

**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**  
**SINGAPORE**

## **SC3020 Database System Principles**

### **Project 2**

#### **Group 1**

<b>Name</b>	<b>Email</b>  (excluding @e.ntu.edu.sg)	<b>Matriculation Number</b>
ANG YU JUAN	YANG045	U2121373G
CHAN WEN XU	CHAN1020	U2220390A
NG TZE KEAN	TNG042	U2121193J
TAN SAY HONG	STAN266	U2120837E
YU BOXUAN	BYU005	U2221435A

# Table of Contents

<b>1. Introduction</b>	<b>3</b>
1.1. Access and Installation Guide	3
1.2. Dataset	4
<b>2. Main</b>	<b>5</b>
<b>3. Interface</b>	<b>7</b>
3.1. Database Login	7
3.2. Visualisation	9
<b>4. Preprocessing</b>	<b>16</b>
<b>5. Pipe-syntax</b>	<b>17</b>
5.1. Obtaining Query Execution Plan	17
5.2. Query Execution Plan Format	17
5.3. Controller for QEP to Pipe-syntax	18
5.4. QEP to Graph Structure	19
5.5. Graph to Pipe-syntax	19
5.5.1. FROM	21
5.5.2. JOIN	21
5.5.3. WHERE	22
5.5.4. ORDER BY	22
5.5.5. AGGREGATE	22
5.5.6. LIMIT	23
5.6. Estimated Cost Annotation	23
<b>6. Limitations</b>	<b>24</b>
6.1. Difficulty in Handling Unsupported Queries	24
6.2. Hard-coded values from PostgreSQL	24
6.3. Long Processing Time for Large Queries	25
6.4. Limited Extensibility in Streamlit Interface	25
<b>7. Further Implementations</b>	<b>27</b>
7.1. Hosting Database on Amazon Cloud	27
7.2. Deploying on Streamlit Platform	28
7.3. Database Setup	30
7.3.1. PostgreSQL Setup	30
7.3.2. Database Initialisation	30
7.3.3. Data Ingestion	31
7.4. Use of GitHub for source control	31
<b>8. References</b>	<b>33</b>

# 1. Introduction

The goal of this project is to exploit the query execution plan (QEP) of a given SQL query to create an equivalent variant of the SQL using pipe-syntax to improve understanding of the query. The pipe-syntax provides a more linear and readable format that mirrors how the database engine sequentially processes a query.

We developed an interactive and user-friendly web application where users can input and execute SQL queries, view their QEPs in a graphical format and the corresponding pipe-syntax version. By analyzing the PostgreSQL source code, we designed a method to generate the pipe-syntax by parsing the QEP into a graph-based structure and processing it using object-oriented principles. Both the PostgreSQL database and the web application are hosted on the cloud for convenient access (as detailed in the section “*7. Further Implementations*”).

## 1.1. Access and Installation Guide

Unzip the downloaded source code `.zip` file. Before the application can be run, the required Python dependencies specified in the `pyproject.toml` file need to be installed. The dependencies can be installed using either pip or Poetry package managers.

### Option 1: Using pip

1. Open a command prompt or terminal window in the root of the source code directory.
2. Create a Python virtual environment (only needs to be done once):

```
python -m venv .venv
```

3. Activate the virtual environment for the directory:

```
# MacOS / Linux
source .venv/bin/activate
# Windows
.venv/Scripts/activate
```

4. Install the required dependencies:

```
python -m pip install .
```

### Option 2: Using Poetry

1. Open a command prompt or terminal window in the root of the source code directory.
2. Install the required dependencies:

```
poetry install
```

### Run the Project

In the root directory, run the application locally:

```
python project.py
```

## 1.2. Dataset

Our project uses the IMDB dataset [1], which has a schema with the tables *name\_basics*, *title\_akas*, *title\_basics*, *title\_crew*, *title\_episode*, *title\_principals*, *title\_ratings* (Figure 1).

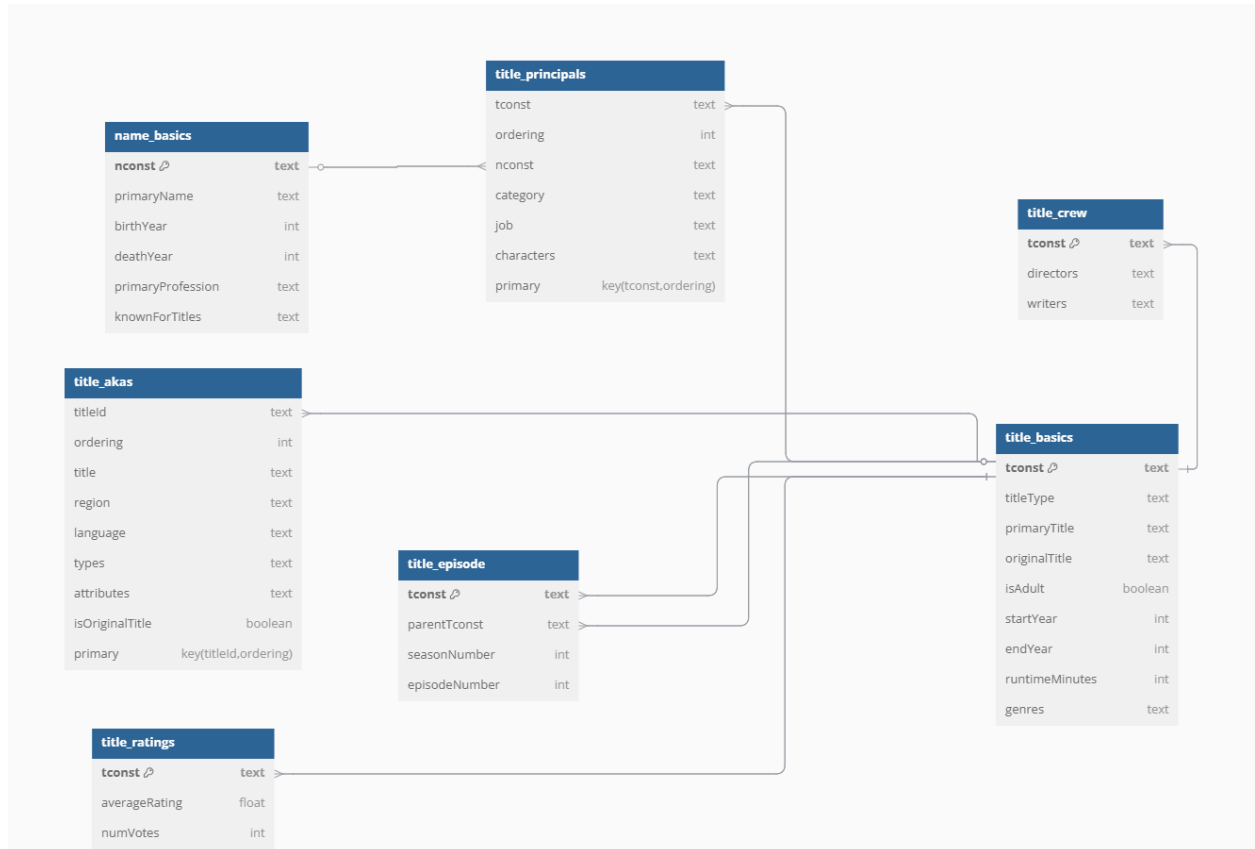


Figure 1: IMDB dataset schema

## 2. Main

The `project.py` file is the main entry point of the application that invokes all the necessary procedures from all the other program files. When the project is run, the web application graphical user interface (GUI) is launched by making a system call, which mimics the terminal command to execute the Streamlit server: `streamlit run interface.py`.

The system call creates a new shell process (instead of a subprocess) as we found that creating a new shell to run the interface is generally more stable. Note that users should have the necessary file system permissions to run the project file (i.e. execution rights).

## 3. Interface

The `interface.py` file contains the code for the GUI. The GUI is implemented as a web application using the Streamlit framework [2]. Streamlit was chosen because it allows rapid development of interactive, user-friendly interfaces for data-driven applications, with minimal overhead. Streamlit also facilitates quick and seamless cloud deployment, allowing users to access the application directly through a web browser without the need for any software installation.

### 3.1. Database Login

The login page serves as the entry point of the application for users. At the top of the page, a brief summary of the project and the key features of the application are presented to provide users with context (Figure 2). In the “*Database Login*” section, users are provided with the flexibility of two options for connecting to a database (Figure 2). By selecting the “*Cloud*” option, users can easily connect to a preconfigured database hosted on the cloud without the need for any initial setup or credentials. Alternatively, by selecting the “*Local*” option, users can connect to a local database instance by entering the necessary database credentials into the login form: database name, username, password, host, and port number (Figure 3).

**QEP Visualizer [SC3020 Group 1]**

**? Can Query Execution Plans Help You Generate Better SQL?**

This project aims to enhance SQL query comprehension by transforming input SQL queries into an equivalent, easier-to-read pipe-syntax format using query execution plans (QEP).

**🌟 Features:**

1. **Generate Pipe-Syntax Format:** Convert SQL queries into its pipe-syntax format, annotated with estimated execution costs.
2. **Visualize QEP:** Display an interactive visualization of the QEP for the SQL queries.
3. **Query Interface:** Provide a user-friendly interface to write and execute SQL queries with syntax highlighting, error validation, and display of query results in a dataframe. The database schema can also be viewed.

**Database Login**

Database Location

☒ Cloud

☐ Local

Login

*Figure 2: Login page with project summary*

Database Location

☐ Cloud

☒ Local

**Local Database Credentials**

Database Name

Enter database name (e.g. imdb)

Username

Enter username (e.g. group1)

Password

Enter password (e.g. group1)

Host

Enter host IP address (e.g. localhost)

Port

Enter port (e.g. 5432)

Login

*Figure 3: Login form to connect to a local database*

For a more intuitive user experience, the login form implements input validation and error handling. If the application is unable to connect to the database, an appropriate error message is displayed (Figure 4).

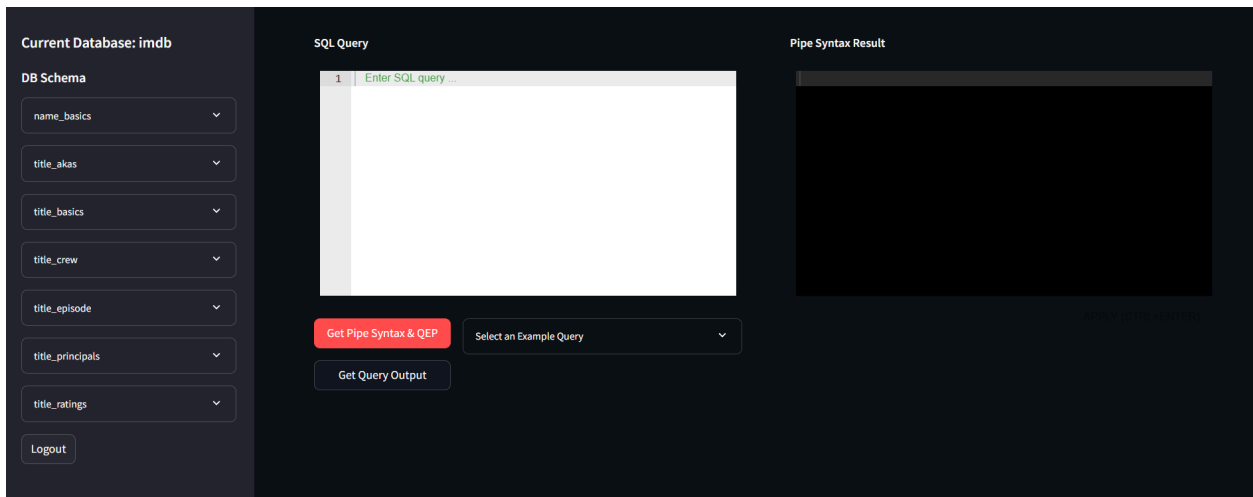




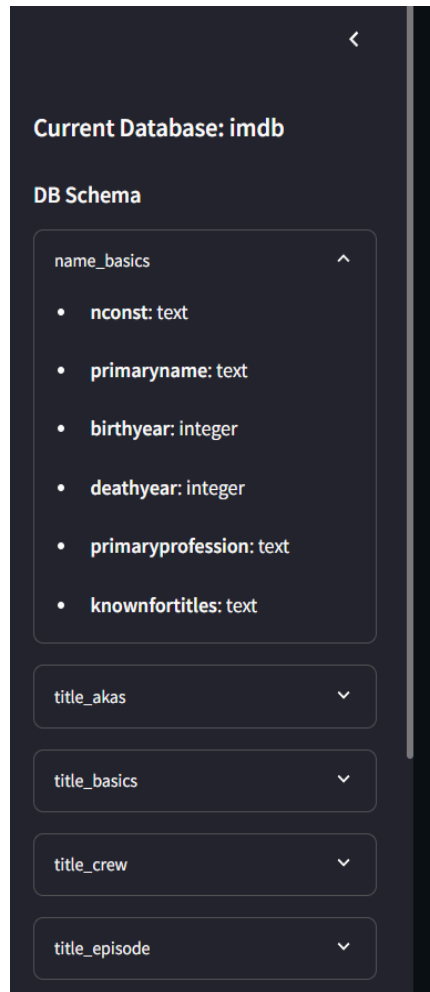
*Figure 4: Error message displayed on database connection failure*

## 3.2. Visualisation

Figure 5 shows the main visualisation page of the application where users can interact with the connected database, construct and execute SQL queries, visualise QEPS and corresponding pipe-syntax formats. On the left sidebar, users can refer to the database schema, which displays a list of all available tables. Each table can be expanded to view its attributes and data types (Figure 6).



*Figure 5: Main application page*



*Figure 6: Sidebar showing database schema, with an expanded table*

Users can input SQL queries into an editor that supports syntax highlighting (Figure 7) and autocomplete suggestions for SQL keywords (Figure 8). Additionally, users can choose from a predefined list of example queries to prepopulate the query editor (Figure 9). When the input query is executed or its QEP is retrieved, the input query will be validated beforehand. If the input SQL query is invalid, an appropriate error message will be displayed (Figure 10).

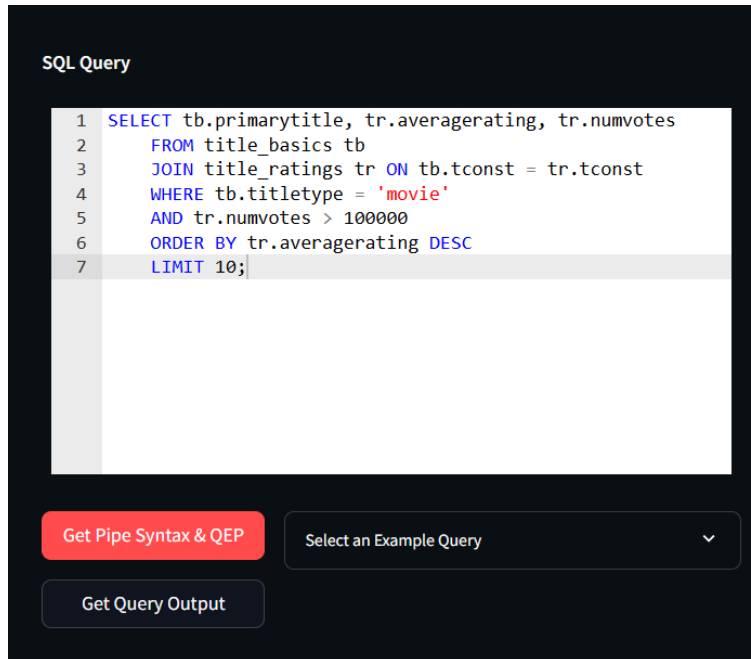


Figure 7: Syntax highlighting in SQL query input

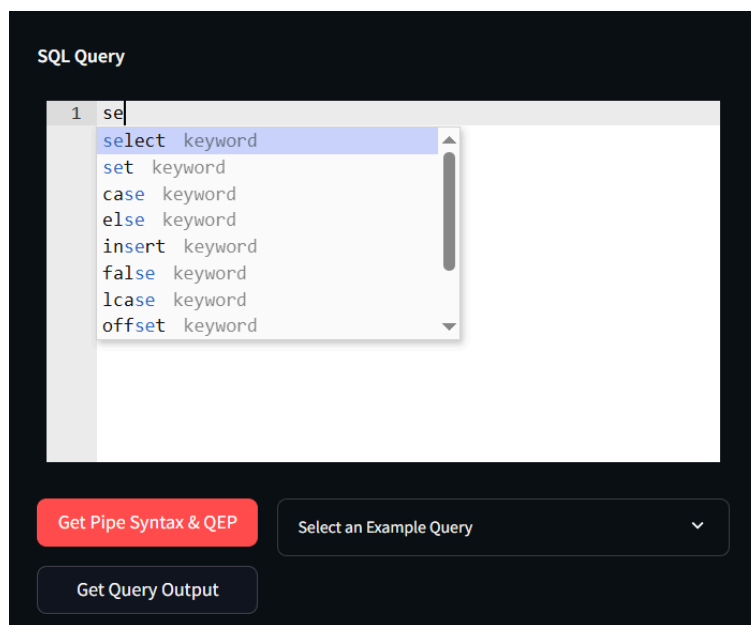


Figure 8: Autocomplete suggestions for SQL keywords in SQL query input

SQL Query

```
1 SELECT startyear, COUNT(*) AS num_titles
2 FROM title_basics
3 WHERE startyear BETWEEN 2020 AND 2022
4 GROUP BY startyear
5 ORDER BY startyear DESC;
```

Get Pipe Syntax & QEP

Get Query Output

Select an Example Query ^

- 1 - Get the top 10 highest-rated movies with more than 100,000 votes
- 2 - Get number of titles per year from 2020-2022
- 3 - Get all actors in a specified movie

Figure 9: Example queries that can be selected for SQL query input

SQL Query

```
1 se
```

Get Pipe Syntax & QEP

Select an Example Query v

Get Query Output

Failed to get QEP. Error:  
syntax error at or near "se"

SQL Query

```
1 select * from x;
```

Get Pipe Syntax & QEP

Select an Example Query v

Get Query Output

Failed to get QEP. Error:  
relation "x" does not exist

Figure 10: Error message displayed on invalid SQL query input submitted

When the user clicks the “*Get Pipe Syntax & QEP*” button with a valid SQL query, the application performs two operations. First, it converts the SQL query into a pipe-syntax format, which is displayed in the “*Pipe Syntax Result*” section (Figure 11). Each operation of the QEP along with its associated cost is represented in the sequential pipe-syntax format.

### SQL Query

```

1 SELECT tb.primarytitle, tr.averagerating, tr.numvotes
2 FROM title_basics tb
3 JOIN title_ratings tr ON tb.tconst = tr.tconst
4 WHERE tb.titletype = 'movie'
5 AND tr.numvotes > 100000
6 ORDER BY tr.averagerating DESC
7 LIMIT 10;

```

### Pipe Syntax Result

```

FROM title_ratings AS tr <|COST 30761.1|>
|> WHERE tr.numvotes > 100000 <|COST 30761.1|>
|> JOIN (
  FROM title_basics AS tb <|COST 8.25|>
  |> WHERE tb.titletype = 'movie';:text <|COST 8.25|>
) ON tb.tconst = tr.tconst <|COST 61152.41|>
|> AGGREGATE tb.primarytitle, tr.averagerating, tr.numvotes
|> ORDER BY tr.averagerating DESC <|COST 61157.93|>
|> LIMIT 10 <|COST 61157.38|>

```

Get Pipe Syntax & QEP

Select an Example Query

Get Query Output

Figure 11: Display of pipe-syntax result

Second, the QEP for the query is generated and visualized with an interactive flow diagram in the “*QEP Visualizer*” section (Figure 12). Users can zoom, pan, and drag nodes within the visualization, allowing them to dynamically explore the QEP. Each node in the visualization displays key execution information, including the node type, additional details (such as sort key and join type), total cost, and estimated execution time. Users can also expand each node to view detailed explanations (Figure 13) to effectively interpret the QEP.

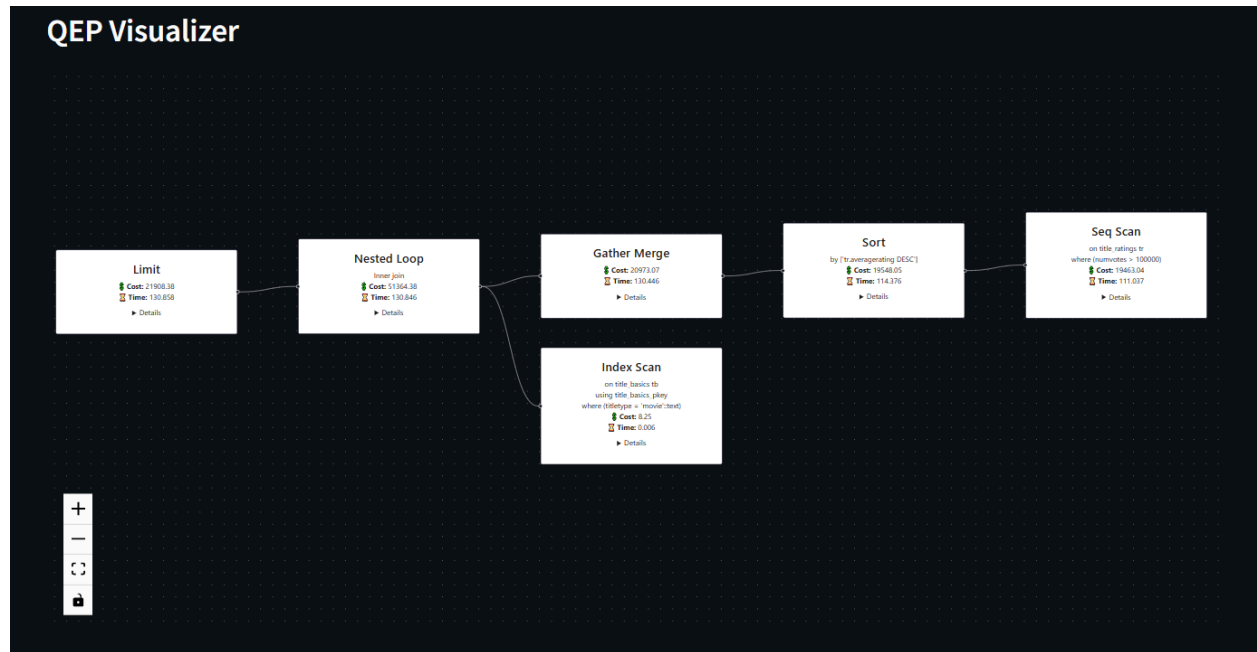


Figure 12: Flow diagram visualization of the input query's QEP

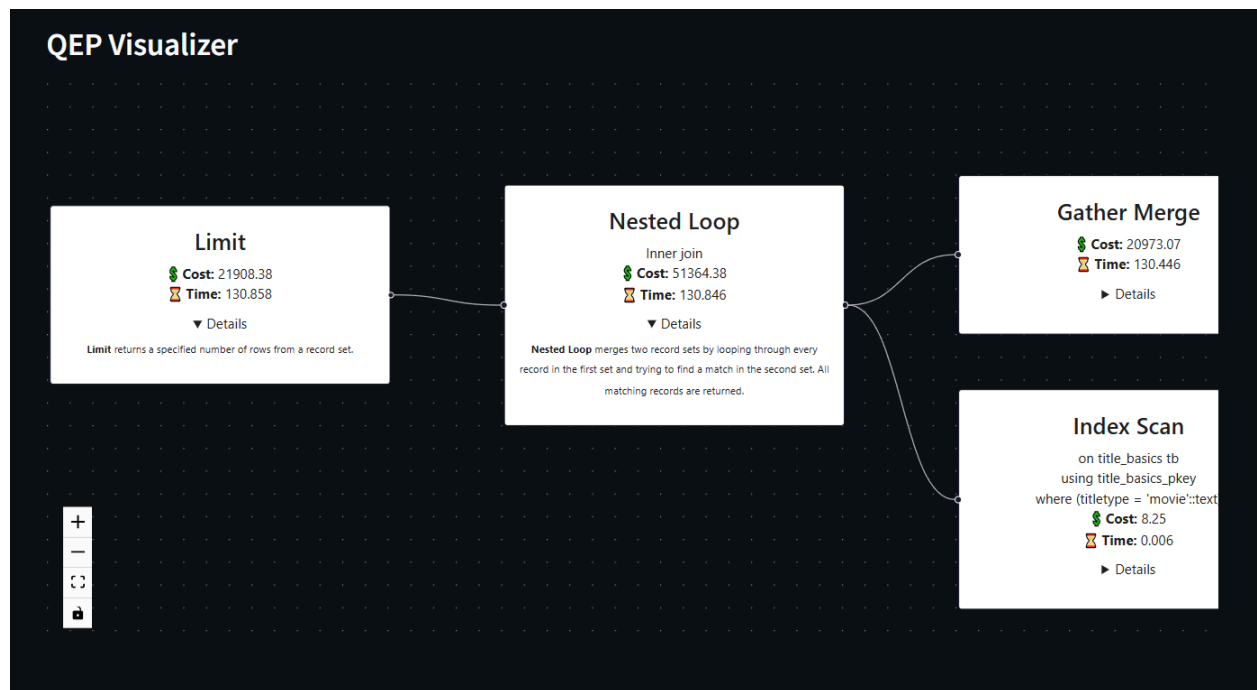
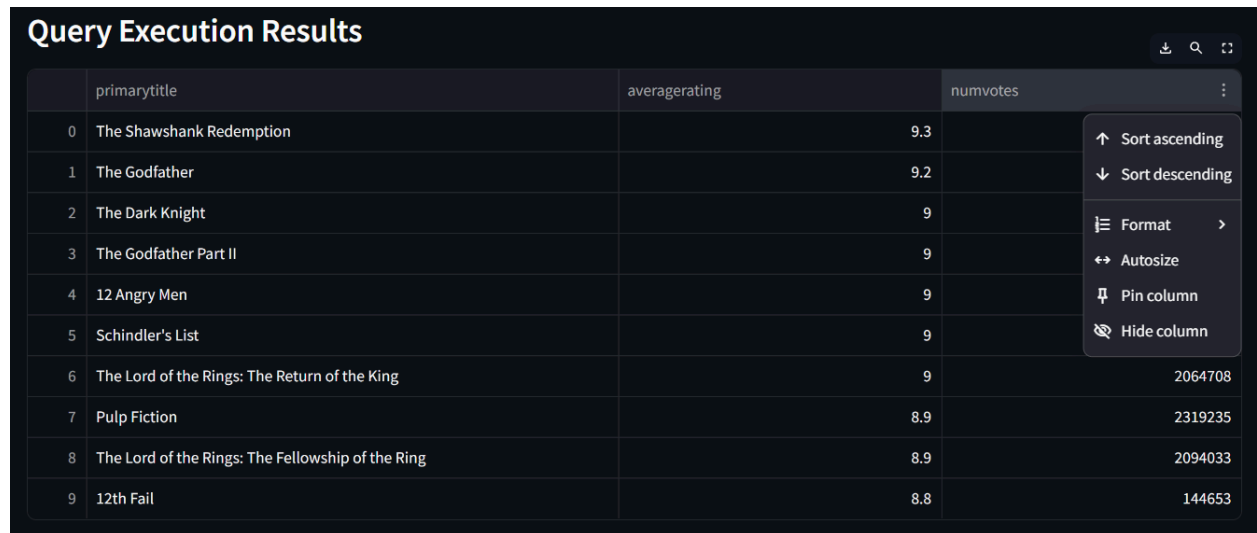


Figure 13: Expanded explanation for respective node types

Users can execute the input SQL query by clicking the “*Run Query*” button. The application then executes the query in the database and displays the results as an interactive dataframe in the “*Query Execution Results*” section (Figure 14).



The screenshot shows a dark-themed interface titled "Query Execution Results". It contains a table with 10 rows of movie data. The columns are "primarytitle", "averagerating", and "numvotes". An interactive menu is open on the right side of the table, showing options: "Sort ascending", "Sort descending", "Format", "Autosize", "Pin column", and "Hide column".

	primarytitle	averagerating	numvotes
0	The Shawshank Redemption	9.3	
1	The Godfather	9.2	
2	The Dark Knight	9	
3	The Godfather Part II	9	
4	12 Angry Men	9	
5	Schindler's List	9	
6	The Lord of the Rings: The Return of the King	9	2064708
7	Pulp Fiction	8.9	2319235
8	The Lord of the Rings: The Fellowship of the Ring	8.9	2094033
9	12th Fail	8.8	144653

Figure 14: Query execution results displayed as an interactive dataframe

## 4. Preprocessing

The `Database` class in the `preprocessing.py` file is designed to interact with a PostgreSQL database. It provides methods to other classes to use PostgreSQL without needing to create another connection. The `Database` class mainly handles the following:

- Establish a connection to the database.
- Retrieve the database schema (tables and their columns).
- Retrieve the QEP for a given SQL query.
- Execute SQL queries and return the results as a Pandas `DataFrame`.
- Close the database connection.

When initialising the connection to the database, we disable parallel cost estimation to ensure accurate cost estimations in the QEP. This is achieved by disabling parallel processes per execution node with the following command:

```
SET max_parallel_workers_per_gather = 0;
```

The key aspect of this file other than creating the connection and to safely close the connection to the database, would be to retrieve the QEP which is done as shown below. By appending the `EXPLAIN` clause, PostgreSQL returns the QEP instead of the query results. The output is then converted into a Python JSON object which will be passed to the other classes for usage.

```
self.cursor.execute(
    sql.SQL("EXPLAIN (ANALYZE, FORMAT JSON, VERBOSE TRUE)
    {}").format(sql.SQL(query))
)
qep_result = self.cursor.fetchone()[0]
return json.dumps(qep_result)
```



Additionally, when retrieving the QEP, the DBMS will check whether the query contains syntax errors, refers to non-existent tables or columns, or has invalid logic and returns the specific error to be displayed on the GUI.

## 5. Pipe-syntax

### 5.1. Obtaining Query Execution Plan

To parse the QEP given by the DBMS we take an object-oriented approach by creating various classes to process the QEP output. There are various ways that one can use PostgreSQL to obtain the QEP used by the DBMS, one such way is to invoke the **EXPLAIN** keyword [3] with the corresponding query. This includes the action that will be performed, the tables that will be referred and the physical plans that will be run to obtain the results.

The PostgreSQL **ANALYZE** clause [4] runs the actual statement and appends the actual runtime statistics to the QEP output. We will leverage this knowledge to obtain the QEP that we will need to parse.

### 5.2. Query Execution Plan Format

There are various modes that PostgreSQL can output the QEP in. Passing the text format will cause PostgreSQL to output the QEP as a human-readable format.

```
QUERY PLAN
-----
Seq Scan on Movie (cost=0.00..155.00 rows=10000 width=4)
```

(1 row)

While this is something that is easy to read, we want a more detailed output where we are able to fetch the relevant details of each node and to process it so that we are able to display the output as a graph and to also parse it as a tree to obtain the pipe-syntax. Instead, we output the QEP in JSON format which we can easily use with the Python `json` package to process the output.

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;  
QUERY PLAN
```

```
-----  
[                                         +  
  {                                         +  
    "Plan": {                             +  
      "Node Type": "Seq Scan",+  
      "Relation Name": "foo", +  
      "Alias": "foo",           +  
      "Startup Cost": 0.00,     +  
      "Total Cost": 155.00,    +  
      "Plan Rows": 10000,      +  
      "Plan Width": 4          +  
    }                               +  
  }                               +  
]                                   +  
(1 row)
```

### 5.3. Controller for QEP to Pipe-syntax

To handle the parsing of the QEP we created various classes which we will detail in the next few subsections. The overall logic controller is managed by `PipeSyntaxParser`. The class obtains the QEP and processes the QEP into pipe-syntax. We expose 2 public functions `get_pipe_syntax` and `get_execution_plan_graph` which returns processed objects to the interface for display purposes.

This design focuses on creating a consistent API for `interface.py` to call upon to allow us to work on this project in parallel without conflicts in the expected output of upstream and downstream objects.

## 5.4. QEP to Graph Structure

The first step in processing the QEP was to parse the JSON file and to convert it into a graph structure so that we are able to locate the leaves of the tree. The advantage is that when we see a node with 2 children, we know that a join is being performed and when a node only has a single child, it is likely performing a scan, aggregate or sort operation.

With these advantages in mind, we construct a class `QueryPlanNode` that represents a node object. We also create a class `QueryExecutionPlanGraph` that handles the conversion of the QEP to a graph structure. The class would iterate through the JSON level-by-level and create a node at each juncture, and append any children to the current node. Note that this is a top-down processing of the JSON structure. With that, we now have a graph that we can work on to generate the pipe-syntax.

## 5.5. Graph to Pipe-syntax

We analyse how the QEP is created by first analysing the PostgreSQL source code [5].

```
switch (((Join *) plan)->jointype)
{
    case JOIN_INNER:
        jointype = "Inner";
```

```

        break;
    case JOIN_LEFT:
        jointype = "Left";
        break;
    case JOIN_FULL:
        jointype = "Full";
        break;
    case JOIN_RIGHT:
        jointype = "Right";
        break;
    case JOIN_SEMI:
        jointype = "Semi";
        break;
    case JOIN_ANTI:
        jointype = "Anti";
        break;
    case JOIN_RIGHT_SEMI:
        jointype = "Right Semi";
        break;
    case JOIN_RIGHT_ANTI:
        jointype = "Right Anti";
        break;
    default:
        jointype = "???";
        break;
}

```

This is an example of the parsing done by PostgreSQL and the possible output of the join type from the QEP. Through such examination of the source code, we are able to identify the possible types of join such that we can cater for all possible join types when converting the QEP into the pipe-syntax format. To give an example, if we encounter a `jointype == "Left"` then we will parse the QEP node and handle it as a left join operation, of course we would also make use of other meta information like the node type if it was a “*Nested Loop*”, “*Merge Join*” or “*Hash Join*”.

The key idea to our approach is to examine the possible cases of the QEP output and then to ensure that our parsing will take into account the cases and to have a default handler if we encounter a case that we have not taken into consideration [6]. The node is then converted to pipe syntax based on the node information in the `convert_node_to_pipe_syntax` function.

### 5.5.1. FROM

- If the node represents a `FROM` clause, it generates an instruction with the table name and alias (if applicable).
- Filters and child nodes are processed to add additional instructions.

```
from_ins = ["FROM", n.from_info.relation_name]
if n.from_info.alias != n.from_info.relation_name:
    from_ins.extend(["AS", n.from_info.alias])
instruction = [from_ins]
```

### 5.5.2. JOIN

- If the node represents a join (e.g. “*Nested Loop*”, “*Merge Join*”, “*Hash Join*”), it processes the left and right child nodes recursively.
- Filters and conditions (e.g. `ON` clauses) are extracted and added to the join instruction.

```
join_instruction = [
    "JOIN" if n.join_type == "Inner" else f"{n.join_type} JOIN",
    right_text,
    f"ON {self._unwrap_expr(join_filters)}"
]
```

### 5.5.3. WHERE

These are straightforward to handle since we are able to check the hash or index condition in the QEP node generated. The keywords used by PostgreSQL are “*Hash Cond*” and “*Index Cond*” which we can extract through a dictionary.

```
if filters:
    instruction.append(["WHERE", self._unwrap_expr(filters)])
```

### 5.5.4. ORDER BY

Similarly to the where clause, the order by condition can also be found in the QEP by fetching the “*Order By*” keyword.

### 5.5.5. AGGREGATE

The aggregate clause requires us to fetch the output of the QEP, which we can obtain with a small trick. That is to add `VERBOSE TRUE` to the `EXPLAIN` command of PostgreSQL. This will cause the QEP to output the columns to project as well. The grouping is trivial since we can obtain it as a keyword “*Group Key*”.

```
aggregate_instruction = ["AGGREGATE", self._clean_output(n.output)]
if n.group_keys:
    aggregate_instruction.append("GROUP BY")
    aggregate_instruction.append(", ".join(n.group_keys))
instruction.append(aggregate_instruction)
```

### 5.5.6. LIMIT

When there is a limit set on the number of output rows, obtain the estimated rows from the QEP output. If it does not exist then the pipe syntax will default to “???” since this information is not known to the QEP as well.

```
if n.node_type == "Limit":
    limit_row_guess = "???"
    if n.cost_info and n.cost_info.plan_rows:
        limit_row_guess = str(n.cost_info.plan_rows)
    instruction.append(["LIMIT", limit_row_guess])
```

## 5.6. Estimated Cost Annotation

In the pipe-syntax, we annotate each component of the output query with the estimated cost retrieved using the QEP. To obtain the estimated cost, we can fetch from the keyword *"Total Cost"* to obtain the corresponding cost for each node and perform a mapping to the generated pipe-syntax.

We note that there are times where the estimated cost would repeat itself in the QEP. As we do not perform additional processing on this cost figure, we will report the cost per line as per the QEP results. The formula for the cost estimation in PostgreSQL is defined as follows:

$$\textit{Total Cost} = \textit{Startup Cost} + \textit{Run Cost}.$$

- The start-up cost is the cost expended before exactly fetching the first tuple in the table.
- The run cost is the cost to fetch all tuples.

For accuracy, we report the total cost as an aggregate of the 2 possible costs reported by PostgreSQL.

## 6. Limitations

### 6.1. Difficulty in Handling Unsupported Queries

The `QueryPlanNode` class explicitly excludes certain node types (e.g. “*Sample Scan*”, “*Function Scan*”, “*Tid Scan*”, “*Foreign Scan*”, etc.) by asserting that they are not supported.

```
if node_type in ["Sample Scan", "Function Scan", "Tid Scan",
                 "Tid Range Scan", "Foreign Scan", "Custom Scan",
                 "WindowAgg", "Modify Table"]:
    logger.error("Unexpected node type: %s", node_type)
    assert False
```

This restricts the ability to handle more complex or less common QEP node types, which may appear in real-world queries. In the case where a user does input such an uncommon QEP then the program will fail to respond since we will not handle such cases as we are using the assumption that the user is purely trying to query the database rather than to perform operations such as modifications to the database.

### 6.2. Hard-coded values from PostgreSQL

To obtain the values of the QEP node, we use hard-coded values that we found in the PostgreSQL source code (e.g. “*Hash Cond*”). These values could potentially change in source, and if that were to happen, we would run into unexpected errors such as `hash_cond` being `None` which can cause errors in our application.

```
hash_cond = node_data.get("Hash Cond")
index_cond = node_data.get("Index Cond")
```



```
filter_cond = node_data.get("Filter")
order_by = node_data.get("Order By")
sort_key = node_data.get("Sort Key")
```

### 6.3. Long Processing Time for Large Queries

We note that query processing will be slow as we disable the parallel worker threads for accurate cost estimation. As we also want to obtain the true cost of the query, we must wait for the DBMS to execute the query before we are able to obtain the QEP output with the corresponding costs.

```
SELECT * FROM name_basics
```

When executing a query that returns a large result set, such as selecting all records from the `name_basics` table, which contains approximately 50 million rows, the DBMS can take a significant amount of time to process the request. In such cases, the application is forced to wait for the DBMS to respond, which may cause the app to appear unresponsive to the user. For the given query, it takes approximately 1 minute to process due to the large table size.

Our suggestion would be to automatically input a `LIMIT` clause to limit the amount of processing work the DBMS has to go through for a faster output. For example:

```
SELECT * FROM name_basics LIMIT 10;
```

### 6.4. Limited Extensibility in Streamlit Interface

We recognise that Streamlit has its limitations; however, we made the trade-off to use Streamlit for this project due to its ease of use and ability to bootstrap a working GUI quickly. One of Streamlit's primary drawbacks is its limited component library and inflexible layout system,

which makes it difficult to implement advanced or highly interactive features without resorting to custom JavaScript or frontend frameworks like React. Additionally, Streamlit may not scale well for larger or more complex applications. The Streamlit engine re-runs the entire script from top to bottom on every user interaction, which can be inefficient for compute-heavy tasks or complex state management.

## 7. Further Implementations

### 7.1. Hosting Database on Amazon Cloud

During the data ingestion process, we observed that initialising the database takes approximately 30 minutes and loads around 21 GB of data. To avoid re-initialising the database on each member's machine and to ensure consistency across the team, we opted to host the database in the cloud. This approach allows all team members to connect to a centralised instance with standardised configurations.

While hosting the database, our main consideration is cost. Given the constraint of operating within the AWS Free Tier, we explored two hosting options:

Feature	Amazon RDS	Amazon EC2
Free Tier Hours	750 hours/month	750 hours/month
Instance Types	db.t3.micro / db.t4g.micro	t2.micro / t3.micro
Database Engine	Fully managed by AWS	Self-managed PostgreSQL
Operating System	Fully managed by AWS	Linux
Storage	20 GB General Purpose SSD	30 GB General Purpose SSD
Public IPv4 Access	Not included	750 hours/month included

While Amazon RDS offers a simpler, fully managed setup, it comes with two key limitations for our use case. Firstly, the 20 GB storage limit under the free tier is insufficient for our dataset. Secondly, configuring a public IP to access via the Internet will incur additional costs. To stay within the free tier and meet our storage requirements, we used AWS EC2 to manually host the PostgreSQL server.

The EC2 instance was provisioned as a *t2.micro* instance with 1 vCPU, 1 GB RAM, and 28 GB Amazon EBS storage. Figure 15 shows the running EC2 instance configured for our database deployment.

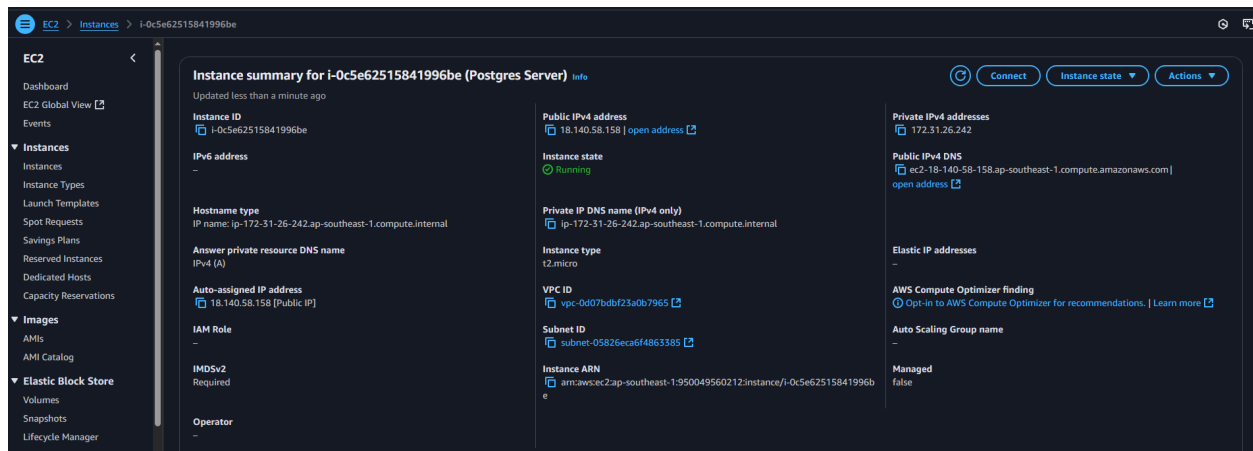


Figure 15: Screenshot of Running EC2 Instance Hosting PostgreSQL

## 7.2. Deploying on Streamlit Platform

Streamlit Deployment URL: <https://sc3020-group1-ay24s2.streamlit.app/>

We deployed our web application using Streamlit Community Cloud. The deployment process required a connection to our codebase on GitHub and Python library dependencies, which we

specified in a `uv.lock` file. Once deployed, the hosted application is shown (Figure 16) and users will be able to publicly access the URL to access the application.

#### hiiamtzekean's apps



*Figure 16: Deployed application on Streamlit Community Cloud*

Note that connecting to a local database on the Streamlit deployment requires port forwarding. The user's host address and port (Figure 17) must be exposed publicly so that the public web application can access the user's local database. Alternatively, the user can connect to the deployed cloud database.

A screenshot of a "Local Database Credentials" form. The form has a dark background with light-colored text. It contains five input fields: "Database Name" (placeholder: "Enter database name (e.g. imdb)"), "Username" (placeholder: "Enter username (e.g. group1)"), "Password" (placeholder: "Enter password (e.g. group1)"), "Host" (placeholder: "Enter host IP address (e.g. localhost)"), and "Port" (placeholder: "Enter port (e.g. 5432)"). The "Host" and "Port" fields are grouped together and highlighted with a red rectangular border. Below these fields is a "Login" button.

*Figure 17: Local database credentials that need to be exposed publicly*

## 7.3. Database Setup

### 7.3.1. PostgreSQL Setup

The PostgreSQL database can be either set up using Docker or manually:

#### Option 1: Using Docker

Run the following command to build and start the PostgreSQL container automatically:

```
docker-compose up -d --build
```

#### Option 2: Manual Setup

1. Install PostgreSQL (minimally version 13) on the local system.
2. Create a database named `imdb`.
3. Create a user named `group1` with the password `group1`.

### 7.3.2. Database Initialisation

To initialise the database, we provide an additional `ingest_data.py` file. This file contains a script which handles the connection to the PostgreSQL database and performs the ingestion of data into the appropriate tables. The steps to initialise the database are as follows:

1. Download all listed `.tsv` files in the IMDb Dataset from the following URL:  
<https://datasets.imdbws.com/>
2. Extract the files and place them in the `/init/data` directory
3. Run the data ingestion script:

```
python ingest_data.py
```

### 7.3.3. Data Ingestion

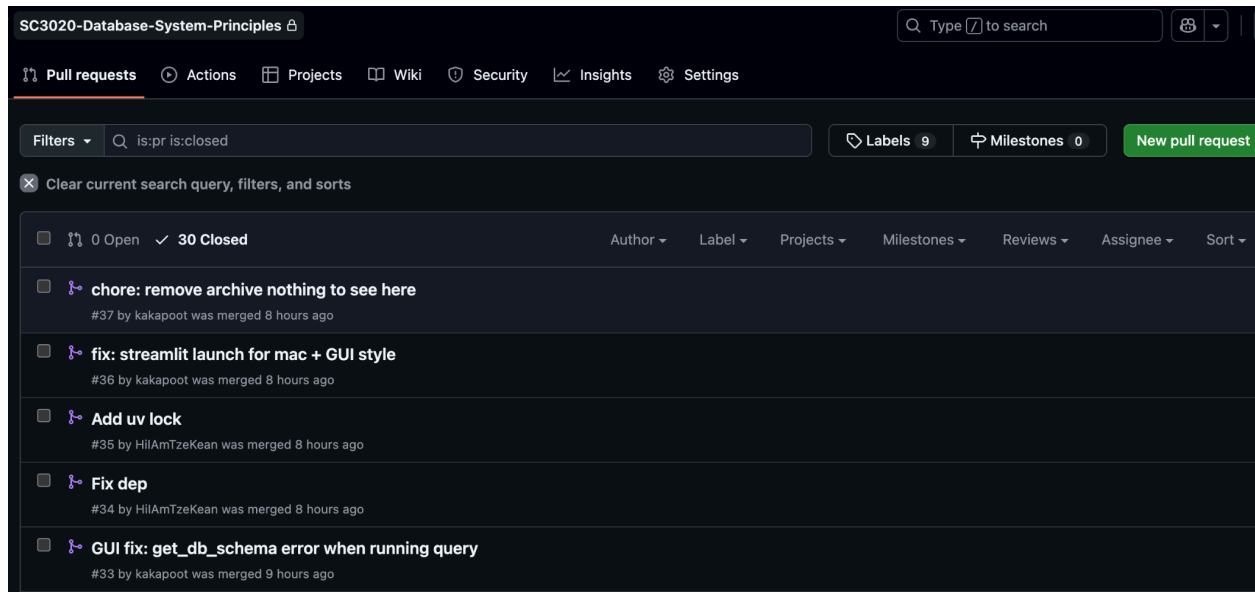
We used the `psycopg2` package to create a connection to the database. Using this connection, we created the database tables based on the schema. Once the schema is set up, our code then iterates through each `.tsv` file, line by line, to check if the data contents can be safely decoded from Latin-1 to UTF-8 encoding. The lines that could not be decoded were deemed to be corrupted data and were excluded to prevent encoding-related errors during ingestion. Finally, the cleaned lines were loaded into memory using a `StringIO` buffer and then ingested into the respective database tables using the `copy` command.

## 7.4. Use of GitHub for source control

GitHub URL:

[https://github.com/HiIAmTzeKean/SC3020-Database-System-Principles/tree/master/proj\\_2](https://github.com/HiIAmTzeKean/SC3020-Database-System-Principles/tree/master/proj_2)

In this project, we use GitHub as a source control tool for concurrent project development. To coordinate our work and to track our changes, we used GitHub Issues to track features and bugs that we needed to implement (Figure 18), allowing us to comment on the issue and to track its progress. We also utilised pull requests as a form of proposal to the changes to the code base so that team members can review before we merge the changes in our codebase.



*Figure 18: GitHub Issues for our project repository*



## 8. References

- [1] “IMDb Data Files Download.” Accessed: Apr. 19, 2025. [Online]. Available: <https://datasets.imdbws.com/>
- [2] “A faster way to build and share data apps,” Streamlit. Accessed: Apr. 14, 2025. [Online]. Available: <https://streamlit.io/>
- [3] “EXPLAIN,” PostgreSQL Documentation. Accessed: Apr. 19, 2025. [Online]. Available: <https://www.postgresql.org/docs/17/sql-explain.html>
- [4] “PostgreSQL: Documentation: 17: ANALYZE.” Accessed: Apr. 19, 2025. [Online]. Available: <https://www.postgresql.org/docs/current/sql-analyze.html>
- [5] “postgres/src/backend/commands/explain.c at master · postgres/postgres,” GitHub. Accessed: Apr. 19, 2025. [Online]. Available: <https://github.com/postgres/postgres/blob/master/src/backend/commands/explain.c>
- [6] J. Shute *et al.*, “SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL,” *Proc. VLDB Endow.*, vol. 17, no. 12, pp. 4051–4063, Aug. 2024, doi: 10.14778/3685800.3685826.