# SC3020 Database System Principles

# Project 2 Report

**Group Members:**

| **Name of Team Member** |
| --- |
| Gupta Tushar Sandeep |
| Jadhav Chaitanya Dhananjay |
| Sethi Aryan |
| Lin Run Yu |

**Installation Instructions: Please Refer to the Appendix in this Report**
**Github Repository:**
**https://github.com/TushG29/SC3020_DatabaseSystemPrinciplesProject2**

# Table of Contents


**1. Introduction** **3**
1.1. PostgreSQL 4
1.2. TPC-H Dataset 4
1.3. Data Massaging of TPC-H 5
1.4. Data Seeding of TPC-H 5
**2. Graphical User Interface** **6**
2.1. Login 6
2.1.1 Error Handling 7
2.2. Main Application 7
2.2.1. Query Input 8
2.2.2. Execution of Query 8
2.2.3. Display of the Query Execution Plan 8
2.2.4. Analysis of the Query Execution Plan 8
2.2.5. Display of the Query Execution Plan Tree for Visualization 9
2.2.6. Display of the Block Visualization 11
2.2.7. Reset Query Input 14
2.2.8. Invalid Query 15
**3. Design and Implementation** **16**
3.1. Assumptions 16
3.2. Libraries 16
3.3.1. psycopg2 16
3.3.2. customtkinter 16
3.3.3. networkx 16
3.3.4. Flask 17
3.3.5. matplotlib 17
3.4. Structure of Code 17
3.4.1. project.py 17
3.4.2. interface.py 17
3.4.3. explore.py 18
**4. Algorithms** **19**
4.1. Natural Language of the Query Execution Plan 19
4.2. Query Execution Plan Tree 19
4.3. Block Access Visualization 19
4.3.1. Pie Chart 19
4.3.2. Heat Map 20
4.4. Analysis of the Query Execution Plan 20

**5. Conclusions** **23**
**6. Appendix: Instructions to Run Software** **24**
**7. Acknowledgements** **25**
**8. Reference** **26**

# 1. Introduction

## 1.1. PostgreSQL

PostgreSQL is also known as Postgres. It is one of the world's most advanced open source object-relational database management systems (DBMS). It is an object-oriented DBMS that is ACID (Atomicity, Consistency, Isolation, Durability) compliant and highly extensible, which allows the community to easily add new features and capabilities as the industry evolves.

PostgreSQL offers many noteworthy features such as

a) complex queries
b) foreign keys
c) triggers
d) updatable views
e) transactional integrity
f) multiversion concurrency control

It can even be extended by the user for adding aggregate functions, index methods as well as procedural languages. PostgreSQL is compatible with operating systems such as Linux, Windows etc. Furthermore, it provides library interfaces for seamless integration with major programming languages, including Java, C/C++, PHP, Perl, and Python.

In this project, we use Python and PostgreSQL to build an application that retrieves information and displays the Query Execution Plan (QEP), and the Query Analysis. We are also visualizing the query execution with the help of a Query Execution Plan Tree and Block Visualization.

## 1.2. TPC-H Dataset

The TPC-H (Transaction Processing Performance Council - H) dataset is used in our project and it serves as a database benchmark to assess the performance of high-complexity decision support databases. It allows our team to develop and test SQL queries on small datasets of records, before applying those same queries to larger datasets. TPC-H schema is designed to reflect the structure of a typical data warehouse. Table 1 depicts a summary of the relations in TPC-H, as discussed in Section 1.4.

## 1.3. Data Massaging of TPC-H

The TPC-H data is originally in the .tbl format. Hence we converted them into .csv as stated in the Project2 Appendix shown in Figure 1.

**Figure 1**

- Build the tpch project. When the build is successful, a command prompt will appear with "TPC-H Population Generator <Version 2.17.3>" and several *.tbl files will be generated. You should expect the following .tbl files: customer.tbl, lineitem.tbl, nation.tbl, orders.tbl, part.tbl, partsupp.tbl, region.tbl, supplier.tbl
- Save these .tbl files as .csv files
- These .csv files contain an extra " | " character at the end of each line. These " | " characters are incompatible with the format that PostgreSQL is expecting. Write a small piece of code to remove the last " | " character in each line. Now you are ready to load the .csv files into PostgreSQL

## 1.4. Data Seeding of TPC-H

Once we have generated the .csv files, we then move one to generate the tables with their appropriate schema in Postgresql based in Appendix A. Furthermore, we execute commands in order to import each of the .csv files. For example, we are importing the nation.csv file using the command in Figure 2.

**Figure 2**

```
TPC-H=# \copy "nation" from '/Users/tushar19/Documents/NTU Semester 1 Year 3/NTU Projects/
SC3020Project2/data/nation.csv' DELIMITER '|' CSV;
```

The above code is run for each of the eight tables. After the database seeding is complete, we attain the following tables in our database as shown in Table 1.

**Table 1: Summary of TPC-H Relations**

| Relations | Number of Records | Description |
|---|---|---|
| region | 5 | The continents that the nation belongs to. |
| nation | 25 | The list of supported nations. |
| part | 200000 | The parts sold along with their respective suppliers and prices etc. |
| supplier | 10000 | The details of the suppliers of the parts |
| partsupp | 800000 | The information given by the supplier about the parts, inventory and price etc. |
| customer | 150000 | A list of all customer details. |
| orders | 1500000 | A list of customer orders, including the order status and price of the order etc. |
| lineitem | 6001215 | The information of all order items of each order. |

# 2. Graphical User Interface

## 2.1. Login

The Login to PostgreSQL window allows users to be able to key in their username and password credentials. The Login window can also be expanded to the desired size of the user. it requires the user to enter the database name, username and the password as shown in Figure 3a and Figure 3b.

**Figure 3a**
A default size window

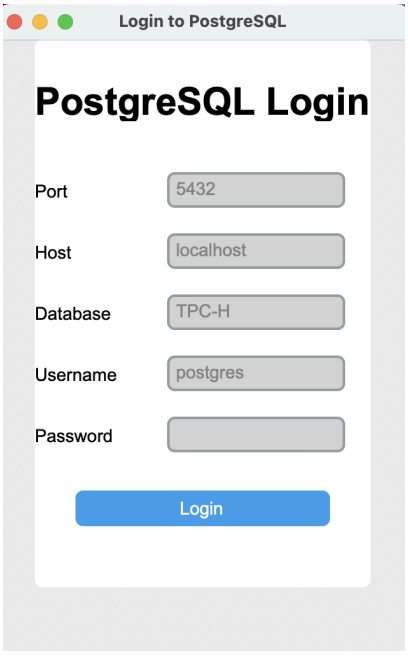
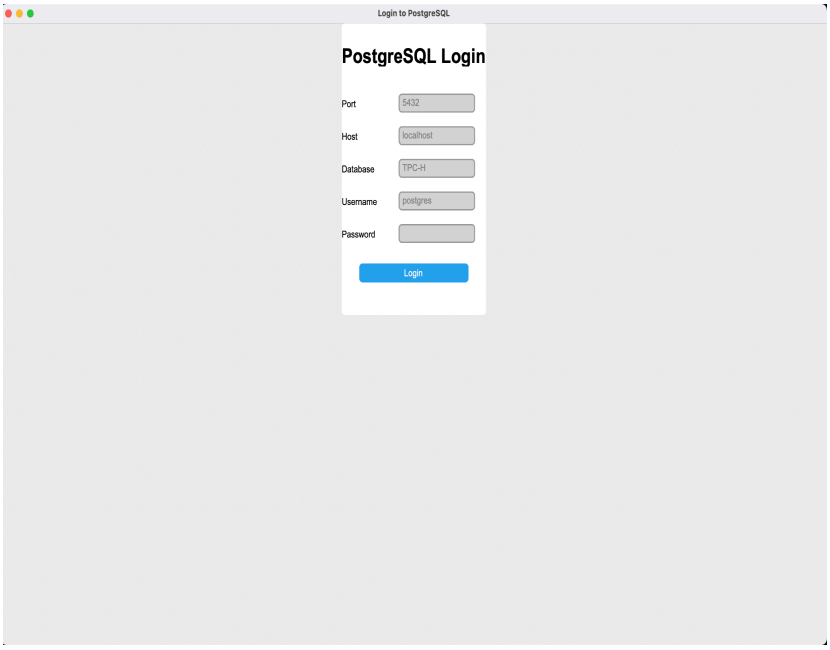


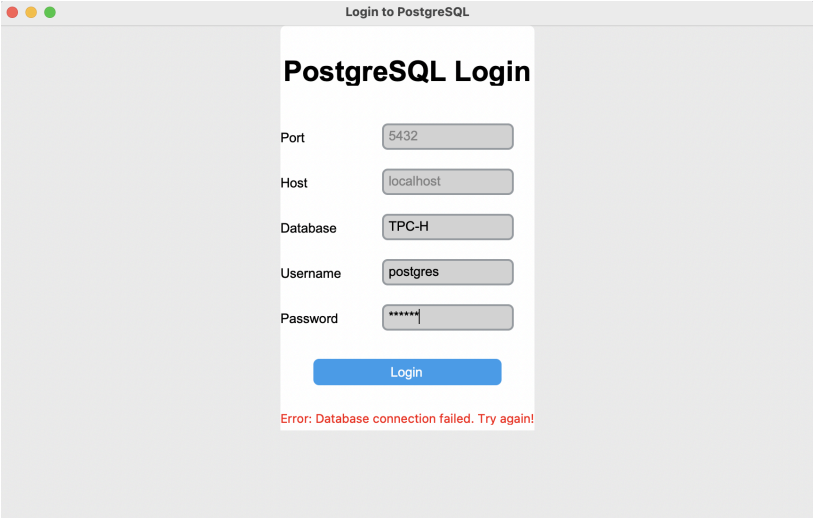

**Figure 3b**
A full screen size window



### 2.1.1 Error Handling

In the case when the credentials entered are incorrect, an error message is displayed in the Login to PostgreSQL window as shown in Figure 4.

**Figure 4**

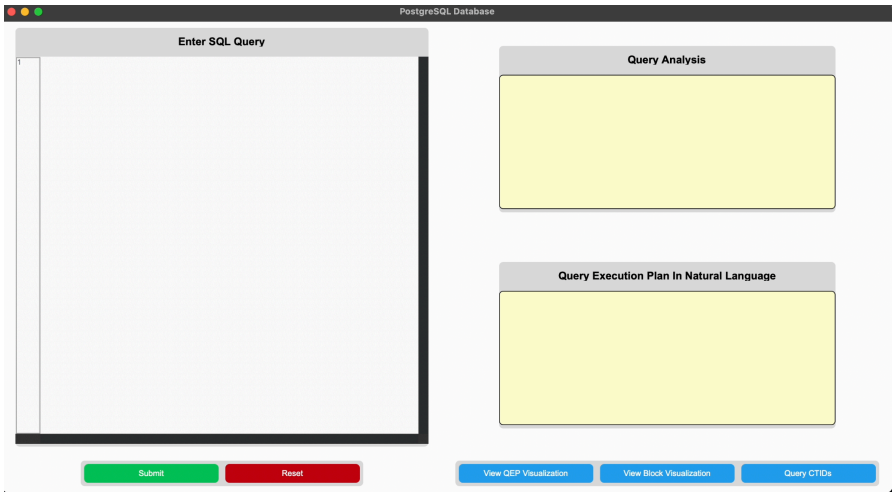



## 2.2. Main Application

If the login is successful, the user is brought to the main application screen called the PostgreSQL Database as shown in Figure 5. The main application contains three panels namely:

a) Enter SQL Query: for writing the input of the query
b) Query Analysis: for displaying the analysis of the query execution plan
c) Query Execution Plan in Natural Language: for displaying the query execution plan in natural language

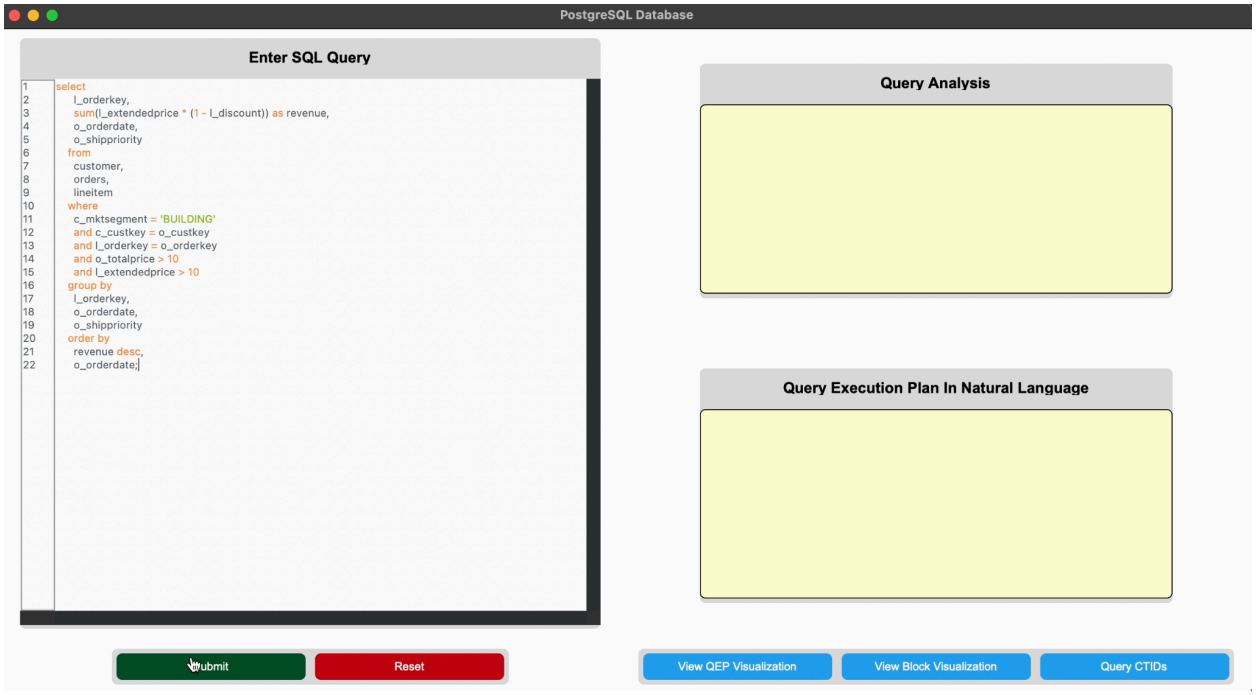
**Figure 5**



### 2.2.1. Query Input

Upon successful login, users have the capability to input their customized queries within the PostgreSQL Database window, facilitating precise database interactions in accordance with specific requirements and preferences as shown in Figure 6. We utilize CodeView from the chlorophyll module for our query input box.

**Figure 6**




### 2.2.2. Execution of Query

After the query is input, the user can proceed to execute the query by clicking the “Submit” button. Our software then proceeds to analyze the query.

### 2.2.3. Display of the Query Execution Plan

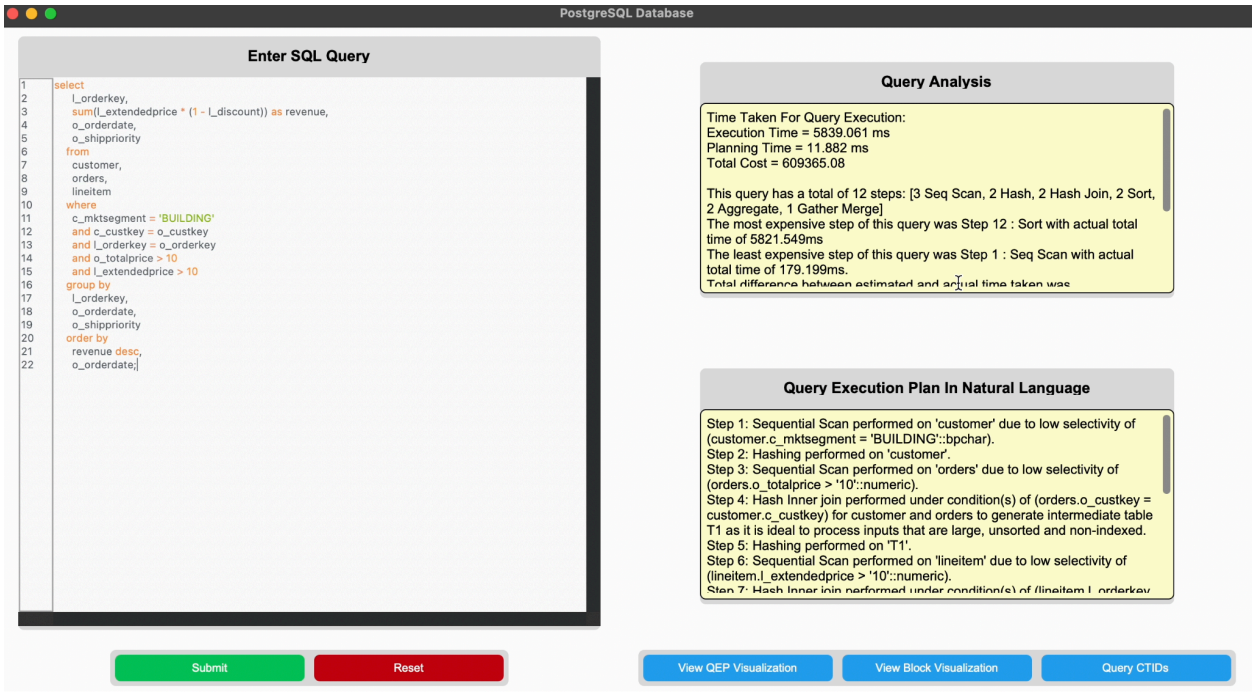
Once the query has been executed using the ‘Submit’ button, the query execution plan is generated. The detailed annotation of the steps of the optimal query plan is displayed to the user in the ‘Query Execution Plan in Natural Language’ textbox. It describes each of the steps taken by the query processor in an easy-to-understand natural language. Refer to Figure 7 in Section 2.3.4.

### 2.2.4. Analysis of the Query Execution Plan

Once the query has been executed, the analysis of the query execution plan is generated as well. The information is displayed in the ‘Query Analysis’ text-box. It displays information about the execution time, planning time, total cost and additional query analysis details. These details include information about the most and least expensive steps, total difference, total plans, node counts, and average actual time for each node type. Moreover, we display statistics about the

shared buffer as well. The analysis is displayed along with the query execution plan in natural language in Figure 7.

**Figure 7**

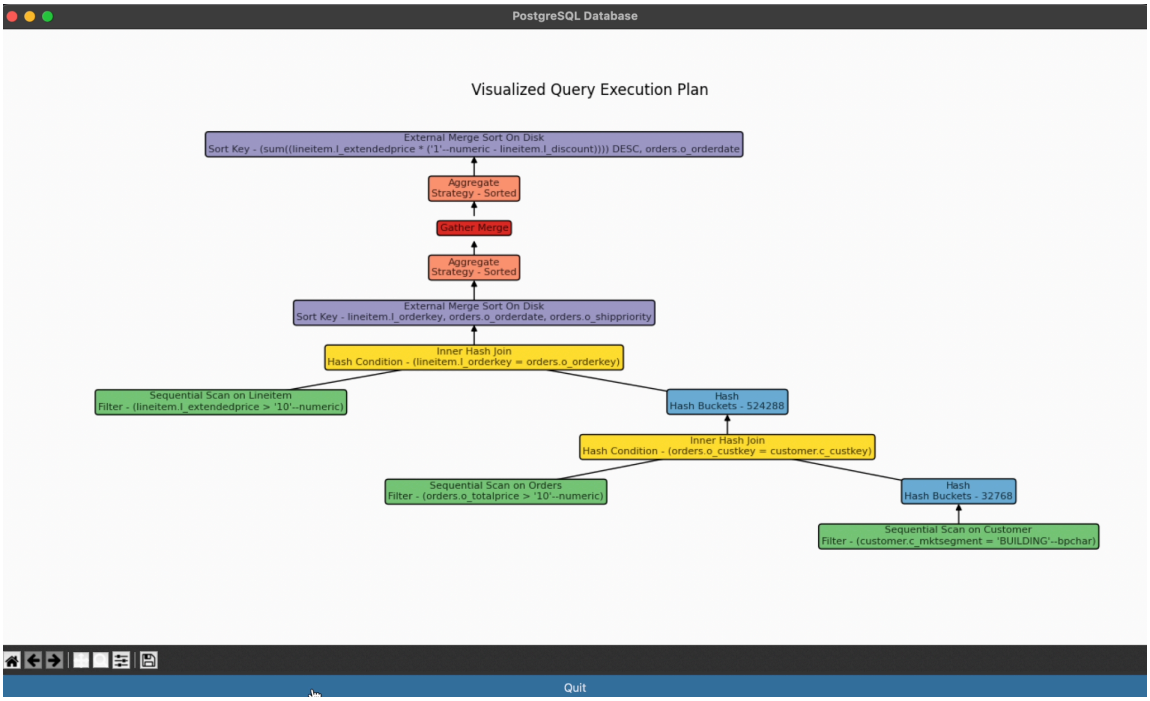



## 2.2.5. Display of the Query Execution Plan Tree for Visualization

The tree is visualized using the NetworkX library. Each node represents a unique operation in the query plan. The visualization includes node labels with customized colors based on the operation types in order to identify the different operations present in the query execution plan generated. The tree is generated in a top-down layout, and the resulting visualization serves as a comprehensive overview of the query execution flow. The user can click on the ‘View QEP Visualization button’ as shown in Figure 7 in section 2.3.4 in order to view the QEP Tree as shown in Figure 8 below.

In order to proceed further and view the block visualization, the user is required to click on the ‘Quit’ button present at the bottom of the window while viewing the Tree Visualization. This action causes the user to return back to the main application screen.

**Figure 8**



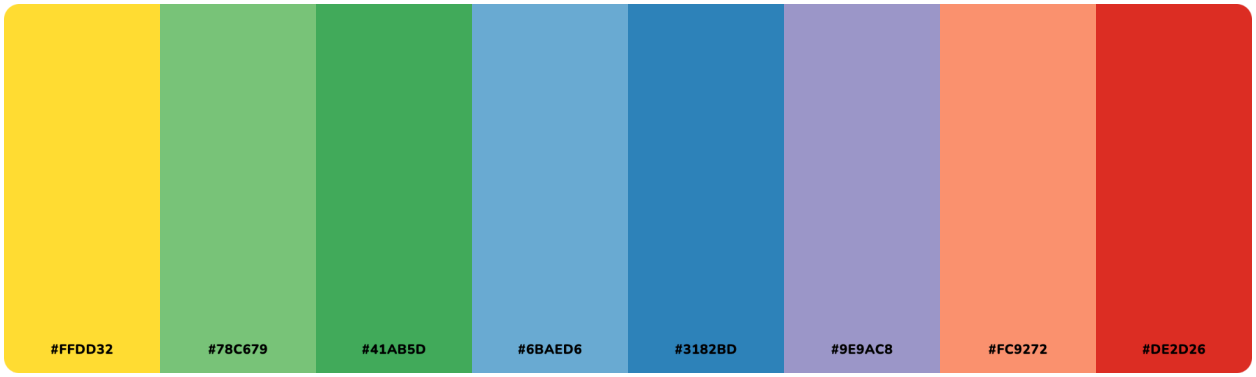

The various node types and the color corresponding to them as displayed in the QEP Tree are shown below in Table 2.

**Table 2: The Node Types and their Colors**

| Node type | Color |
|---|---|
| Nested Loop Join | Yellow |
| Hash Join | Yellow |
| Sequential Scan | Light Green |
| Index Scan | Dark Green |
| Hash Buckets | Light Blue |
| Memoize | Dark Blue |
| Sort | Purple |
| Aggregate | Light Orange |
| Gather Merge | Red |
| Gather | Red |

A snippet of the color palette used to display the different operation types in the Query Execution Plan Tree Visualization is shown in Figure 9.

**Figure 9**



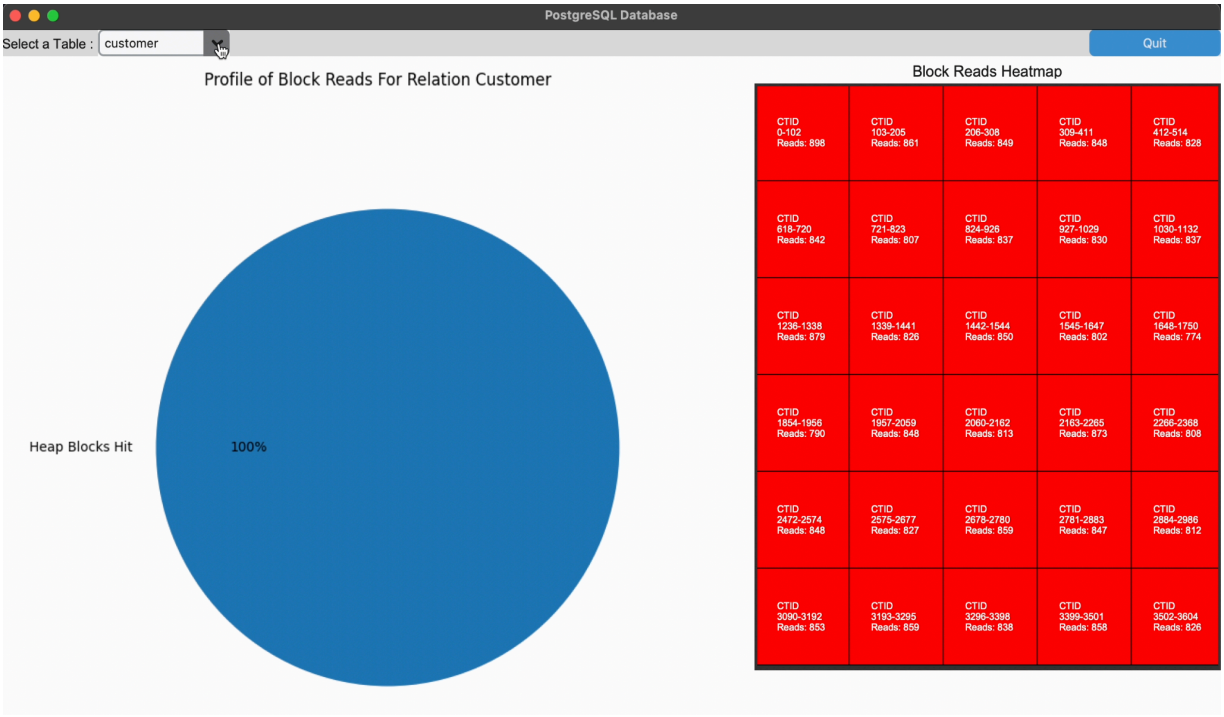

### 2.2.6. Display of the Block Visualization

If the user is currently at the main application view, the user can click on the View Block Visualization button in order to view the output in Figure 15.

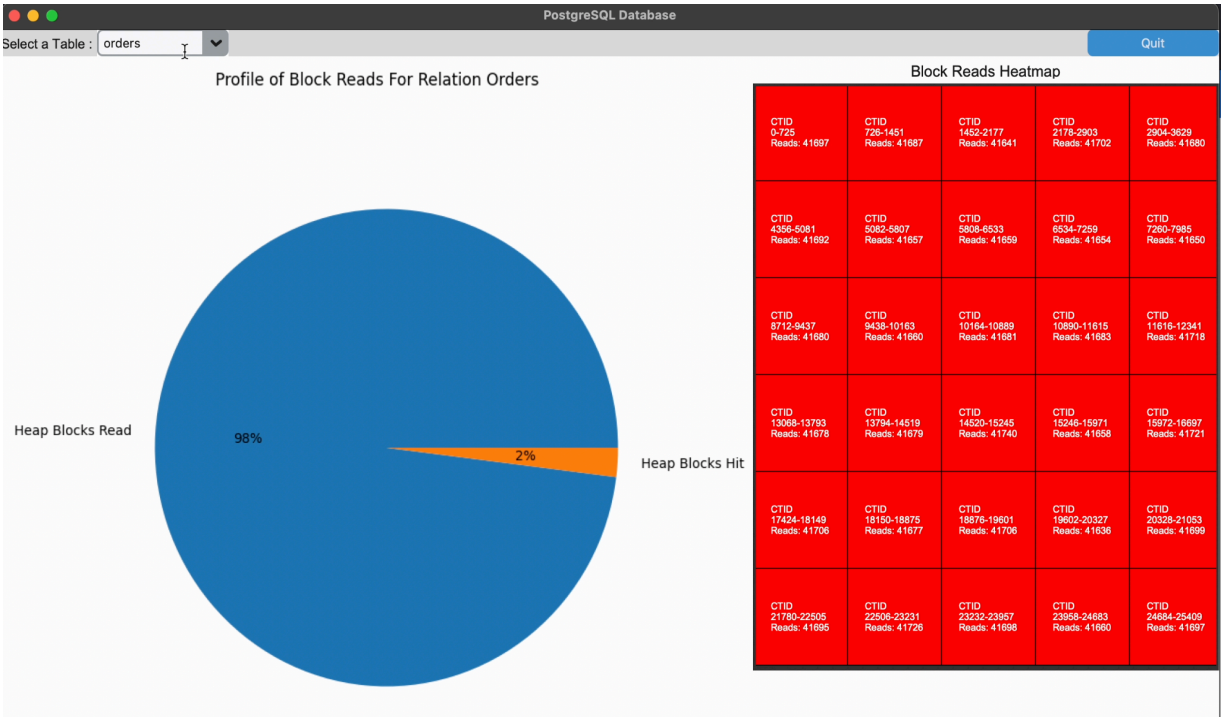
a) The below output displays a pie chart displaying the percentage of Heap Blocks Hit and the Heap Blocks Read for all the relation orders relevant to our input query.
b) Moreover it features a Select a Table option which has a drop down box to select the desired relation and display its information.
c) Finally, the user can quit the Block Visualization window, causing the user to return to the main application window.

## Figure 10a

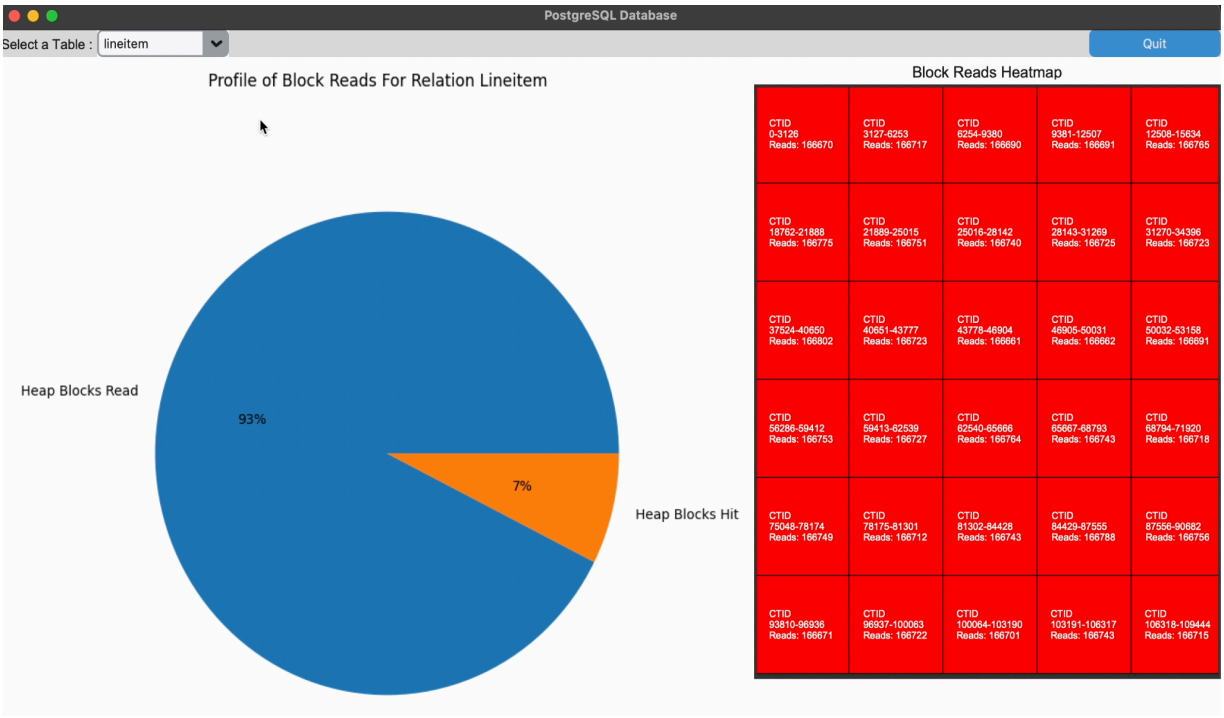

## Figure 10b



## Figure 10c



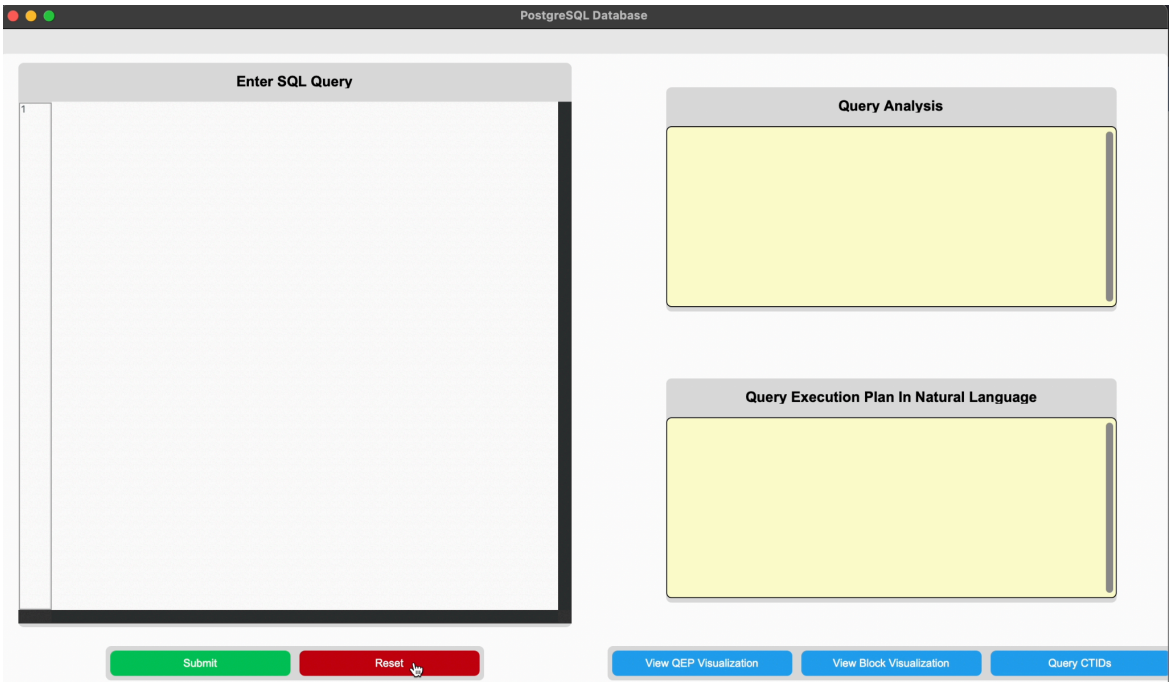

## Figure 10d



### 2.2.7. Reset Query Input

If the user wishes to visualize the outputs for another query, the user can reset the input query section by clicking on the Reset button in the Login to PostgreSQL window in order to proceed further as shown in Figure 11.

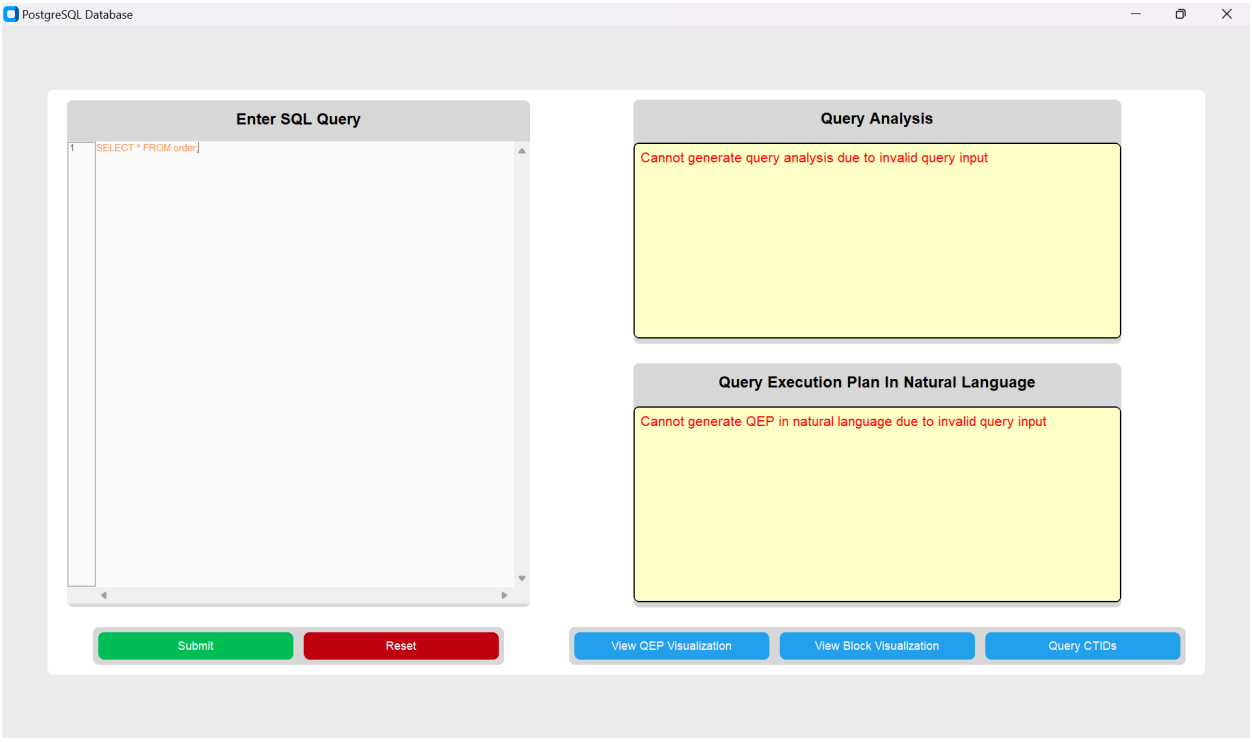
**Figure 11**



### 2.2.8. Invalid Query

If the user enters an invalid query, error messages will be displayed in the ‘Query Analysis’ as well as in the ‘Query Execution Plan In Natural Language’ text-boxes as displayed in Figure 12.

**Figure 12**



# 3. Design and Implementation

## 3.1. Assumptions

- The application is only used for reading results from a table, and doesn't handle inserts, updates or deletes.

## 3.2. Libraries

We have used several libraries in our project, amongst them there are the main 5 libraries in Python to assist in the development of our application:

a) Psycopg: For connecting to PostgreSQL, executing queries and handling results in our Flask application.
b) Customtkinter: For the GUI application that allows users to log in, submit queries etc.
c) Networkx: For creating and visualizing a top-down tree representation of the Query Execution Plan.
d) Flask: For creating a web application that provides endpoints for connecting to a PostgreSQL database.
e) Matplotlib: For data visualization, it has been used to create a pie chart as well as a heatmap which is used for visualizing the block information.

### 3.3.1. psycopg2

Psycopg is the most popular PostgreSQL database adapter for the Python programming language. Some of its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection).

### 3.3.2. customtkinter

CustomTkinter is a python UI-library based on Tkinter. It provides new modern and fully customizable widgets. All colors of the widgets can be customized All CustomTkinter widgets and windows support HighDPI scaling (Windows, macOS). CustomTkinter With CustomTkinter we are able to get a consistent and modern look across all desktop platforms (Windows, macOS, Linux). CustomTkinter is a tkinter extension which provides extra UI-elements like the CTkButton, which can be used like a normal tkinter.Button, but can be customized with a border and round edges.

### 3.3.3. networkx

NetworkX is a Python library that has been developed for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Networkx nodes can be any object that is hashable, meaning that its value never changes. These can be text, images, strings etc. It has the capacity to act on extremely large graphs with over 10 million nodes and 100 million edges. It’s mainly used for standardizing the programming environment for graphs,

working with non standard datasets as well as in rapid development of collaborative, multidisciplinary projects.

One notable feature is its compatibility with Graphviz, with the graphviz_layout function enabling sophisticated graph visualizations. ‘graphviz_layout’ has been used to compute the positions of the nodes which are then used for visualization. This helps in creating a structured and visually appealing representation of the Query Execution Plan in the top-down tree layout.

#### 3.3.4. Flask

Flask is a web framework for Python, which is designed to simplify the development of web applications. Flask provides a WSGI (Web Server Gateway Interface) compliance interface and it also includes a built-in development server and supports URL routing, template rendering, as well as request handling. Flask does not impose a rigid structure, thus allowing developers the flexibility to choose components and extensions based on the requirements. High-level tasks like database access, web form and user authentication are supported through "extensions".

#### 3.3.5. matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible. With the help of matplotlib, we can create publication quality plots, make interactive figures that can zoom, pan and update as well. Matplotlib supports both 2D and 3D plotting, and its output can be exported in various formats.

## 3.4. Structure of Code

#### 3.4.1. project.py

The project.py initiates two processes using Python's multiprocessing module. It is used to launch the Flask backend from explore.py and Tkinter’s frontend window from interface.py. In order to perform concurrent execution, multiprocessing is used which helps to improve the efficiency of the project.

#### 3.4.2. interface.py

The interface.py represents the front-end of our software. It constructs a user-friendly interface for PostgreSQL database interaction using the Tkinter library. It helps in creating a login window where users are required to input their credentials like port, host, database name, username, and password. After the user clicks on the ‘Login’ button, the program sends a request to a local server to establish a connection. A successful connection closes the login window and opens the main application window.

Moreover, the main application window contains a text-box in order to input the query, several buttons for submission and resetting, as well as other text-boxes used to display the query execution plan and analysis. The code relies on external modules for customized Tkinter

widgets, syntax highlighting, and PostgreSQL database interaction. The 'frontend()' function is used for initializing the login window and launching the overall graphical user interface.

Furthermore, it employs ‘networkx’ and ‘matplotlib’ to visually represent the complex query execution plan as a top-down tree. It utilizes the customtkinter library to develop a graphical user interface.

Additionally, the application provides users with the ability to select specific blocks for in-depth analysis. The script also includes functions for query execution plan annotation (QEPAnnotation) and analysis (QEPAnalysis).

### 3.4.3. explore.py

The explore.py represents the backend of our software. It creates an application that connects to a PostgreSQL database, executes SQL queries, and stores the resulting query execution plan in the ‘queryplan.json’ and ‘readinfo.json’ file.

The explore.py serves as a Flask web server designed for database interactions. Its functionalities encompass establishing connections to a PostgreSQL database, parsing information from queries, and furnishing detailed explanations as well as statistics for executed queries. The backend contains endpoints for operations like database connection, blocks retrieval, block visualization, and SQL query handling.

The backend() function acts as the entry point, initializing the Flask application.

# 4. Algorithms

## 4.1. Natural Language of the Query Execution Plan

a) The algorithm analyzes the PostgreSQL Query Execution Plan (QEP), and converts the technical steps into natural language explanation.
b) It traverses the plan's hierarchical structure, captures and interprets each operation by reading data from “queryplan.json” and extracting various information such as node type.
c) The resulting natural language annotations offer a step-by-step breakdown, explaining actions such as aggregations, scans, and sorts etc.
d) The final annotations enhance accessibility and thus provides users with a comprehensible and insightful overview of the Query Execution Plan.

## 4.2. Query Execution Plan Tree

a) The createQEPTree function is used to create a visual representation of the Query Execution Plan (QEP) in a top-down tree structure. The Query Execution Plan Tree algorithm processes data from “queryplan.json" to construct a directed graph using the iterateOverQEP function.
b) Here each node in the QEP is assigned a customized label based on its type and relevant information (e.g., for sorts, joins, scans, aggregates). Based on the type of node found, the function looks for additional information to insert into the query tree. For example, if the program finds an Index Scan, it looks for the index used to access the relation and the filter applied, if any. The final output is stored in the operatorSeq list, while the parent-child relationships are recorded in the parents list.
c) The function then employs the create_top_down_tree function to construct a directed graph using the collected node and parent information.
d) Finally, the visualize_tree function utilizes networkx and matplotlib to create a visual representation of the QEP tree. Each node is color-coded based on its type as mentioned earlier in Section 2.3.7. (e.g., Nested Loop Join, Sequential Scan) for improved readability.

## 4.3. Block Access Visualization

### 4.3.1. Pie Chart

a) The Block Visualization algorithm processes data from "readinfo.json" to construct a visual representation of block reads and hits for various database relations.
b) Using the PostgreSQL internal statistics tracker, we are able to track the number of the following types of disk reads:
    i) Blocks read from Heap
    ii) Blocks read through Indexes
    iii) Blocks read through table’s TOASTs
    iv) Blocks read through TOAST indexes
    v) Blocks read from main memory

c) This information is used to create a pie chart with matplotlib to provide users with a representation of how the tuples are obtained from a relation for a query.

#### 4.3.2. Heat Map

a) The “queryplan.json” file is processed again. All scan operations on tables are isolated along with the filters used for scanning, if any.
b) Using the information of relation and filter, SQL queries are re-constructed and executed to obtain the CTIDs and tuples that would have been read by the database.
c) For every page ID in postgres, our application backend generates the number of records that were read from a particular page. This information is passed to our frontend heatmap generator function.
d) The heatmap displays in sequential blocks:
    i) The CTIDs used to read a table
    ii) The value of the number of reads
e) In the event that the number of CTIDs exceeds the available block size in the heatmap, aggregation is carried out to divide the range of CTIDs into equal intervals, and summing up the reads for each CTID in this aggregated block. This ensures that our program will still provide a reasonable output for queries with large results, for example, selecting all records from the lineitems table in the TPC-H dataset.

#### 4.3.3. CTID Table Visualisation

a. This feature allows the user to view the blocks of each table accessed during a query. Tuples accessed by the query are highlighted in green.
b. In order to obtain this information, our application obtains and repeats all scans that were carried out by a query via SQL queries.
    i. Filters used in the scan are also attached to the query and aliases of tables are replaced with the actual table names. This enables our data gathering to work on all types of queries.
c. Once we have isolated the relation and filters used,

## 4.4. Analysis of the Query Execution Plan

a) The QEPAnalysis function processes the Query Execution Plan (QEP) and generates a detailed analysis of the plan's performance metrics. It uses the analyze_execution_plan function, which is recursively used to traverse the nested structure of the QEP, and extracts information about each node, including node type, actual total time, and estimated total time.
b) Within the analyze_execution_plan function, the JSON data representing the QEP is parsed, and the algorithm accumulates insights, such as the most and least expensive steps, the total number of steps etc.
c) The algorithm distinguishes the most and least expensive steps based on actual total time and also computes the overall difference between estimated and actual times. The results are organized into a dictionary. Moreover, the algorithm provides a breakdown of

node counts and the average actual total time for each node type. The analysis extends to shared buffer block statistics, offering valuable insights into resource utilization.

## 4.5. Database Connection

a) The function connect_to_db() handles the connections to the database. The endpoint receives the database credentials and creates a connection to the database.
b) After a successful connection, a unique token is created which is used with every other endpoint to ensure secure transmission of data. The connection is stored in a global variable to allow subsequent requests to refer to this database connection.

## 4.6. Query Handling

a) The endpoint handle_query() handles all data retrieval. It authenticates the user first, looking for the database connection token.
b) If the authentication is successful, the query is retrieved from the request. Two helper functions are used for query handling, one for execution of query and one to execute EXPLAIN command queries. This is done to follow a more structured and convenient approach.
c) Before and after every query execution, two particular queries are executed. The first query before the execution is `"SELECT pg_stat_reset();"` which is used to reset the statistics of the PostgreSQL server. The second query is `SELECT * FROM pg_statio_user_tables;` which monitors the I/O activity related to user-defined tables in the database.
d) The second query used to get the data from *pg_statio_user_tables* helps us monitor the server activity and draw conclusions from the data. The pie chart drawn for block visualization uses the data retrieved and helps us understand the I/O patterns by comparing the heap blocks read to heap blocks hit. This can be used to draw conclusions such as the database doing a lot of disk reads, which could be a performance bottleneck
e) After query execution, the results of the EXPLAIN query, the server activity and the blocks per relation in the database are returned to the frontend, where it is dumped into 2 files, *querydata.json* and *readinfo.json*. The data is dumped into these files and read in the frontend for faster processing so that there is no need to keep repeating the execution of the same query.

## 4.7. Block-level access of tuples

a) The endpoint tuples_in_blocks() handles all data retrieval of tuples from the block level. The endpoint also authenticates the request first like other endpoints and only proceeds with execution if the authentication is successful.
b) The following query is executed:

```
WITH all_blocks AS ( SELECT (ctid::text::point)[1] AS tuple_id, * FROM
{relation} WHERE (ctid::text::point)[0] = {block}) SELECT ab.*, CASE WHEN
bc.tuple_id IS NOT NULL THEN 'Yes' ELSE 'No' END AS result_column FROM
all_blocks ab LEFT JOIN (SELECT (ctid::text::point)[1] as tuple_id, * FROM
```

```
{relation} WHERE (ctid::text::point)[0] = {block} AND {filter}) as bc ON
ab.tuple_id = bc.tuple_id;
```

The query retrieves all the blocks and adds a column called result_column to the results, with “Yes” given to all the queries that were accessed during the first query given by the user. This helps to highlight the queries accessed in the UI.

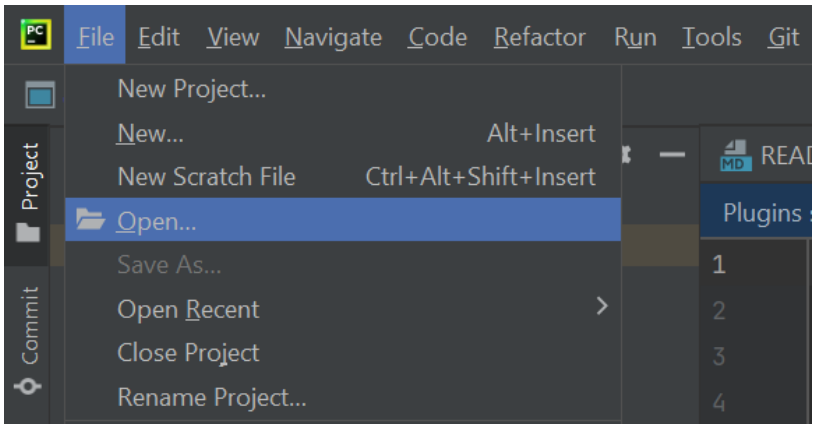
# 5. Conclusions

Our software enables users to better understand what is happening *“inside the box”* of a database system when they pose a SQL query. It has an easy to use interface and is feature-rich. Users will know various behind the scenes processes of their query, such as time taken and most/least costly steps. Users can also better understand the set of physical operators that takes one or more data streams as input and produces an output data stream through our QEP output in natural language and our QEP tree diagram. Last but not least, users can gain a better understanding of blocks in their database. As such, users will gain more insights into the inner workings of databases.

# 6. Appendix: Instructions to Run Software

**ENSURE THAT YOU ARE USING PYTHON 3.11 TO RUN OUR APPLICATION CODE.**
**WE RECOMMEND CREATING A VIRTUAL ENVIRONMENT TO INSTALL OUR PACKAGE DEPENDENCIES.**

1. Install GraphViz 9.0.0 from https://graphviz.org/download/ for the operating system you are running the source code from. This software is used to render our QEP. No further action is needed after it is installed on your computer.
    a. It is available via Windows Package Manager with command - winget install graphviz
    b. It is available via Homebrew with command - brew install graphviz.
    c. Please restart the terminal / IDE after installing this package and before running our code to allow the path variable to be updated.
2. Open PyCharm
3. Click '**Open…**' under '**File**' tab:




4. Select our folder
5. Press '**Alt+F12**' to go to PyCharm terminal:


6. Type '**python -m venv project2**' to create virtual environment:

```
PS C:\GitHub\dbsp-project-2> python -m venv project2
```

7. Type '**project2\Scripts\activate**' to activate virtual environment:

```
PS C:\GitHub\dbsp-project-2> project2\Scripts\activate
(project2) PS C:\GitHub\dbsp-project-2>
```

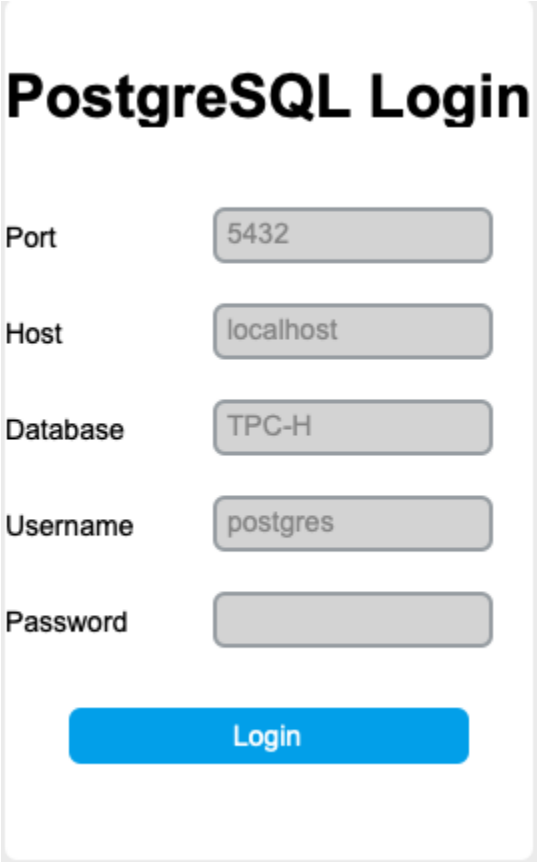
8. Type '**pip install -r requirements.txt**' to install all dependencies:

```
(project2) PS C:\GitHub\dbsp-project-2> pip install -r requirements.txt
```

9. Type '**python project.py**' to run our software:

```
(project2) PS C:\GitHub\dbsp-project-2> python project.py
```

10. On the login screen, please enter the following information:
    a. Port - corresponds to the port number that your PostgreSQL server is running on. This can be obtained through inspecting server properties on applications like pgAdmin4.
    b. Host - localhost if you are running the PostgreSQL server on the same machine as the code is being run on.
    c. Database - name of the database on your PostgreSQL server that you wish to make queries to.
    d. Username - postgres username
    e. Password - postgres password




11. It is advised to close the application every time from the terminal via <Control + C> keyboard shortcut and not the window directly to ensure the flask app is terminated and the port 5000 is cleared. In the case that the port is occupied, the following commands can help you clear the 5000 port to ensure a smooth connection to the flask app.

    sudo lsof -i :5000 (take note of the Process ID)

    sudo kill -9 <*Process ID of the python app*>

These commands should help you clear the port.

## 7. Acknowledgements

We hereby acknowledge the use of ChatGPT [ A large language model-based chatbot developed by OpenAI ] to aid us in the enhancement of this document as well as for the improvement of our code structure for the following parts of the course project:

a) Natural language of QEP
b) Analysis of QEP
c) Implementation of some portions of user interface
d) Accessing of elements in the json file

## 8. Reference

[1] https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp
[2] https://pypi.org/project/Flask/
[3] https://matplotlib.org/
[4] https://pypi.org/project/customtkinter/0.3/
[5] https://networkx.org/