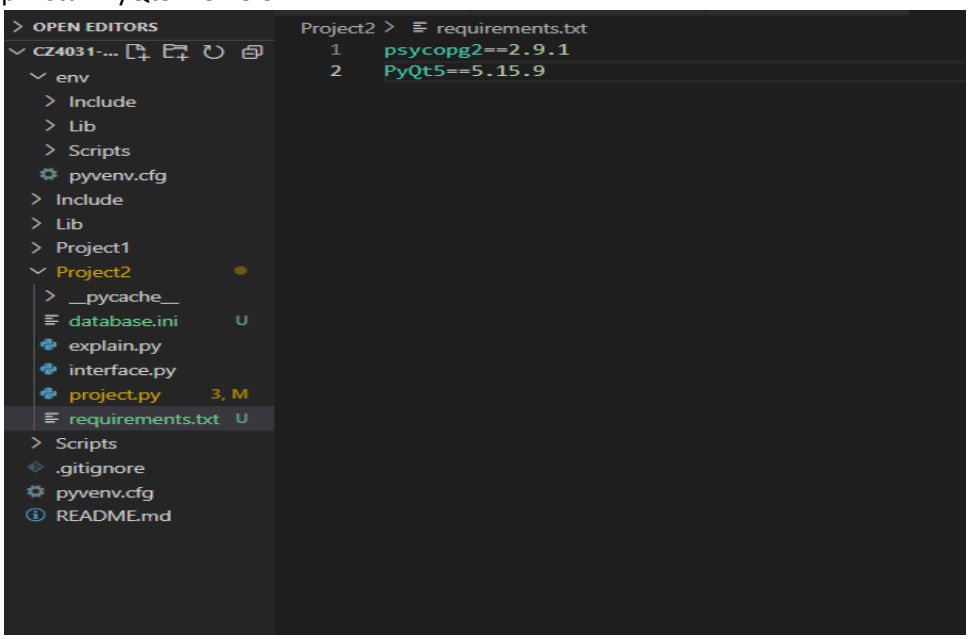

Figure 7: Figure of requirements.txt

    Virtual environment is now set up.
3. PostgreSQL extension is enabled/installed in VisualStudioCode(VSC)

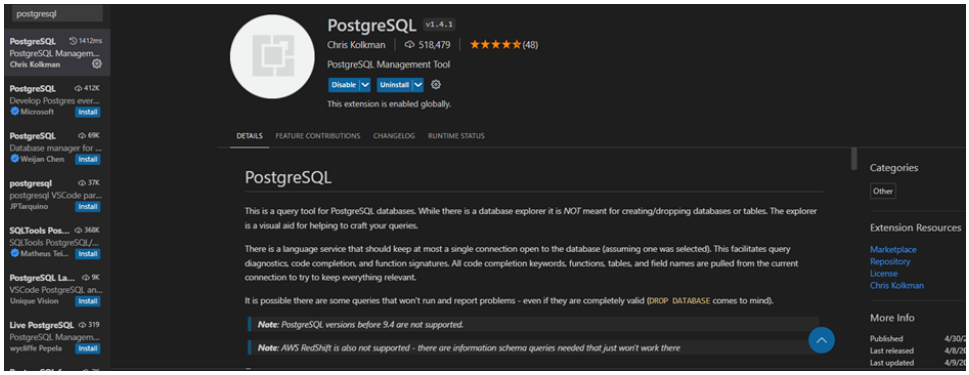

Figure 8: Figure of PostgreSQL extension in VSC

4. Modify database.ini with login details for database

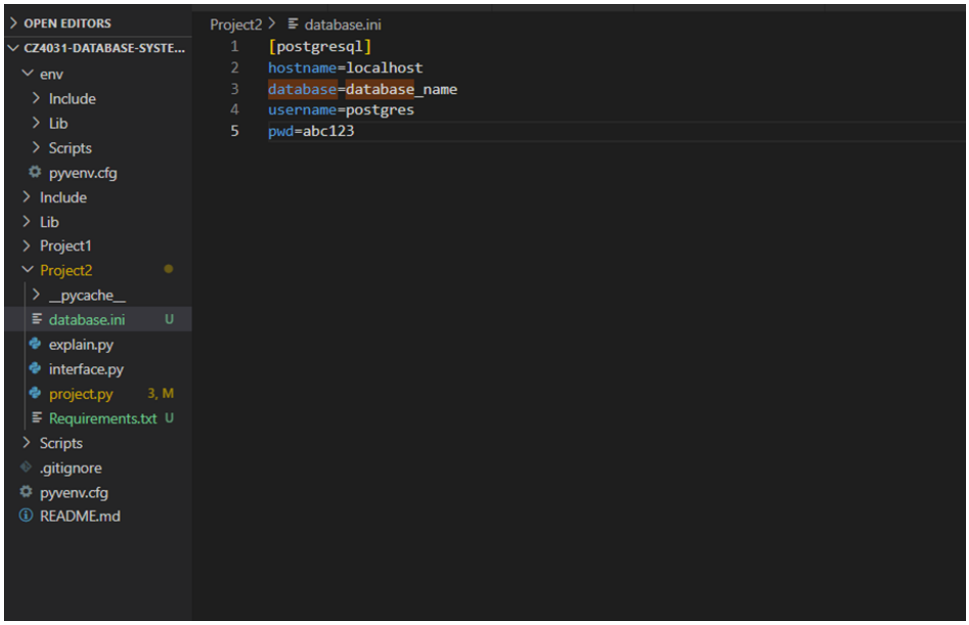

Figure 9: Figure of database.ini in the project folder

Appendix B: Examples

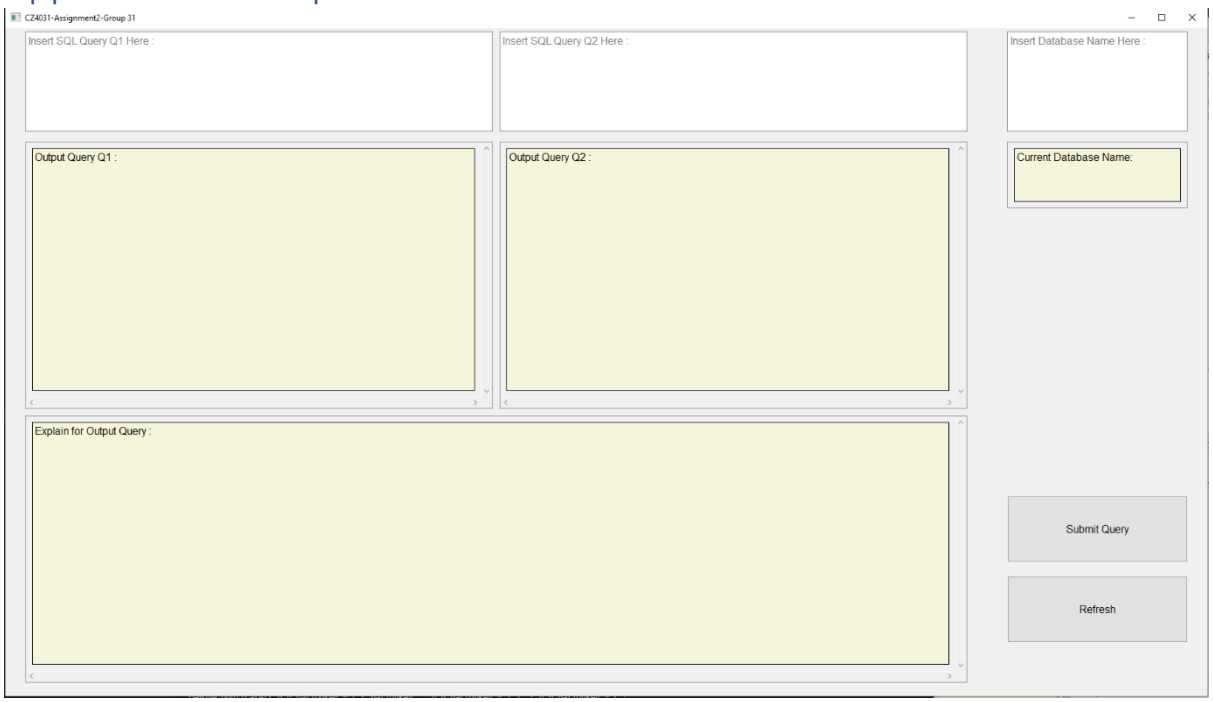

Figure 10: UI for showing the explanation and output of the query.

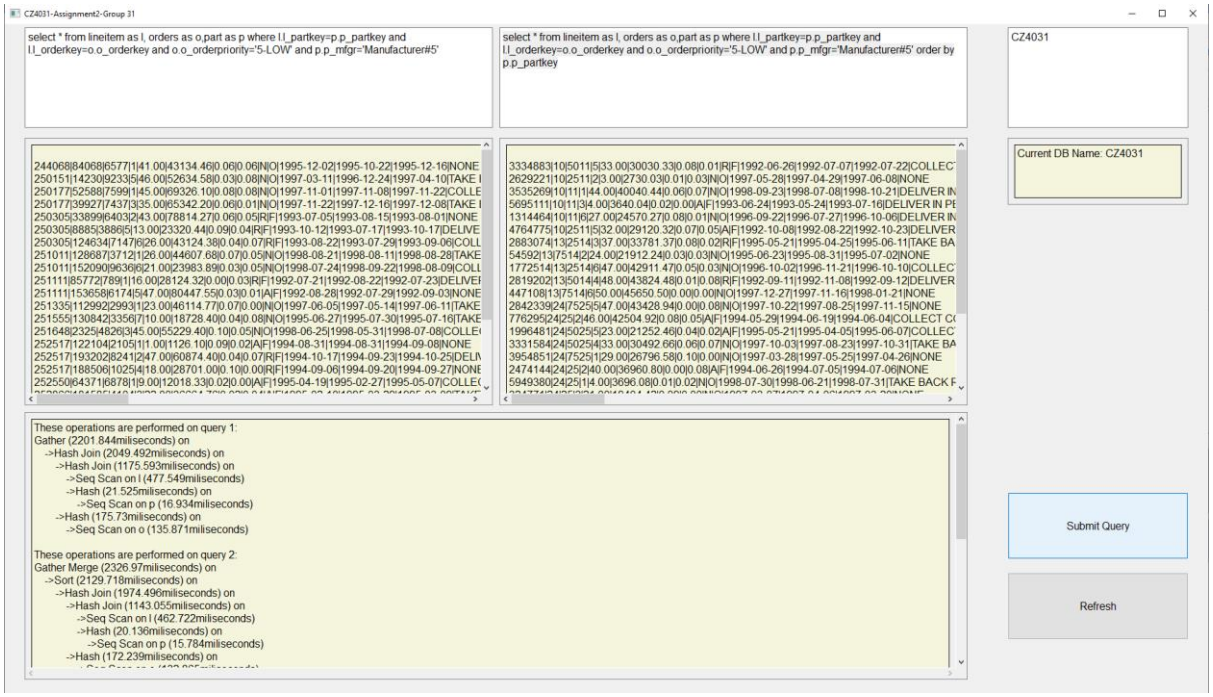

Figure 11: Execution of Query Pair A part 1

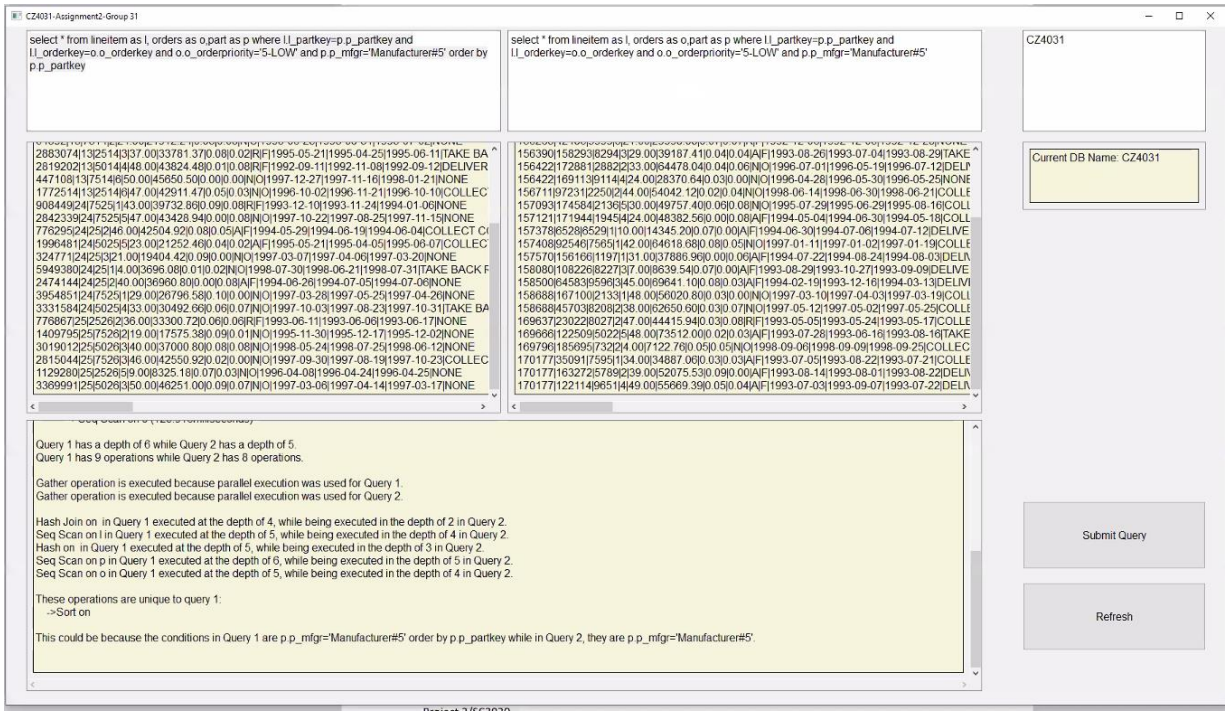

Figure 12: Execution of Query Pair A part 2

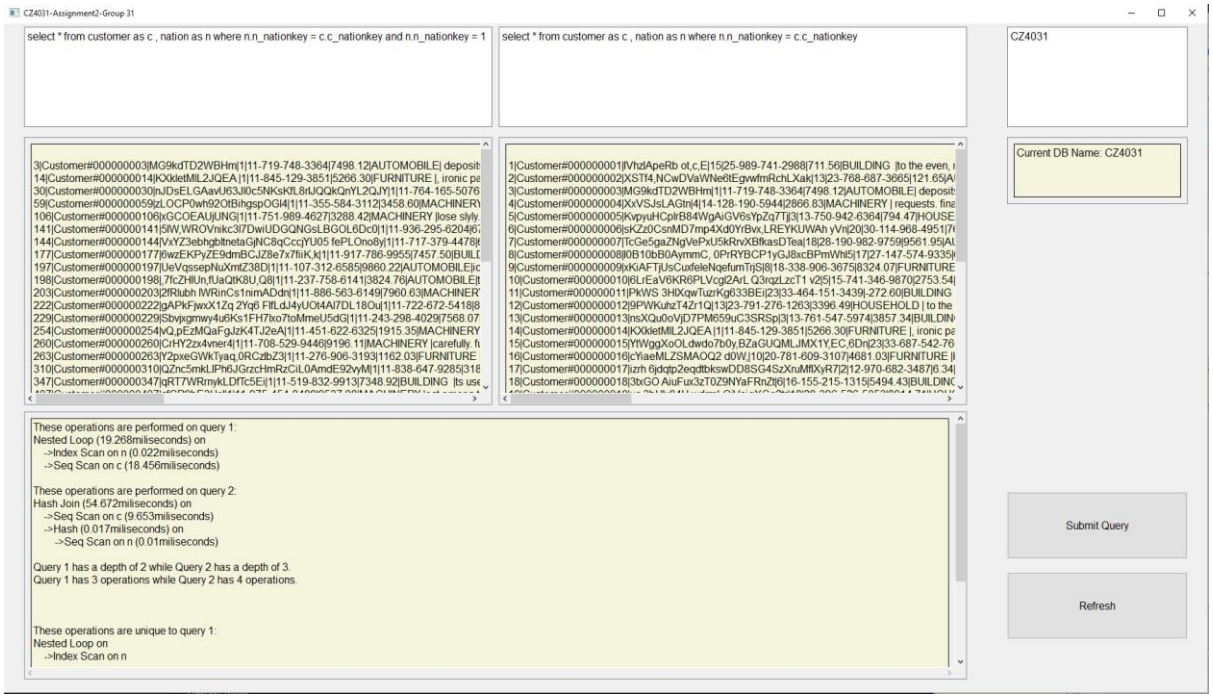

Figure 13: Execution of Query Pair B part 1

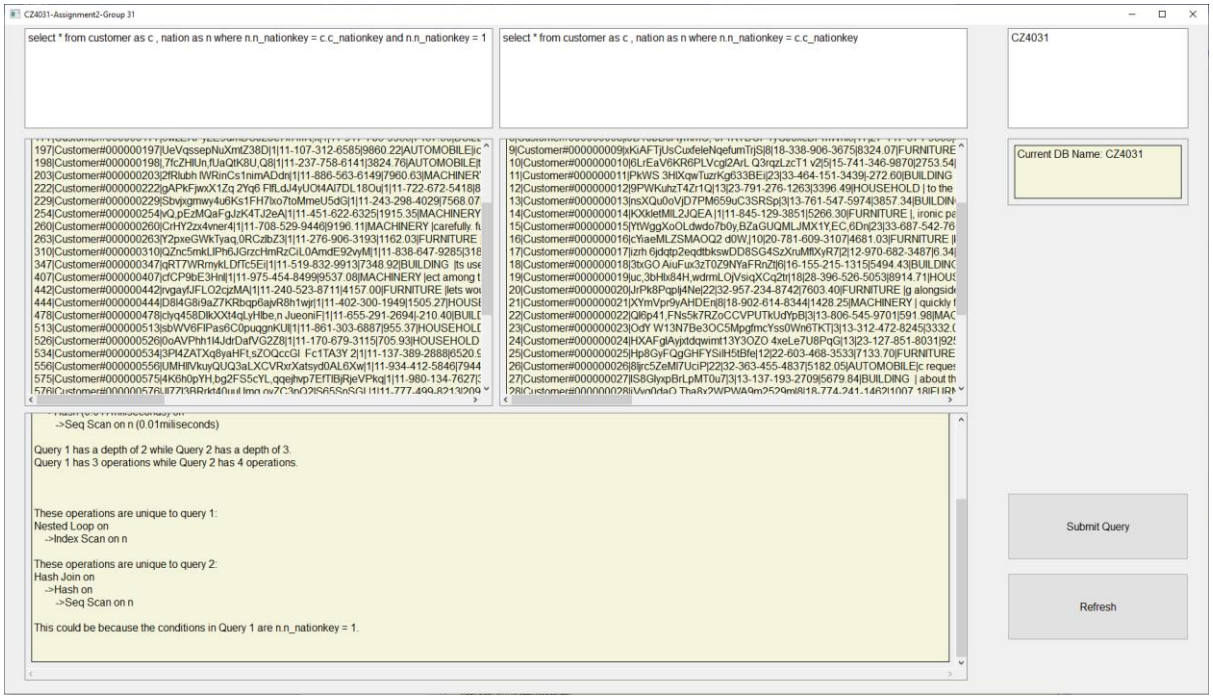

Figure 14: Execution of Query Pair B part 2