****

**CZ4031 Database System Principles**

**Project 2 Report**

**Submission Date: 13/11/2022**

**Group 58**

|  |  |  |
| --- | --- | --- |
| **Member’s Name** | **Matriculation Number** | **Contributions** |
| Gideon Patrick Manik | U2021002K | annotation.py, Report |
| Leow Ken Hing Bryan | U2021729K | project.py, interface.py, preprocessing.py, Report |
| Mark Yong Wei Jie | U1922271D | annotation.py, Report |
| Ong Si Ying | U2021473H | project.py, interface.py, preprocessing.py, Report |
| Rajagopal Tejas R | U2023194K | preprocessing.py, Report |

School of Computer Science and Engineering

**Table of Content**

[**1. Introduction**](#_tn95qyvbcf83) **3**

[1.1 PostgreSQL](#_9w99vercdbkz) 3

[1.2 TPC-H Dataset](#_xk2bqq5hcu0n) 3

[1.3 Data Massaging of TPC-H](#_kqj2avhl8ucr) 3

[1.4 Data Seeding of TPC-H](#_buo4pbrn7f8x) 5

[**2. Graphical User Interface (GUI)**](#_le4ddx42sf8u) **6**

[Colour Scheme](#_oozg464c9xwn) 6

[2.1 [Generality] Login](#_c32scna1v2ze) 6

[2.2 Error Handling](#_wvexmh4e4njx) 7

[2.3 Main Application](#_8e4jio521h0k) 7

[2.3.1 [Generality] Database Selection](#_2bujekadjf1e) 8

[2.3.2 Tables and Columns in Database](#_tjyx4ihrxj1g) 9

[2.3.3 Query Input](#_9xbpeiz8qp58) 9

[2.3.4 Selection of Parameters](#_vh79t6wyiakl) 9

[2.3.5 Execution of Query](#_vznmblukkl95) 9

[2.3.6 Display of Query Plan](#_srjuuew8fefi) 10

[2.3.7 [Generality] Visualisation of the Total Cost against the Plan Configuration](#_askpxzz2nyvs) 10

[2.3.8 Table Summary of the Total Cost](#_hgnunn3aflpk) 10

[**3. Design and Implementation**](#_8diymbh5v4oc) **11**

[3.1 Libraries](#_cl2jtx3reaad) 11

[3.1.1 Psycopg](#_vfhdxv3nd1ds) 11

[3.1.2 PyQt](#_xy2ts97o6pge) 11

[3.2 Code Structure](#_8vf1pwnmaimb) 11

[3.2.1 preprocessing.py](#_ltde68abl2ql) 11

[3.2.2 annotation.py](#_42kpii7rqgs6) 11

[3.2.3 interface.py](#_bobcqdhb1tr4) 12

[3.2.4 project.py](#_fx8ldkg48160) 12

[**4. Algorithms**](#_i84xpiimvp5w) **13**

[4.1 Algorithm to Generate Annotations](#_jp9uki2qrlco) 13

[4.2 Algorithm to Generate Alternative Query Plans](#_9bhfjh5iuiz4) 13

[**5. Experiments**](#_d4nnd2el43h1) **14**

[**6. Conclusion**](#_c31rm6q5ywe1) **15**

[**Appendix: Instructions to Run Software**](#_bil5l1lyuy0) **16**

[**References**](#_66lzxsq6iu5g) **19**

# 

# 1. Introduction

## 1.1 PostgreSQL

PostgreSQL also known as Postgres is one of the world’s most advanced open source database management systems (DBMS) and the fourth most popular DBMS [1]. PostgreSQL is managed by an open-source community of experienced and well-organised engineers. It is an object-oriented DBMS that is ACID (Atomicity, Consistency, Isolation, Durability) compliant and highly extensible, thus allowing the community to easily add new features and capabilities as the industry evolves. PostgreSQL offers state-of-the-art database solutions which are being extensively used across several industries, including the government, financial services, information technology, pharmaceutical, among several others.

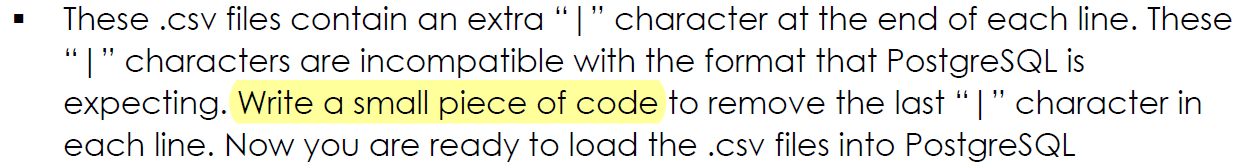
Due to its extensibility, PostgreSQL offers many built-in data types including JSON, XML, HSTORE (key-value), Geo-spatial (PostGIS), IPv6 and features like flexible indexing, composite indexes, and full text search among many others. PostgreSQL runs on all popular operating systems including Linux, Windows, and UNIX-based systems. It also offers library interfaces which allows for seamless integration with most major programming languages including Java, C/C++, PHP, Perl, and Python, among several others. In this project, we leverage Python and PostgreSQL to build an application that retrieves information from a Query Execution Plan (QEP) and Alternate Query Plans (AQPs) to annotate the corresponding SQL query and explain the process behind the execution of various components by the query processor. We also aim to examine the rationality around why certain operators are selected amongst others.

## 1.2 TPC-H Dataset

The TPC-H datasets were used in our project [2]. It serves as a database benchmark to assess the performance of high-complexity decision support databases. It consists of numerous schemas that differ in the amount of data it contains. Hence, TPC-H allows our team to develop and test SQL queries on small datasets of records, before applying those same queries to larger datasets. *Table 1* depicts a summary of the relations in TPC-H, as discussed in Section 1.4.

## 1.3 Data Massaging of TPC-H

As the TPC-H dataset is originally in .tbl format, we were required to first convert them into .csv as stated in the Project Brief.



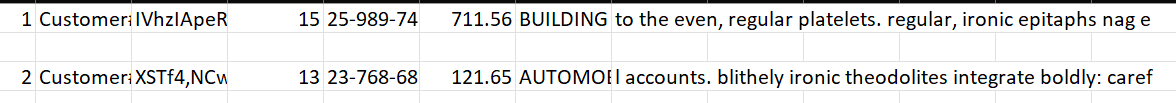
We hence wrote a small piece of code to perform this conversion:

|  |
| --- |
| import csv  import os    def tbl\_to\_csv(filename):  path = os.path.abspath(os.getcwd())  # prevent csv writer from writing new line after every row  csv\_file = open("".join([path, "\\", filename, ".csv"]), "w+", newline='')  tbl\_file = open("".join([path, "\\", filename, ".tbl"]), "r")  lines = tbl\_file.readlines()  for line in lines:  # truncate \n at the end of every line  csv.writer(csv\_file).writerow(line.split('|'))  tbl\_file.close()  csv\_file.close()      if \_\_name\_\_ == "\_\_main\_\_":  files = ['customer', 'lineitem', 'nation', 'orders', 'part', 'partsupp', 'region', 'supplier']  for file in files:  tbl\_to\_csv(file)  break |

By placing the above script into the same folder containing the .tbl datasets and running it, we obtain the converted .csv files. The use of relative paths (os.getcwd()) and filename variables lends the code high flexibility and convenience.

While writing to .csv, care must be taken in handling commas - commas are used in both textual fields and as column delimiters. Making use of the csv library in Python and its .writerow() function allows us to handle this parsing easily. By using line.split(‘|’), we can pass in an array indicating data for a specific row.

Lastly, whitespace must also be handled properly. For one, newline must be set to ‘’ to prevent the csv writer from writing a new line after every row. If this is not done, the .csv will be printed as such:



## 1.4 Data Seeding of TPC-H

Once the .csv files are generated, we generate the tables with their appropriate schema in Postgresql based in Appendix A. Afterward, we run a small .sql script to import the .csv files:

|  |
| --- |
| COPY region  FROM 'C:\%directory%\database\region.csv'  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:

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

|  |  |  |
| --- | --- | --- |
| **Relations** | **Number of Records** | **Description** |
| Region | 5 | The continents that nation belongs in |
| Nation | 25 | A list of supported nations |
| Part | 200000 | Parts sold, with their respective suppliers and prices |
| Supplier | 10000 | Details of the suppliers of parts |
| Partsupp | 800000 | Information from the supplier about the parts, inventory and price |
| Customer | 150000 | A list of all customers details |
| Orders | 1500000 | A list of customer orders, including the order status and price of the order |
| Lineitem | 1048576 | Information of all order items of each order |

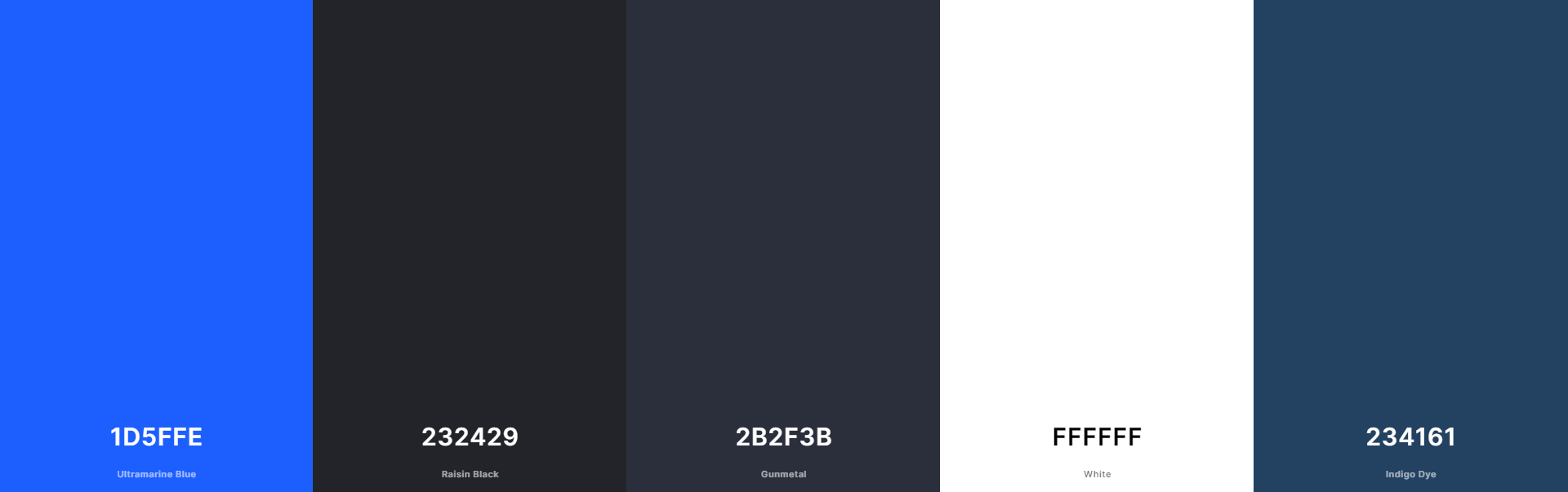
# 2. Graphical User Interface (GUI)

Before discussing the functionalities offered by our main GUI application (section 2.3), the supporting GUIs are described here. They were implemented to support and complement the main GUI to deliver a great user experience.

Ben Shneiderman is an expert in the field of human-machine interaction and has formulated eight golden rules with regards to designing “user-friendly” UI/UX [3]. We applied some of these rules in the design of our application package.

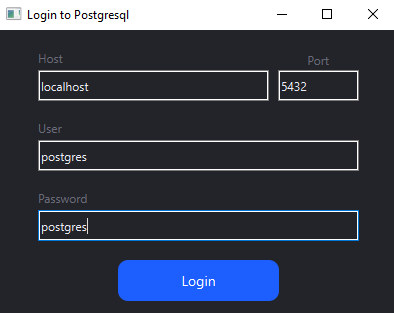
## Colour Scheme

The first rule is to **Strive for consistency**. Standardising the way information is conveyed ensures users are able to apply knowledge from one click to another without the need to learn new representations for the same actions. Consistent colour, layout, capitalisation, fonts, etc should be employed throughout the application. We applied this throughout the entire design of our application, using a single font for all elements and applying a consistent colour scheme.



**Figure 1: Colour palette used in the application**

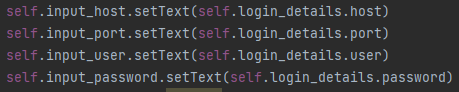
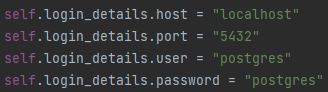
## 2.1 [Generality] Login



**Figure 2: Login screen with default credentials**

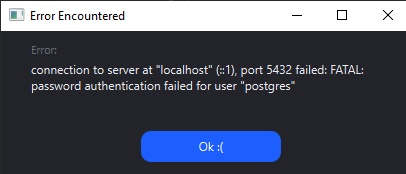
The next rule we applied is **Seek universal usability**, giving rise to our login screen. We opted to design a full login screen as opposed to using values in a json config file. This allows users to be able to key in their username and password credentials easily without having to navigate our codebase. This allows us to cater to all sorts of users—technical and non-technical.

We also applied the principle to **Enable frequent users to use shortcuts** by setting default values for the login information. These are the defaults in Postgresql unless the user has configured them separately, and hence having these values appear by default will enhance user convenience.



**Figure 3: Code snippet of default login credentials**

## 2.2 Error Handling

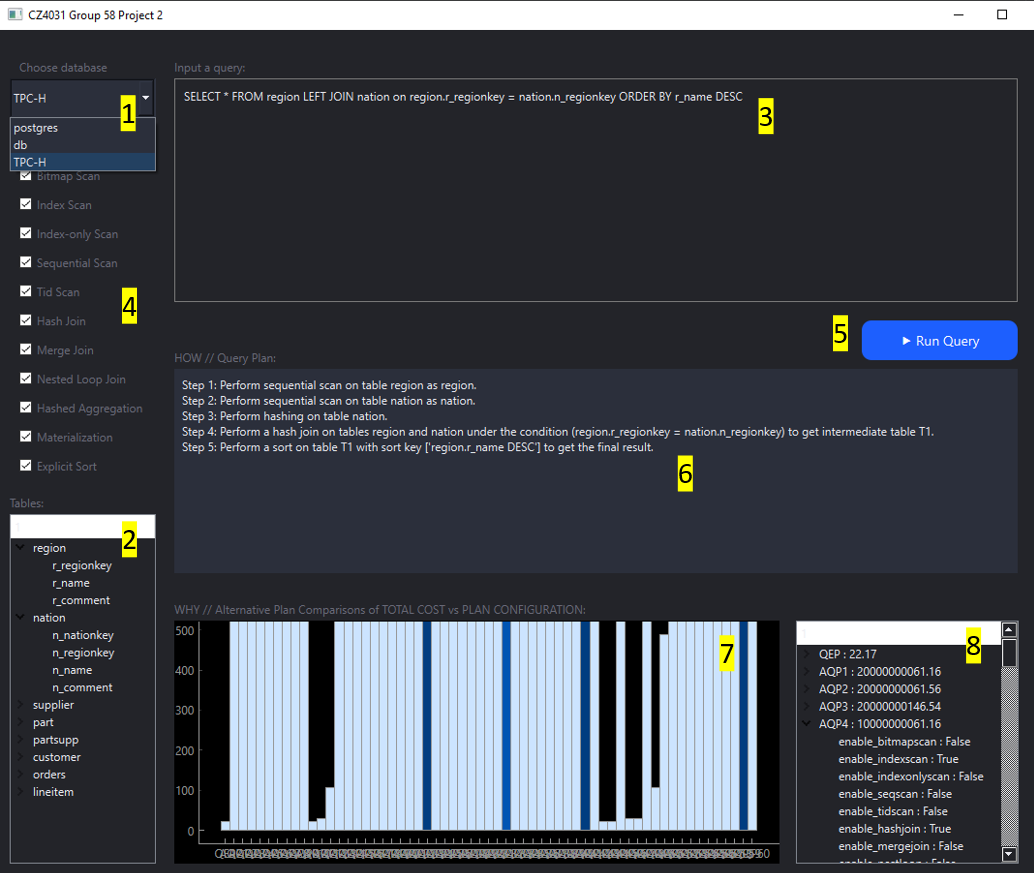


**Figure 4: Error message shown to user for invalid credentials**

We also **Design dialogs to yield closure**. In the event of a login fail, an error popup will appear letting the user know of the error and its corresponding error message.

## 2.3 Main Application

If login is successful, users will be brought to our main application screen.

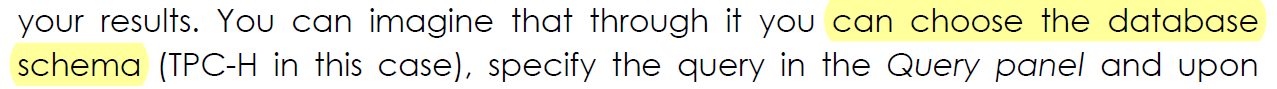


**Figure 5: Main GUI of the application**

For each of the numbered labels 1 through 8, we discuss them in their respective sections 2.3.1 through 2.3.8.

### 2.3.1 [Generality] Database Selection

The Project Brief has a requirement allowing the user to choose the database schema.



**Figure 6: Screenshot of project brief highlighting the requirement for choice of database schema**

To ensure generality of our solution, our application displays all the schemas present in the user’s Postgresql database and allows them to choose which database to connect to via a dropdown selection. We retrieve all database schemas present using the following query:

|  |
| --- |
| "SELECT datname FROM pg\_database WHERE datistemplate = false;" |

In this scenario, the user selects the database named TPC-H among a list of 3 databases. Details of the TPC-H database have been discussed in Section 1.2

### 2.3.2 Tables and Columns in Database

Upon the selection of the database, a connection to said database will be established using the Psycopg database adaptor, discussed in Section 3.1.1. We then display all the tables present in that particular database using the following query:

|  |
| --- |
| "SELECT table\_name FROM information\_schema.tables WHERE table\_schema='public' AND table\_type='BASE TABLE';" |

For each table, we also use the following query to display all columns in that table:

|  |
| --- |
| f"SELECT column\_name FROM information\_schema.columns WHERE table\_name = '{table}' AND table\_catalog = '{db}';" |

The result is a tree directory that lists the tables and columns present in the database.

In this scenario, the 8 tables in the TPC-H database are displayed and this corresponds to *Table 1* discussed in Section 1.4.

### 2.3.3 Query Input

The user is allowed to input their own queries to perform testing. As shown in Figure 5, the query written was “SELECT \* FROM region LEFT JOIN nation on region.r\_regionkey = nation.n\_regionkey ORDER BY r\_name DESC”.

### 2.3.4 Selection of Parameters

The user can select the parameters to configure how the alternative plans are generated by Postgresql. The selection of parameters was created with checkboxes for different node types for easy selection. The available node type selection includes the following:

* Scan: Bitmap Scan, Index Scan, Index-only Scan, Sequential Scan, Tid Scan
* Join: Hash Join, Merge Join, Nested Loop Join
* Others: Hash Aggregation, Materialization, Explicit Sort

### 2.3.5 Execution of Query

After the input and selection of parameters, the user can proceed to execute the query by clicking the “Run Query” button.

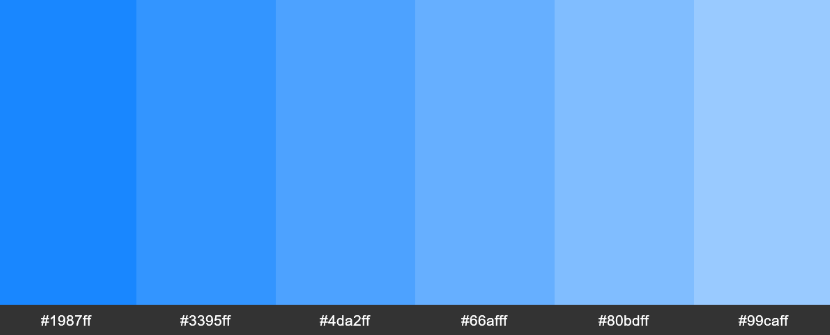
### 2.3.6 Display of Query Plan

Upon the execution of the query, the query plan will be generated. A detailed annotation of the steps of the optimal query plan will be displayed to the user. The optimal query plan was generated and processed by Postgresql. It describes the steps taken by the query processor in easy-to-understand natural language. The explanation includes the conditions of scan, join, grouping functions, etc.

### 2.3.7 [Generality] Visualisation of the Total Cost against the Plan Configuration

Upon the execution of the query, the visualisation of the total cost against the plan configuration was generated. It provides an easy graphical view for the user to compare the different costs of the optimal query plan, and the alternate query plans.

The intensity of the shade of blue in the bar graphs are used to indicate their values. Lighter values indicate lower values and vice versa. A snippet of our colour scheme used is shown below.



Our application calculates the range of costs present in the data and partitions the costs into colour bins. The number of bins is calculated based on the length of the COLOUR\_PALETTE array. The colour scheme is hence able to work for any range of data supplied to the graph and for any number of data points.

### 2.3.8 Table Summary of the Total Cost

The table of summary also provides a clear indication of the total cost in case the visualisation of total cost against plan configurations is difficult to comprehend.

Each row can be expanded upon to view the configurations passed into Postgresql that resulted in that total cost.

# 3. Design and Implementation

## 3.1 Libraries

We used two main libraries in Python to assist in the development of our application—Psycopg for connecting to Postgresql in Python, and PyQt for the GUI.

### 3.1.1 Psycopg

Psycopg is the most popular PostgreSQL database adapter for Python [5]. It includes the complete implementation of Python DB API 2.0 specification and features thread safety—several threads can share the same connection. It was designed for multi-threaded applications which involve a large amount of concurrent “INSERT”s and “UPDATE”s. Psycopg is primarily implemented in C as a libq wrapper—hence ensuring efficiency and security. It features asynchronous communication, notifications, and client-side & server-side cursors.

### 3.1.2 PyQt

PyQT is a set of Python bindings for Qt, which is a set of C++ libraries and development tools which allows users to build graphical user interfaces (GUIs) [5]. It includes abstractions of network sockets, threads, Unicode, regular expressions, SQL databases, among many others. It contains a rich collection of GUI widgets which enables quick development of user-friendly interfaces. It is compatible with Windows, Unix, Linux, macOS, iOS, and Android. Hence, it seems like an attractive option for developing multi-platform applications that have a native design on each platform. Other GUI toolkits in Python include Tkinter, wxPython, PySide and a few others.

## 3.2 Code Structure

We adopt the Model-View-Controller (MVC) architecture when architecting our codebase. This complements well with the existing constraints of needing 4 separate Python files for this project, and this is elaborated further below.

We also applied clean code principles and made use of typing to enhance type-hinting and facilitate collaboration between developers.

### 3.2.1 preprocessing.py

preprocessing.py is the model of the application. It contains the database connector and is in-charge of all functions that require a connection to our database. It also contains the miscellaneous data structures used in the application, such as the Login Details and Query Info data types. Since this file handles database connections, it also contains a method for permuting the various joins and scans on the query.

### 3.2.2 annotation.py

annotation.py is also the model of the application containing the source code to annotate the SQL query. This gives the user insight into the query execution process and the steps involved in a query plan.

### 3.2.3 interface.py

interface.py is the view of the application and contains code implementing the UI components of the application for login, error, and the main page. The main page displays our query annotation and a graph of the costs of the AQPs compared to the QEP.

### 3.2.4 project.py

project.py is the controller of the application. It is the primary file which bridges all components together and acts as the central messenger facilitating flow of information between the different parts of the application.

For example, if the user triggers an event on the view (interface.py), data will be passed from the view to the controller (project.py). The controller will then invoke the necessary functions in the model (preprocessing.py or annotation.py) to obtain the required information, parse it, and send it back to the view (interface.py) to display to the user.

Applying the MVC architecture results in loose coupling and high cohesion among these 4 Python files, promoting maintainability and understandability in the flow of the application.

# 4. Algorithms

The following is the algorithm to generate annotation and alternative query plans.

## 4.1 Algorithm to Generate Annotations

The generation of annotations is matched with the string names. This algorithm consists of else if statements to do a search to match a specific query plan that is parsed. Once the node type is found, a message will be returned containing natural language text for the user to understand the annotation. This will be repeated until all of the node types relating to the query are matched. The final output of this algorithm is a concatenated message of the steps involved in the specific query plan that is parsed.

## 4.2 Algorithm to Generate Alternative Query Plans

The generation of alternative query plans requires permutation to all the select parameters, node types, from the user interface. As explained above, the available node types are split into 3 main categories namely Scan, Join, and Others.

* Scan consists of Bitmap Scan, Index Scan, Index-only Scan, Sequential Scan and Tid Scan.
* Join consists of Hash Join, Merge Join and Nested Loop Join.
* Others include Hash Aggregation, Materialization, Explicit Sort.

The following are the steps of implementations. For simplified explanation, assume the user has inputted Bitmap Scan, Index Scan, Merge Join and Nested Loop Join.

1. Based on user selection of parameter types, the input is read and further grouped into their categories list. The scan and join list will then be added together, and append the 3 node types into the list.
2. Filtering is done to generate the list of parameters to be permuted. Our approach is that the scan and join list will always change due to the different nodes in the same categories. However, as the 3 nodes (Hash Aggregation, Materialization, Explicit Sort) are not entirely in the same categories, we will always add them into the list of parameters to be permuted.
3. In an event where more than one scan or join is selected, the parameters will be permuted accordingly to generate all the possible lists to execute the AQP query.

|  |
| --- |
| [('Bitmap Scan', 'Merge Join', 'Hashed Aggregation', 'Materialization', 'Explicit Sort'), ('Bitmap Scan', 'Nested Loop Join', 'Hashed Aggregation', 'Materialization', 'Explicit Sort'), ('Index Scan', 'Merge Join', 'Hashed Aggregation', 'Materialization', 'Explicit Sort'), ('Index Scan', 'Nested Loop Join', 'Hashed Aggregation', 'Materialization', 'Explicit Sort')] |

**Figure 7: List of Parameters to be Permuted**

|  |
| --- |
| [  {'enable\_bitmapscan': True, 'enable\_indexscan': False, 'enable\_indexonlyscan': False, 'enable\_seqscan': False, 'enable\_tidscan': False, 'enable\_hashjoin': False, 'enable\_mergejoin': True, 'enable\_nestloop': False, 'enable\_hashagg': True, 'enable\_material': True, 'enable\_sort': True},  {'enable\_bitmapscan': True, 'enable\_indexscan': False, 'enable\_indexonlyscan': False, 'enable\_seqscan': False, 'enable\_tidscan': False, 'enable\_hashjoin': False, 'enable\_mergejoin': False, 'enable\_nestloop': True, 'enable\_hashagg': True, 'enable\_material': True, 'enable\_sort': True},  {'enable\_bitmapscan': False, 'enable\_indexscan': True, 'enable\_indexonlyscan': False, 'enable\_seqscan': False, 'enable\_tidscan': False, 'enable\_hashjoin': False, 'enable\_mergejoin': True, 'enable\_nestloop': False, 'enable\_hashagg': True, 'enable\_material': True, 'enable\_sort': True},  {'enable\_bitmapscan': False, 'enable\_indexscan': True, 'enable\_indexonlyscan': False, 'enable\_seqscan': False, 'enable\_tidscan': False, 'enable\_hashjoin': False, 'enable\_mergejoin': False, 'enable\_nestloop': True, 'enable\_hashagg': True, 'enable\_material': True, 'enable\_sort': True}  ] |

**Figure 8: Generated AQP Query**

# 5. Experiments

After conducting an experiment to test the program, Figure 9 depicts the output of our program. The following are the selections made to perform this experiment.

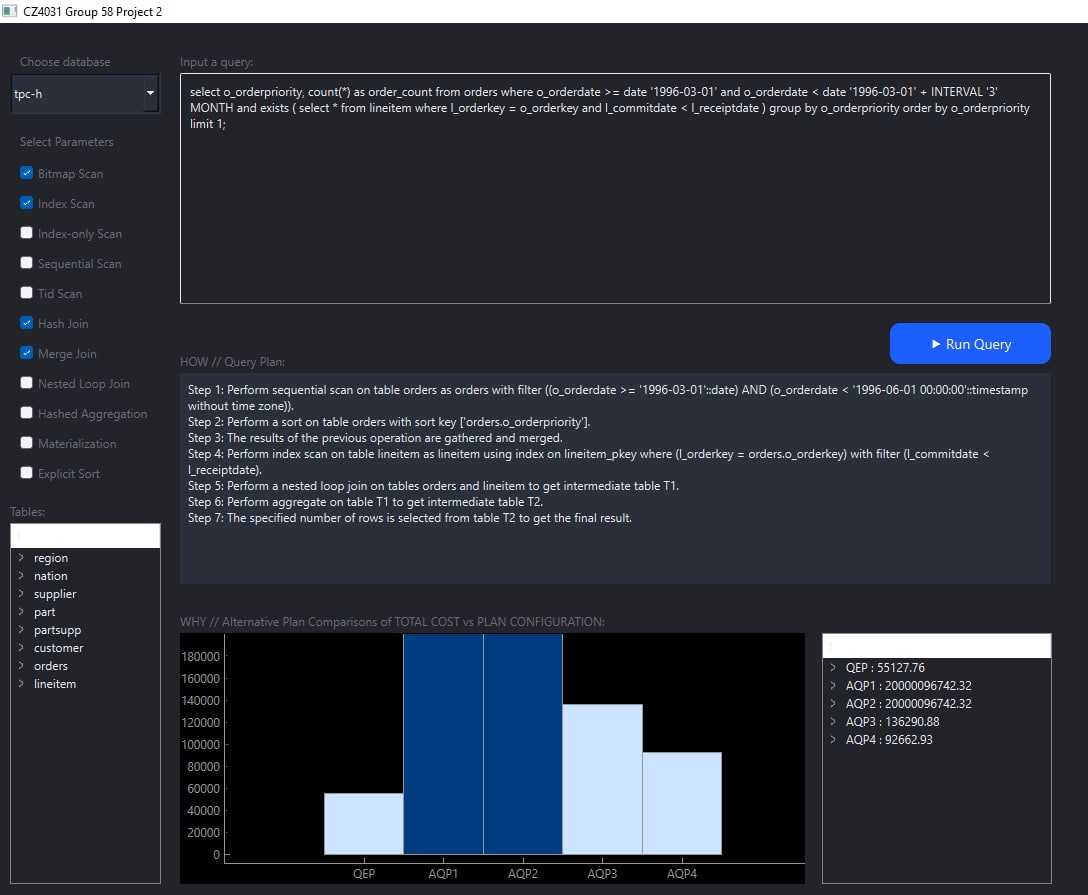
1. Select ‘TPC-H” as the database schema.
2. Input the query:

* select o\_orderpriority, count(\*) as order\_count from orders where o\_orderdate >= date '1996-03-01' and o\_orderdate < date '1996-03-01' + INTERVAL '3' MONTH and exists ( select \* from lineitem where l\_orderkey = o\_orderkey and l\_commitdate < l\_receiptdate ) group by o\_orderpriority order by o\_orderpriority limit 1;

1. Select the parameters to configure:

* Bitmap Scan, Index Scan, Hash Join and Merge Join

Figure 9 illustrates after the execution of step 1 to 4, the application generated the detailed optimal query plan explanation, and it can be observed that this query execution has produced 4 alternate query plans. With sharp observation, we can see that QEP have a total cost of 55127.16, while the remaining AQP1, AQP2, AQP3 and AQP4 have a total cost of 20000096742.32, 20000096742.32, 136290.88 and 92662.93. This evident that QEP indeed is the optimal query plan for the query execution, as it has the lowest total cost among the other 4 AQPs.



**Figure 9: GUI Output of the Experimen**t

# 

# 6. Conclusion

In the real-world, users write SQL queries to gain insights by searching relational databases. Typically, these users treat the DBMS as a “black box” without understanding the rationale behind the query execution process. Hence, there has been a disconnect between an SQL query and its query plan-related information. In this project, we created an application that could bridge this gap by providing users with explanations for various components of the query processor and the process for selecting certain operators over others. We also retrieved alternate query plans considered by the query processor and compared its cost to the most optimal query plan—QEP. We built an application by integrating a user-friendly GUI with an efficient algorithm to retrieve the QEP and AQPs, and output an annotated SQL query that describes the execution of various components of the query in natural language.

This project not only taught us the technical challenges of building such an application but also highlighted the importance of uncovering the “black box” of a query processor as the need for explainability in the field of computing is more prevalent than ever.

# Appendix: Instructions to Run Software

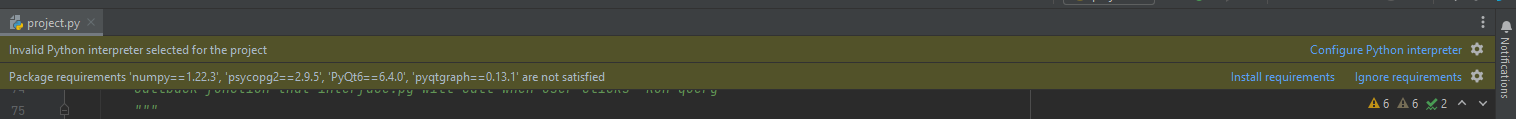
1. Make sure there is a requirements.txt file in the same folder as the 4 python files containing the following text:

|  |
| --- |
| numpy==1.22.3  psycopg2==2.9.5  PyQt6==6.4.0  pyqtgraph==0.13.1 |

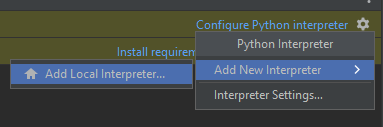
1. Open in Pycharm



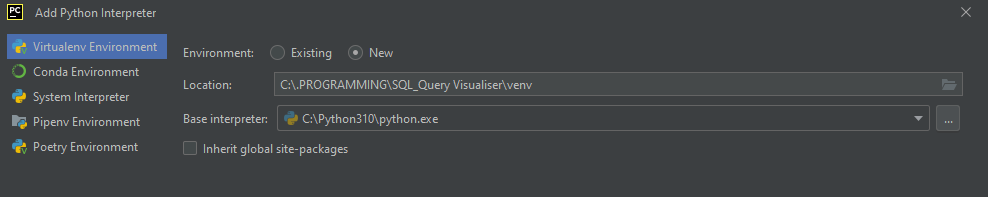
1. Two warnings should appear. Click Configure Python Interpreter



1. Click Add New Interpreter > Add Local Interpreter



1. Select Virtualenv Environment, location is current directory \venv and base interpreter is python.exe



1. Select Ok and venv folder should be created



1. Wait for Indexing to complete



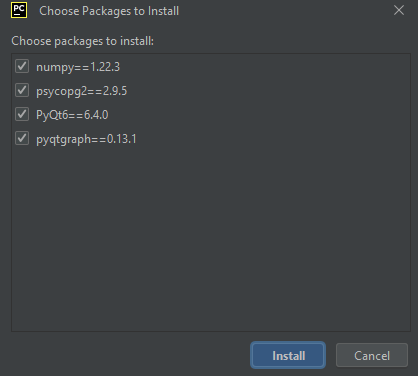
1. One of the warnings from step 2 should disappear, leaving 1 warning



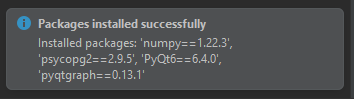
1. Click on Install requirements



1. Install all requirements



1. Wait for installation to finish



1. Click on run project



1. If the above does not exist, go to project.py and click on the run button beside if \_\_name\_\_ == ‘\_\_main\_\_’



# 

# References

[1] <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>

[2] <https://www.tpc.org/tpch/>  
  
[3] Ben Shneiderman’s Eight Golden Rules of UI/UX

[www.interaction-design.org/literature/article/shneiderman-s-eight-golden-rules-will-help-you-design-better-interfaces](http://www.interaction-design.org/literature/article/shneiderman-s-eight-golden-rules-will-help-you-design-better-interfaces)

[4] <https://www.cs.umd.edu/~ben/goldenrules.html>

[5] <https://www.psycopg.org>

[6] <https://riverbankcomputing.com/software/pyqt/>