

**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

23S1 SC3020/CZ4031 Database Systems Principles

Lab Project 2

What Is Your Query Accessing And At What Cost?

Submitted By:

Joanne Christina Salimin (U21020304J)

Priscilla Celine Setiawan (U2123732G)

Tiwana Teg Singh (U2122816B)

Muhammad Rafi Adzikra Sujai (U2120731G)

Submission Date:

17 November 2023

Programming Language Utilized:

ReactJS (Frontend)

Python (with Flask) (Backend)

Github Link:

<https://github.com/tej172/PROJECT-2-DATABASE-SYSTEM-PRINCIPLES>

Total Page Count: 24 pages (excluding cover page and references)

Table of Contents

Table of Contents.....	1
1. Introduction.....	2
1.1. Project Overview.....	2
1.2. Project Structure.....	2
1.2.1. Dataset Used (TPC-H).....	3
1.2.2. PostgreSQL (DBMS used).....	4
1.2.3. ReactJS (Frontend).....	5
1.2.4. Python (with Flask) (Backend & Middleware).....	6
1.3. Installation guide.....	7
1.3.1. Installing Prerequisites.....	7
1.3.2. Project Setup & Running the Project.....	7
1.3.3. Viewing Project Results/Usage.....	9
2. Project Methodology.....	10
3. Project Implementation.....	11
3.1. Input an SQL query.....	11
3.2. Creating a visual exploration framework.....	14
3.2.1. Visualization of disk blocks accessed by the query.....	14
3.2.2. Exploration of the content of these blocks interactively.....	17
3.2.3. Visualizing different aspects of the QEP:.....	19
3.2.3.1. Costs.....	19
3.2.3.2. Actions performed on the table & Sequence of Actions.....	19
3.2.4. User-Friendly, Graphical User Interface (GUI).....	20
3.3. Limitations Of Software.....	22
4. Sample Queries (with Results).....	23
4.1.1. Sample Query 1.....	23
4.1.2. Sample Query 2.....	24
5. References.....	25

1. Introduction

In our journey in the Database System Principles course so far, especially after the Query Execution Plans (QEP) TEL videos, our group has seen how the execution of SQL queries conventionally still remains obscured to most normal users. This essentially hinders these groups of users, which was us previously, from comprehensively understanding what happens "under the hood" of a relational Database Management System (DBMS), like PostgreSQL. Thus, our aim is that with this project, our group can better address this informational gap by helping to design and develop, from the ground up, a sophisticated software tool by helping visualise the different parts of the QEP and disk blocks accessed. It will serve as a window into the complex world of query execution within a relational DBMS, bridging the information gap.

By focusing on PostgreSQL as the chosen DBMS and utilizing the TPC-H dataset for practical simulations and benchmarking as stipulated in the project requirements, our software seeks to empower users with a more nuanced perspective on the intricacies of database interactions and help democratise this information so anyone can run it and learn more in-depth for themselves. We do not solely just depend on the robust capabilities of our backend and middleware made using Flask, a micro web framework written using Python, to handle query processing and database interaction, but also our Frontend too. Our software's frontend, the graphical user interface (GUI), is made using ReactJS not just for the functionality but also for a more intuitive and immersive user experience. The combination of the two helps us more comprehensively explore SQL query execution, making it both accessible to non-technical and technical users.

Thus, our report helps to unfold the narrative of designing, implementing, and utilising this software, providing extensive insights into its functionalities, potential applications and the limitations of the software.

1.1. Project Overview

Essentially our project aims to create an interactive data visualization tool and framework for comprehending the execution plans of SQL queries. Leveraging on PostgreSQL as the underlying DBMS and the TPC-H dataset for the dummy data, our group's software orchestrates a cohesive interaction between various components. These components include a graphical user interface (GUI), coded in ReactJS (./frontend), a robust exploration module handling query processing and visualization, and the main execution orchestrator that was built using Flask (./api). This versatile and multi-dimensional approach ensures that any users of our software have a more holistic understanding of the QEPs.

1.2. Project Structure

Our project is pre-planned and designed with a clear and organized structure in mind, predominantly sub-divided into four primary packages, each serving their own distinct purpose:

1.2.1. Dataset Used (TPC-H)

The Transaction Processing Performance Council Benchmark H (TPC-H) is a industry-wide benchmark designed to determine and access performances of relational Database Management Systems (DBMS) compared to other different systems. They are benchmarked on various kinds of decision support queries, consisting of a suite of business-oriented ad hoc queries, that are used in complex queries and for data analysis in the business analyst environment.

Similar to how the TPC-H is widely used in industry, our group will also be using the TPC-H as dummy data to execute the performance and testing of our software to ensure a good generality of our solution, ensuring it has the capability to manage different types of query plans across various database instances (Snowflake, 2023). Below is the general overview of the TPC-H Standard Specifications as well as it's schema and cardinality ($SF * n$ number of tuples):

TPC-H Relations	Description
NATION	Includes tuples of all supported nations and their respective nationkeys.
REGION	Includes tuples of all continents (regions) of the supported nations and their respective regionkeys.
CUSTOMER	Includes tuples of all customers, including their details, other information and their respective custkeys.
ORDERS	Includes tuples of all orders made by customers, including their order details and their respective orderkeys.
LINEITEM	Includes tuples of all the respective order items made within each order, including their price information, shipping information, quantity and their respective orderkeys, partkeys and suppkeys.
SUPPLIER	Includes tuples of all relevant details about the parts suppliers, like their nation, name, addresses, and their respective suppkeys.
PART	Includes tuples of all parts sold, including their details, prices and their respective partkeys.
PARTSUPP	Includes tuples of all information from the part and supplier table about the part, including current available units, supplier costs and their respective partkeys and suppkeys.

Figure 1: A general overview and description of the tables found within TPC-H (Transaction Processing Performance Council , 2014).

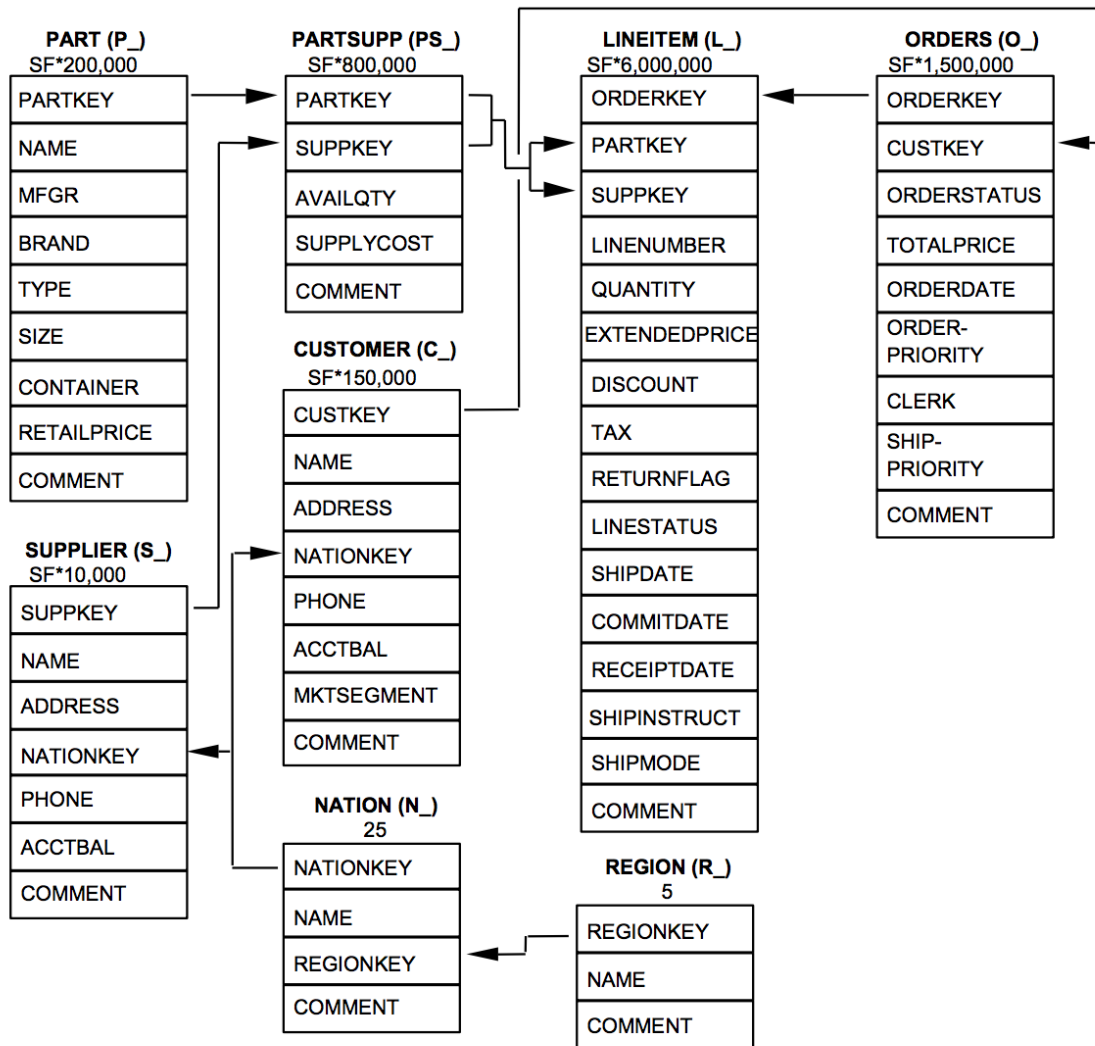


Figure 2: The schema of the TPC-H Benchmark (Transaction Processing Performance Council , 2014).

1.2.2. PostgreSQL (DBMS used)

PostgreSQL is a widely-used, open-source relational DBMS that adheres to SQL standards and is renowned for both its durability and extensibility. Under this, its capability to create and view query execution plans is one of its most notable features that we incorporated in our Software too. The query planner and executor, which is an internal part of PostgreSQL, evaluates SQL queries and creates a plan for their execution. This plan generally describes the sequence of operations, access techniques, and possible optimizations that the database will employ to efficiently execute the query (PostgreSQL Global Development Group, 2023). Thus, we chose PostgreSQL as our DBMS due to its transparency and comprehensive toolset needed to do our project.

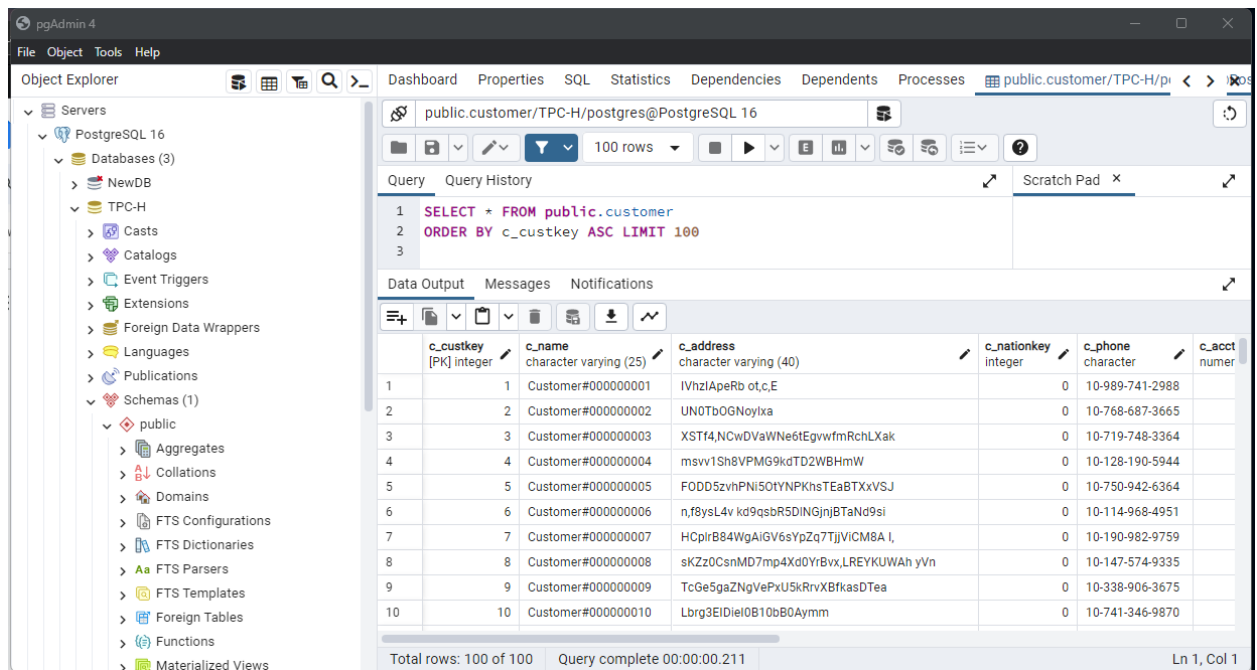


Figure 3: A snapshot of pgAdmin 4 application that we used to access the TPC-H Benchmark (under Databases(3)) using PostgreSQL. pgAdmin4 is a feature rich Open Source development and admin platform for PostgreSQL

1.2.3. ReactJS (Frontend)

ReactJS, a powerful open-source JavaScript library for building interfaces and UIs, is very suitable for our project which requires us to dynamically visualise various query execution plans. Its modular design and use of a Virtual DOM allows our group to quickly render components that can be put together to create an interactive frontend that displays PostgreSQL query execution plans in an easy-to-understand way for both technical and non-technical users.

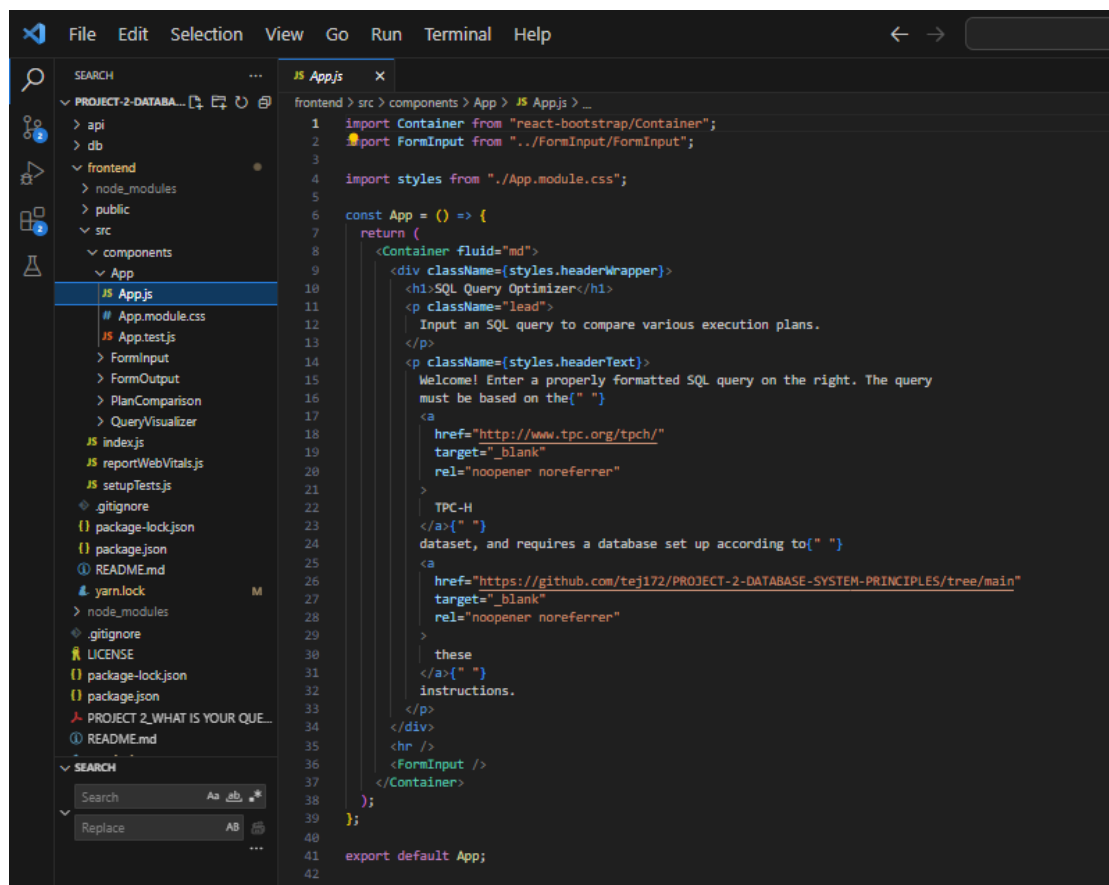
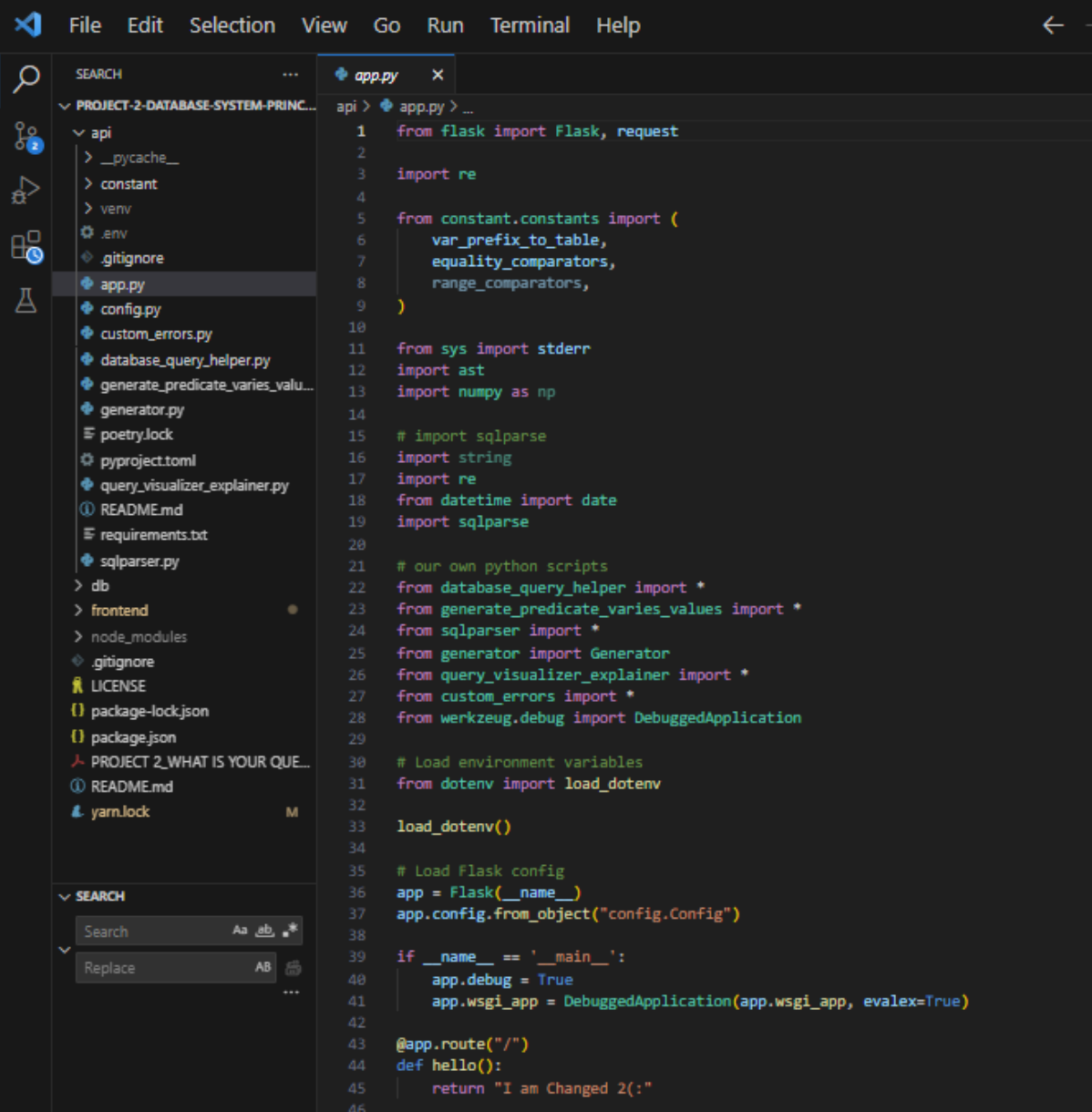


Figure 4: A snippet of our Frontend ReactJS application that is found under the `./frontend` folder of our project.

1.2.4. Python (with Flask) (Backend & Middleware)

We built our backend using python and more specifically leveraged on Flask to help us communicate between the Frontend and the DBMS. Flask, a lightweight web framework, serves as both the backend and middleware for our project visualizing query execution plans. Flask's simplicity and flexibility make it an excellent choice for handling server-side operations for this project. Given this, it also enables seamless integration with PostgreSQL and React, allowing efficient communication between the frontend and the database.



```
1  from flask import Flask, request
2
3  import re
4
5  from constant.constants import (
6      var_prefix_to_table,
7      equality_comparators,
8      range_comparators,
9  )
10
11 from sys import stderr
12 import ast
13 import numpy as np
14
15 # import sqlparse
16 import string
17 import re
18 from datetime import date
19 import sqlparse
20
21 # our own python scripts
22 from database_query_helper import *
23 from generate_predicate_varies_values import *
24 from sqlparser import *
25 from generator import Generator
26 from query_visualizer_explainer import *
27 from custom_errors import *
28 from werkzeug.debug import DebuggedApplication
29
30 # Load environment variables
31 from dotenv import load_dotenv
32
33 load_dotenv()
34
35 # Load Flask config
36 app = Flask(__name__)
37 app.config.from_object("config.Config")
38
39 if __name__ == '__main__':
40     app.debug = True
41     app.wsgi_app = DebuggedApplication(app.wsgi_app, evalex=True)
42
43 @app.route("/")
44 def hello():
45     return "I am Changed 2(:"
46
```

Figure 5: A snippet of our Flask server that is found under the `./api` folder of our project.

1.3. Installation guide

The installation guide provides a step-by-step guide to installing and running our software. The guide covers the following 4 topics:

1.3.1. Installing Prerequisites

Before we dive into the actual installation process, ensure your system meets the following prerequisites:

1. Ensure you have Node.js and npm (Node Package Manager) installed on your system. These are the following minimum version requirements needed to run the project:
 - Node.js: Minimum version v16.16.0
 - npm: Minimum version v8.11.0
2. Ensure you have python and poetry installed on your system.
 - For installing Poetry, use the following command: **pip install poetry**
3. Verify you have a respective compatible Integrated Development Environment (IDE) installed:
 - For Python: Choose a compatible IDE such as **VSCode (recommended)**, PyCharm or Jupyter Notebook.
 - For ReactJS: Select an IDE suitable for React, like **VSCode (recommended)**, Sublime Text or Atom.

1.3.2. Project Setup & Running the Project

You can download the zip version or clone the source code of the project from the [GitHub repository](#) listed at the top of the report.

1. Setting Up Python Virtual Environment:
 - Open your terminal
 - Run the following commands on your terminal:

```
pip3 install --upgrade pip
cd api
pip3 install virtualenv
py -3 -m venv venv
.\venv\Scripts\activate
```
2. Set up the .env file:
 - Navigate to the 'api' folder.
 - Create a file called '.env' inside the api folder.
 - Add the following configuration to the .env file:

```
FLASK_APP=app.py
FLASK_ENV=development
DB_HOST=localhost
```



```
DB_NAME=TPC-H
DB_USER=postgres
DB_PASSWORD=postgres
DB_PORT=5432
```

3. Setting Up and Testing the Backend:

- While still in the 'api' folder of your terminal, run the following commands:

```
pip install poetry
pip install -r requirements.txt
flask run
```

- If the database is running, you should see something like this

```
○ (venv) PS C:\Users\Joanne Christina\Coding Projects\PROJECT-2-DATABASE-SYSTEM-PRINCIPLES\api> flask run
* Serving Flask app 'app.py'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

4. Setting Up and Testing the Frontend:

- Navigate to the project's root folder:

```
cd ..
```

- Move to the 'frontend' folder directory:

```
cd frontend
npm install -f
npm start
```

- Wait a few minutes for the development server to start, and it will automatically open in your default browser.

5. Running the complete project from the root folder:

- Move back to the project's root:

```
cd ..
npm install
npm start
```

- Wait a few minutes for the development server to start, and it will automatically open in your default browser.

Alternative Setup (if initial method of setting up and running does not work):

1. **Alternative:** Setting Up and Testing the Backend:

- In the api directory, create an .env file (if not already created from before) with the same configurations as mentioned earlier.

```
FLASK_APP=app.py
FLASK_ENV=development
DB_HOST=localhost
DB_NAME=TPC-H
DB_USER=postgres
DB_PASSWORD=postgres
DB_PORT=5432
```

- Run the following commands in your terminal:

```
npm install poetry
```

```
poetry install
flask --app app run
```

2. **Alternative:** Setting Up and Testing the Frontend:

- From the project's root folder, navigate to the frontend directory on your terminal:

```
cd frontend
```

- Run the following commands on the frontend directory through your terminal:

```
npm install
npm start
```

- Wait a few minutes for the development server to start, and it will automatically open in your default browser.

1.3.3. Viewing Project Results/Usage

Once both the backend and frontend servers are running, open your default web browser (Google Chrome Recommended) and navigate to the specified URL (usually <http://localhost:3000/>) to interact with the GUI we have created. Explore the visualizations, query results, and delve in-depth into the realm of SQL query execution made accessible through our group's software. We also have made a video on how to use the software, including a functionality guide and walk through (if needed) : <https://youtu.be/ZfDLBhyvj0>

2. Project Methodology

In the course for this project, our objective is to develop an application that addresses the inherent complexity associated in understanding the inner workings of a DBMS when executing an SQL query. The typical user is often clueless about crucial details such as which blocks or pages are accessed, the respective tuples located within those blocks, and the utilization of the buffer.

To shed light on these aspects, our project aims to design and implement a software application that facilitates visual exploration of these disk blocks accessed by a given SQL query. Furthermore, our software also seeks to visualize various features of the corresponding Query Execution Plan (QEP), including cost, what actions is performed on the table and the respective sequence of these actions. The inputs and the outputs of our application are illustrated in the table below:

Software Input	Software Output
An SQL Query	<ul style="list-style-type: none">• Visualization of disk block in each step of QEP
	<ul style="list-style-type: none">• Facilitation exploration in each step of QEP interactively
	<ul style="list-style-type: none">• Visualization different aspects of QEP:<ul style="list-style-type: none">○ What action is performed on the table○ The sequence of actions○ Respective cost & Disk utilisation ratio

Figure 6: A brief tabular overview of the input and output of our Software

The implementation steps of our application is structured into four sequential steps, each contributing to the comprehensive understanding of SQL query execution:

1. Step 1: Parse Input SQL Query
2. Step 2: Generate QEP JSON from the Database
3. Step 3: Make QEP tree node visualization
4. Step 4: Make disk visualization for each tree node

The detailed breakdown of the implementation is provided in the next section of the report: Project Implementation.

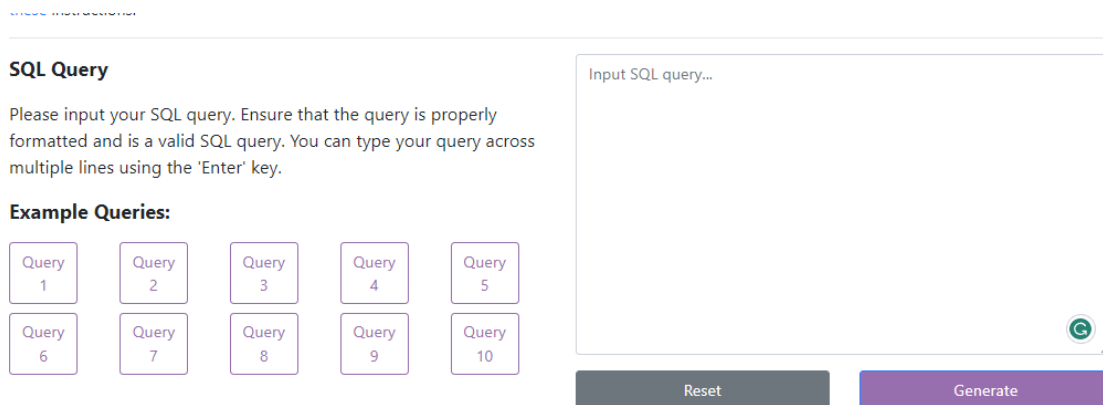
3. Project Implementation

When designing and implementing our software, we tackled each component described in the previous section systematically to ensure that we built a comprehensive and user-friendly tool for visual exploration of SQL query execution. The following sections describe and explain the key algorithms, functionalities, and potential limitations of our application.

3.1. Input an SQL query

In the form, users will input a SQL query and then can click the generate button. They have the option of either clicking one of 8 different example SQL queries, provided by the LANTERN software given to us back in week 9, or can input their own custom SQL queries. These set of provided queries can also be used to test our system and are specifically found in the following file : `./frontend/public/queries.json`.

Once the generate buttons is clicked, the input queries are fetched from the user interface, and thereafter send a 'POST' request to api with path '/generate' to execute in postgresql and get the respective QEP.



The screenshot shows a web interface for entering SQL queries. On the left, under the heading "SQL Query", there is a text box containing the instruction: "Please input your SQL query. Ensure that the query is properly formatted and is a valid SQL query. You can type your query across multiple lines using the 'Enter' key." Below this, a section titled "Example Queries:" displays ten buttons labeled "Query 1" through "Query 10". To the right of these buttons is a large text area with the placeholder text "Input SQL query...". At the bottom right of the text area is a small green circular icon with a white 'G'. Below the text area are two buttons: a grey "Reset" button and a purple "Generate" button.

Figure 7: A brief example of the input fields of our actual software, which includes both example queries and custom input SQL queries that can be provided by the user.

The software input stage also involves the receiving and interpreting the user-provided SQL query. Before sending the query to Postgresql, we needed to do SQL Query Parsing. Using the 'sqlparse' library, which is a non-validating SQL parser for Python, the query undergoes a series of transformations to ensure:

- Proper formatting,
- It is prettified (by adding spaces between operators), and
- Facilitating compatibility with the sqlparse library (to be able to handle relatively low levels of nesting, the majority of SQL query syntax, and multiple conditions).

This step ensures that the SQL query is parsed correctly before moving forward to the next step.

```

input.sql
1  SELECT l_orderkey, l_extendedprice FROM lineitem WHERE l_quantity > 10
2

```

Figure 8: An example of the Input SQL Query before parsing through the sqlparse library.

```

output.sql
1  ✓ SELECT
2      |      l_orderkey,
3      |      l_extendedprice
4  ✓ FROM
5      |      lineitem
6  ✓ WHERE
7      |      l_quantity > 10;
8

```

Figure 9: An example of the Output SQL Query after parsing through the sqlparse library.

```

def create_qep_sql(sql_query):
    try:
        return "EXPLAIN (COSTS, VERBOSE, BUFFERS, FORMAT JSON) " + sql_query
    except CustomError as e:
        raise CustomError(str(e))
    except:
        raise CustomError(
            "Error in create_qep_sql() - Unable to create sql_query string."
        )

```

Figure 10: A snippet of our code that shows one of the examples of the SQL Functions that we implemented. In this python function, we have to add in EXPLAIN(relevant information we wish to retrieve, the the cost and buffers used) in front of the actual query to be able to generate the actual json of the qep plan.

Lastly, in order to connect with our application's user interface to the actual Postgresql server, the psycopg2 python module is used, specifically the function psycopg2.connect() is used to establish the connection. The psycopg2 module is one of the most popular PostgreSQL database adapters for the Python programming language and its ease of use and utility were the reasons we chose it.

For the next few sections, we will be using our sample query 3 as the main sql query example for the next visual exploration framework section.

```

SAMPLE_QUERY_3.sql
1  SELECT
2      l_orderkey,
3      sum(l_extendedprice * (1 - l_discount)) as revenue,
4      o_orderdate,
5      o_shippriority
6  FROM
7      customer,
8      orders,
9      lineitem
10 WHERE
11     c_mktsegment = 'BUILDING'
12     and c_custkey = o_custkey
13     and l_orderkey = o_orderkey
14     and o_totalprice < 50000
15     and l_extendedprice > 1200
16 GROUP BY
17     l_orderkey,
18     o_orderdate,
19     o_shippriority
20 ORDER BY
21     revenue desc,
22     o_orderdate
23

```

Figure 11: This is our SQL sample query 3 that will be used for the visual exploration framework

3.2. Creating a visual exploration framework

Once the QEP JSON is returned back from the Database, we can make the QEP tree node visualization. This will be at the bottom of the page below the SQL input section and will look something like this under the section 'Graphs':

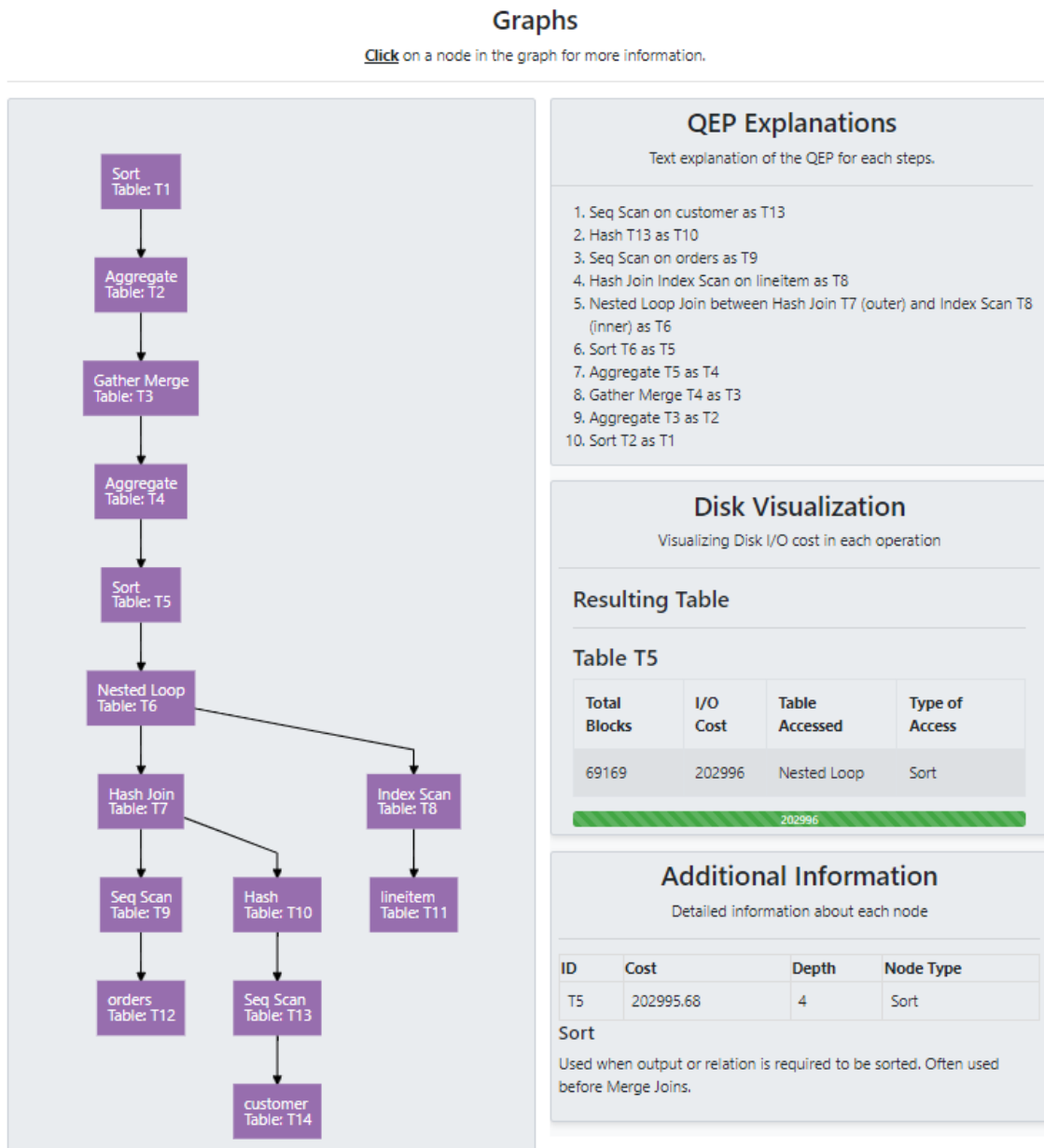


Figure 12: A brief overview of the 'Graphs' output section of our software.

3.2.1. Visualization of disk blocks accessed by the query

To visualize disk blocks, we calculate the total blocks and I/O cost for each table accessed in the SQL query. For sequential scans, the total blocks are approximately equal to I/O cost,

while for index scans, I/O cost is significantly lower. The calculated costs provide insights into the efficiency of each operation in terms of I/O.

As for the Disk Visualization, we provide the users with a visual representation on total blocks of table below in comparison to the I/O cost (i.e. in a ratio):

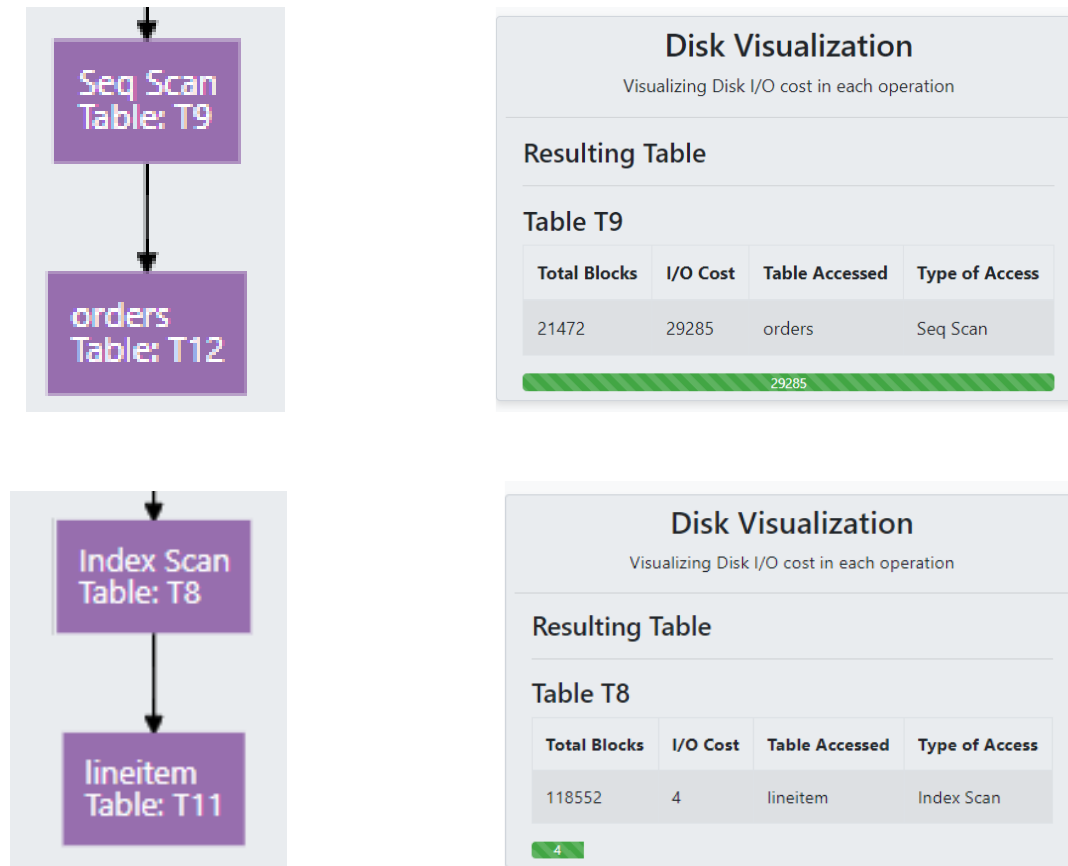


Figure 13: 2 examples of the Disk Visualisation on 2 separate Scans. We can notice how the Disk usage ratio for the Sequential Scan is significantly higher than the Disk usage ratio of the index scan.

For the calculating the disk access view, we need to know the total blocks. Thus, we need to calculate this total blocks value and then store it in a dictionary with key “schema_dict”. For example, for each table we will run the following command in order to calculate their total blocks:

```
SELECT pg_relation_size('{table}') /
current_setting('block_size')::numeric AS block_size_bytes
```

```
1 Results:
2
3 block_size_bytes
4 -----
5 245760
6
```

Figure 14: The command to calculate and store the number of total blocks.

We will also retrieve information on the tables accessed as well as their respective type of access. For example, table T8 is accessing 'lineitem' and Table T9 is accessing 'orders'.

As for the Join relations, the total block count is determined by multiplying the plan_rows obtained from the child node. The plan_rows represent the projected output after performing the previous join operation. Additionally, we enhance the visualization by providing insights into the disk access of the child node, contributing to a more comprehensive understanding of the join operation as shown below:

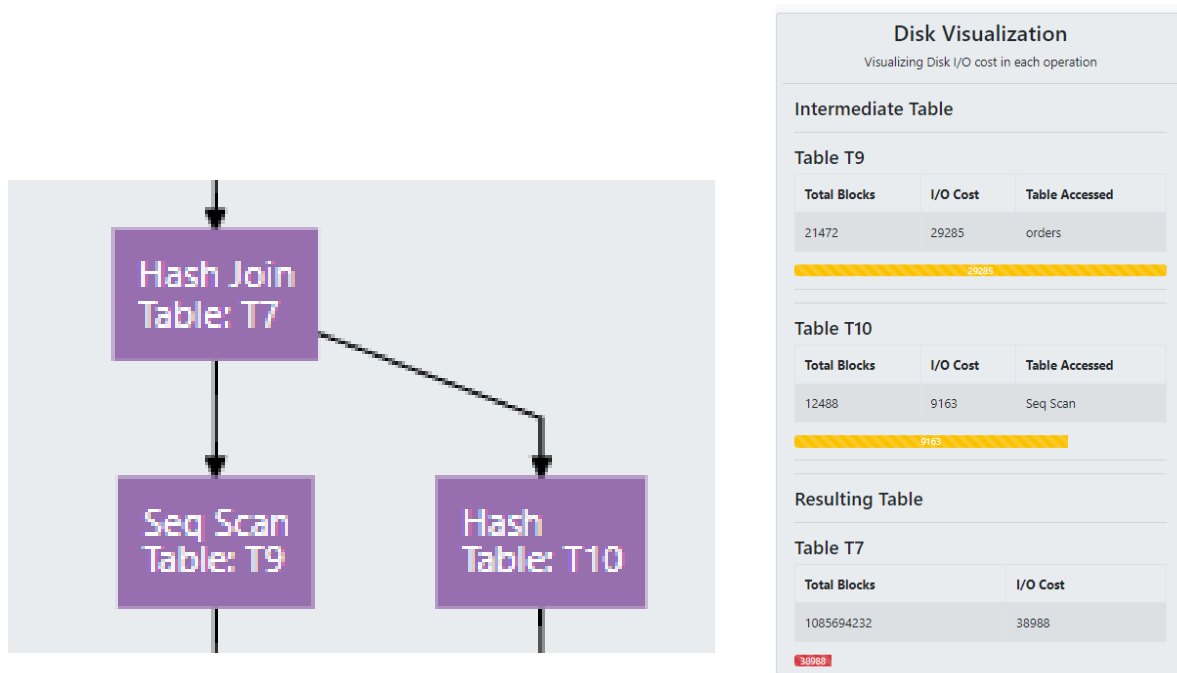


Figure 15: An example of the visualisation of the disk access for the join relations.

Lastly, in the case of other operations like Limit, Aggregate, and Sort, the total block count is inherited from the child node below.

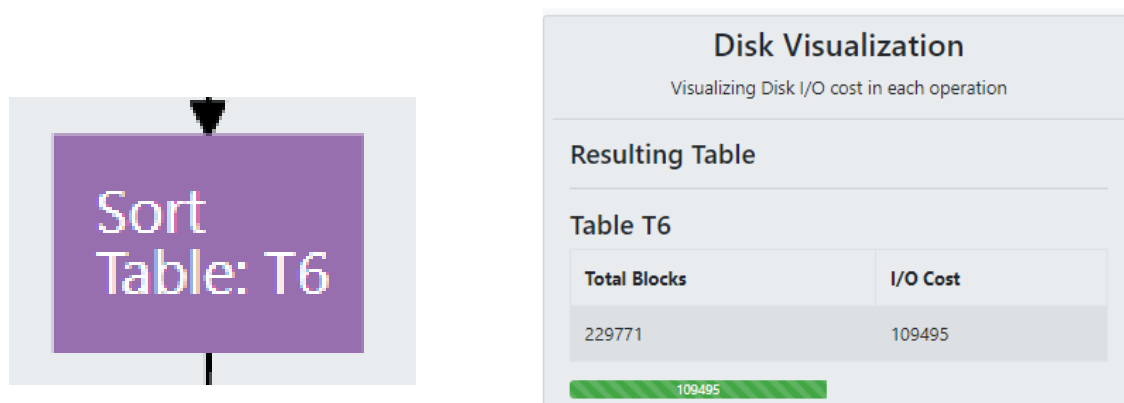


Figure 16: An example of the visualisation of the disk access for the other types of operations.

Thus, this approach ensures that the visualizations accurately reflect the total blocks associated with these operations, allowing for a cohesive representation of the entire Query Execution Plan.

3.2.2. Exploration of the content of these blocks interactively

The Query Execution Plans (QEP) returned to our frontend are visualised using network graphs in sequence of node types returned by the QEP from the root all the way to the leaf nodes. We traverse this structure using the breadth-first-search algorithm. All the corresponding non-leaf nodes will display the types of operations they are executing such as Sort, Joins or Scans as well as its corresponding cost while leaf nodes display the name of the relation accessed.

More specifically for each node in the QEP, the content of the corresponding disk blocks can be interactively explored. Additional information like the depth, node ID, cost, node type, and type of action is also displayed.

For example when we click each node, we retrieve it's respective additional information as well as its disk visualisation. We also provide additional information on what the type of action it is, like explaining what a sequential scan is, briefly explaining what a node scan is, etc. This allows the user to understand the sequential execution of various node actions within the QEP more intuitively.



Additional Information			
Detailed information about each node			
ID	Cost	Depth	Node Type
T9	29284.50	7	Seq Scan
Seq Scan			
This access method is used for retrieval of large portions (approximately > 5-10%) of a relation.			

Figure 17: An example of the Additional Information present when clicking on the Sequential Scan node on Table T9.

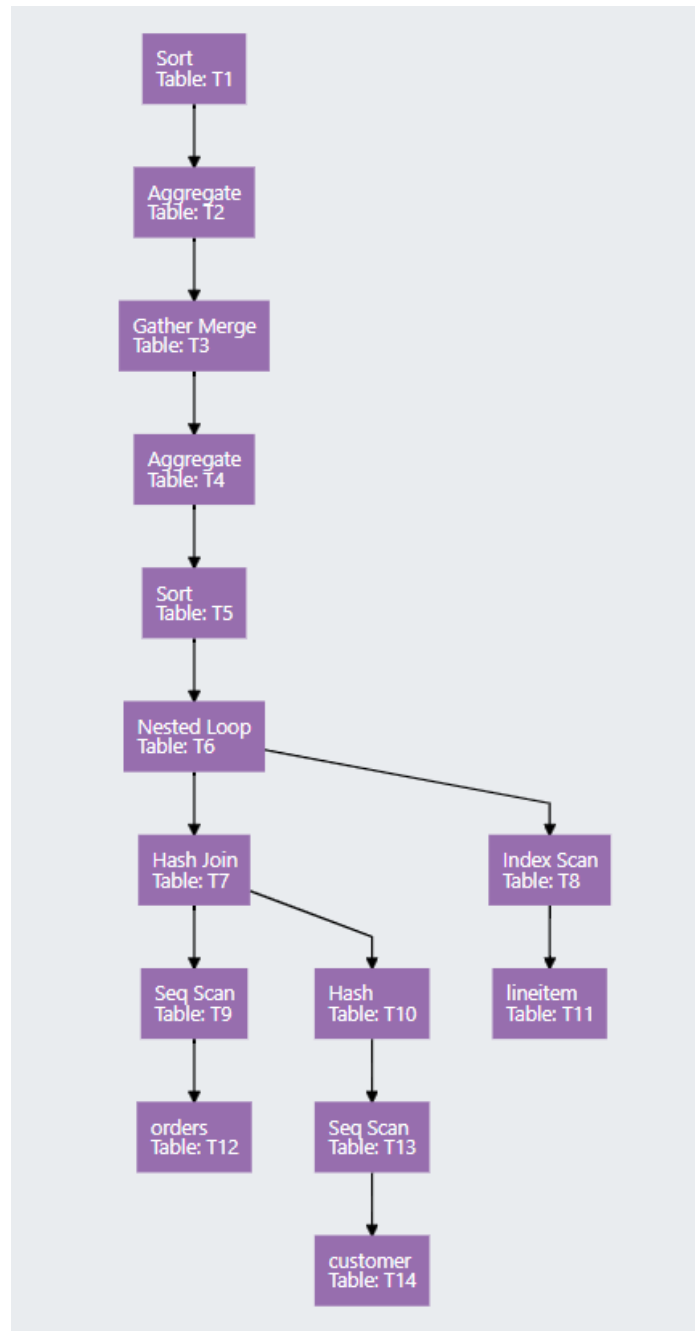


Figure 18: A more general look into the QEP tree created by our software.

3.2.3. Visualizing different aspects of the QEP:

3.2.3.1. Costs

Generally, the main information we have on-hand and available to us are the total number of blocks and the I/O cost. To calculate the I/O cost of each node in the QEP, we based it on the PostgreSQL Explain cost formulas and it is the basis for all our subsequent calculations.

For example, for a node representing a join operation, the cost is calculated based on the cost of its child nodes, considering factors such as `seq_page_cost`, `cpu_filter_cost` and `cpu_tuple_cost`. Let us consider these values for each factor: `seq_page_cost = 1`, `cpu_tuple_cost = 0.01` and `cpu_filter_cost = 0.0025`. The total I/O cost will just be a round up of the total summation of these 3 costs: $Total\ I/O\ cost = 1 + 0.02 + 0.0025 = 1.0225 \approx 2$. Thus for the index scan I/O will be much lower compared to the sequential scan total blocks which will be approximately equal to the I/O. All of this put together, this cost information can be visualized, providing users with insights into the resource utilisation of each operation.

3.2.3.2. Actions performed on the table & Sequence of Actions

The sequence of actions in the QEP is visualised using network graphs and the textual informations under the QEP Explanations. Under the network graphs, the nodes represent different operations, and edges represent the sequence of execution from the root to the leaf nodes. This visualization helps users grasp the flow of operations in the query execution. For example, a non-leaf node labeled with "Sort" indicates that a sorting operation is being performed on the table for that particular step.

Futhermore, in the graph, we can also note that each node has a unique id, and this is to understand the textual explanation provided to users more easily. The text explanation is an explanation of how the relations are accessed and subsequently scanned or joined to produce the output at the end step. The nodes that are referred to will utilise the specific unique ID of the same node within this graphical visualization. This provides a more intuitive method to understanding the sequential execution of the various node actions within the QEP. Essentially, it is another way of viewing the explanation in text form so its easier to follow along for users.

QEP Explanations

Text explanation of the QEP for each steps.

1. Seq Scan on customer as T13
2. Hash T13 as T10
3. Seq Scan on orders as T9
4. Hash Join Index Scan on lineitem as T8
5. Nested Loop Join between Hash Join T7 (outer) and Index Scan T8 (inner) as T6
6. Sort T6 as T5
7. Aggregate T5 as T4
8. Gather Merge T4 as T3
9. Aggregate T3 as T2
10. Sort T2 as T1

Figure 19: An example of the QEP explanations section.

Similar to the actions performed on the table, the sequence of actions in the QEP is also visualized using network graphs. For example, a sequence of nodes in the generated graph could represent a number of scans, followed by a join, and then lastly a sort operation.

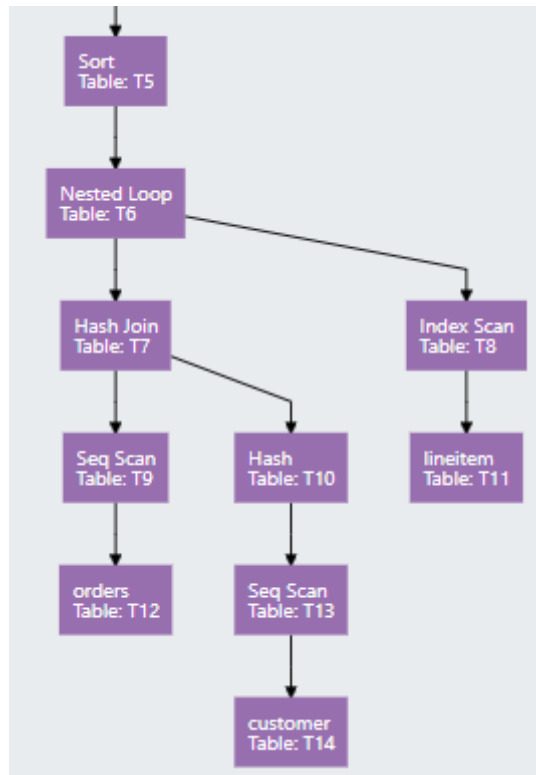


Figure 20: A example of the sequence of of actions performed on the table.

3.2.4. User-Friendly, Graphical User Interface (GUI)

Our group's GUI is designed for a very seamless user experience, prioritising the software's functionality the most over its appearance. Users are simply able to input any SQL query, click "Generate" to go ahead or "Reset" to start over and interactively explore visualisations of that query, specifically of the disk blocks, QEP costs, the sequence of actions and other things. In our case, our frontend features a rather straightforward interface with clear and simple input fields for SQL queries and large buttons for generating visualizations. For example, here is a simple snapshot of our GUI when running the simple SQL input command of "SELECT * FROM region".

SQL Query Optimizer

Input an SQL query to compare various execution plans.

Welcome! Enter a properly formatted SQL query on the right. The query must be based on the [TPC-H](#) dataset, and requires a database set up according to [these](#) instructions.

SQL Query

Please input your SQL query. Ensure that the query is properly formatted and is a valid SQL query. You can type your query across multiple lines using the 'Enter' key.

Example Queries:

Query 1

Query 2

Query 3

Query 4

Query 5

Query 6

Query 7

Query 8

Query 9

Query 10

```
SELECT *  
FROM region
```

Reset

Generate

Graphs

[Click](#) on a node in the graph for more information.

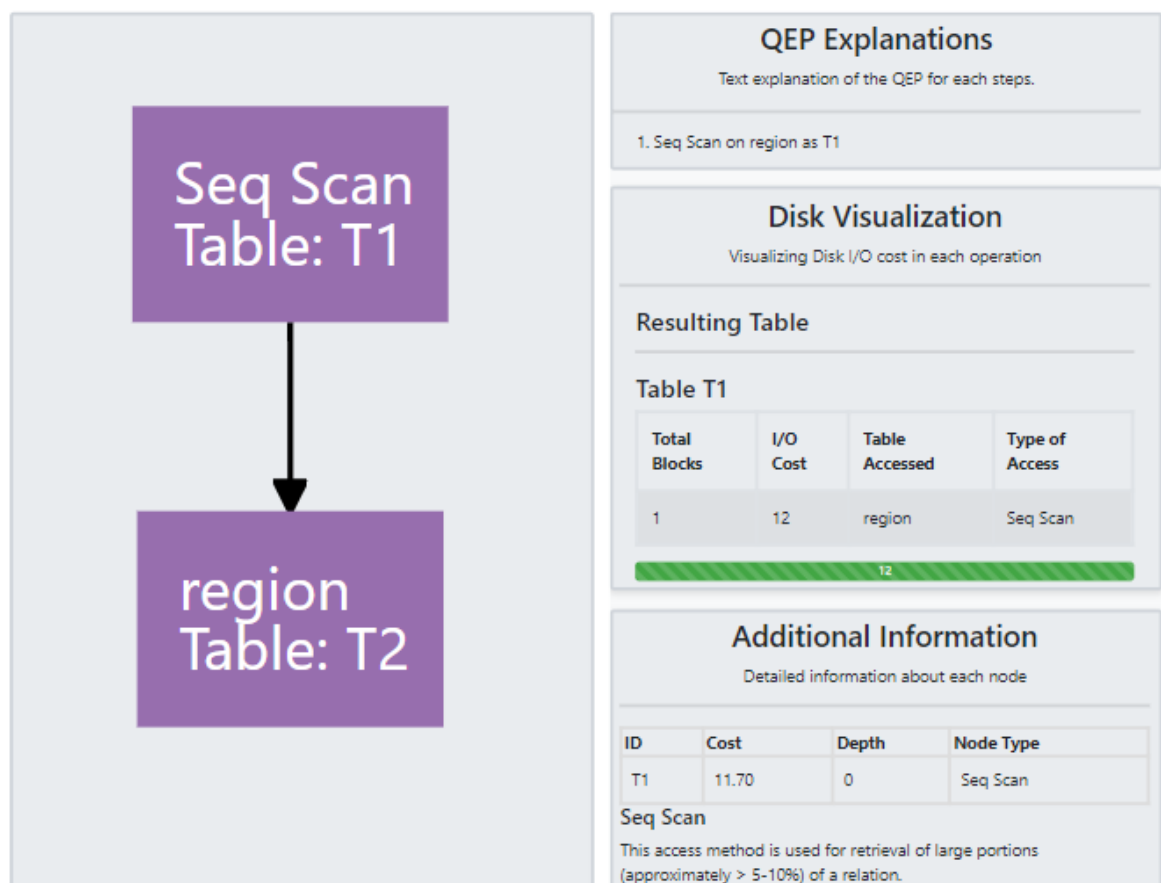


Figure 21: A snapshot of our easy-to-use and understand GUI.

3.3. Limitations Of Software

While our software strives to offer valuable insights into SQL query execution, it's important to recognize certain limitations inherent in its design and implementation. One notable challenge involves complexity handling, particularly when dealing with two or more levels of nested SQL queries. The other one is our Software's Dependency on the Query Execution Plan Accuracy. We noticed that the accuracy of our visualizations relies heavily on the correctness and completeness of the PostgreSQL Query Execution Plan. Thus, if the PostgreSQL's QEP breaks in some way or if it fails to provide the expected information, our visualizations may not accurately represent the actual execution details.

Thus, once we and our users have fully understood these limitations, it will allow them to better interpret and appreciate these results more effectively. They will then have more realistic expectations for our software's capabilities, as not as a software that has the solution for everything related to QEPs, but one that is adequate enough for regular users to shine more light onto how the QEP works and functions.

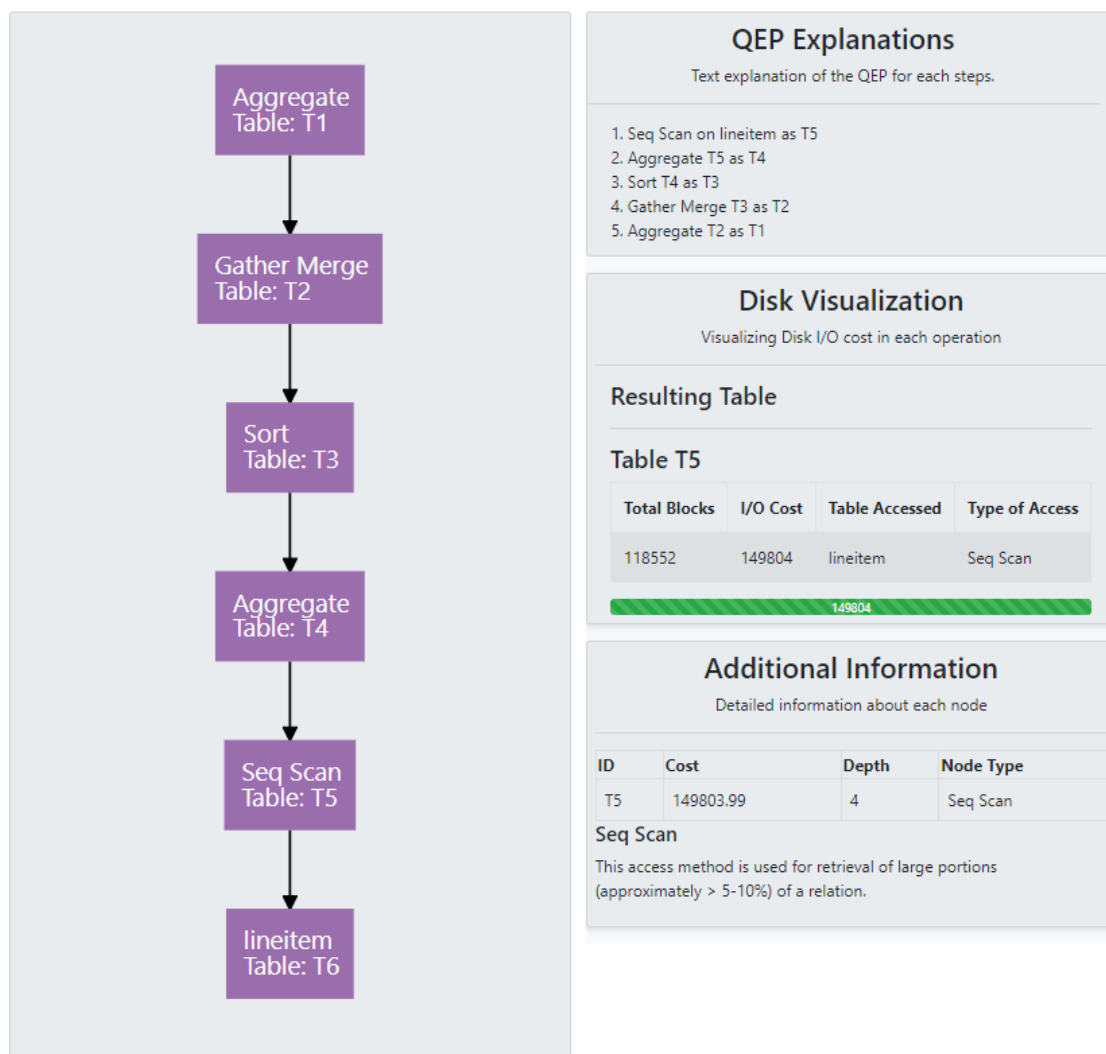
4. Sample Queries (with Results)

4.1.1. Sample Query 1

```
sample_query_1.sql U
api > sample_query_1.sql
1  SELECT
2      l_returnflag,
3      l_linestatus,
4      sum(l_quantity) AS sum_qty,
5      sum(l_extendedprice) AS sum_base_price,
6      sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
7      sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
8      avg(l_quantity) AS avg_qty,
9      avg(l_extendedprice) AS avg_price,
10     avg(l_discount) AS avg_disc,
11     count(*) AS count_order
12 FROM
13     lineitem
14 WHERE
15     l_extendedprice > 100
16 GROUP by
17     l_returnflag,
18     l_linestatus
19 ORDER BY
20     l_returnflag,
21     l_linestatus;
22
```

Graphs

[Click](#) on a node in the graph for more information.



4.1.2. Sample Query 2

```
sample_query_2.sql U
api > sample_query_2.sql
1 SELECT
2     l_orderkey,
3     sum(l_extendedprice * (1 - l_discount)) AS revenue,
4     o_orderdate,
5     o_shippriority
6 FROM
7     customer,
8     orders,
9     lineitem
10 WHERE
11     c_mktsegment = 'BUILDING'
12     AND c_custkey = o_custkey
13     AND l_orderkey = o_orderkey
14     AND o_totalprice > 10
15     AND l_extendedprice > 10
16 GROUP by
17     l_orderkey,
18     o_orderdate,
19     o_shippriority
20 ORDER by
21     revenue DESC,
22     o_orderdate;
```

Graphs

[Click](#) on a node in the graph for more information.

Limit
Table: T1

Aggregate
Table: T2

Nested Loop
Table: T3

Gather Merge
Table: T4

Sort
Table: T6

Seq Scan
Table: T8

orders
Table: T9

Index Scan
Table: T5

lineitem
Table: T7

QEP Explanations

Text explanation of the QEP for each step.

1. Seq Scan on orders as T8
2. Sort T8 as T6
3. Index Scan on lineitem as T5
4. Gather Merge T6 as T4
5. Nested Loop Join between Gather Merge T4 (outer) and Index Scan T5 (inner) as T3
6. Aggregate T3 as T2
7. Limit T2 as T1

Disk Visualization

Visualizing Disk I/O cost in each operation

Resulting Table			
Table T8			
Total Blocks	I/O Cost	Table Accessed	Type of Access
21472	29285	orders	Seq Scan

Additional Information

Detailed information about each node

ID	Cost	Depth	Node Type
T8	29284.50	5	Seq Scan

Seq Scan
This access method is used for retrieval of large portions (approximately > 5-10%) of a relation.

24

5. References

Snowflake (2023) TPC Benchmark™ H (TPC-H) specification, Sample Data: TPC-H | Snowflake Documentation. Available at: <https://docs.snowflake.com/en/user-guide/sample-data-tpch> (Accessed: 13 November 2023).

Transaction Processing Performance Council (TPC) (2014) TPC Benchmark H Standard Specification Revision 2.17.1 , TPC BENCHMARK H (Decision Support) Standard Specification. Available at: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf (Accessed: 13 November 2023).

PostgreSQL Global Development Group (2023) About | What is PostgreSQL?, PostgreSQL. Available at: <https://www.postgresql.org/about/> (Accessed: 15 November 2023).

