



**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

CZ4031 Database System Principles

Project 2 - Understanding Query Plans During Data Exploration

Group 17

CHOI SEUNGHWAN	U2021319B
MUHAMMAD RIAZ BIN JAMALULLAH	U2022016H
TIMOTHY CHANG ZU'EN	U2022114A
YEOW MING XUAN	U2022461J

All members contributed equally to this project

		js	# Folder with templates for styling
		templates	# Interface scripts
			index.html # Contains markup and content for interface
			outline.html # Template html that index inherits from
			Some other html scripts, unimportant
		project.py	
		qepGenerator.py	# contains classes QEPNode, QEPTree
		preprocessing.py	# Retrieves QEP and metrics from Postgres
		annotation.py	# Generates initial QEP natural lang. explanation
		comparisonGenerator.py	# Compares QEPs, generates natural lang. explanation
		interface.py	# Hides the complexity of QEPNode, Tree and Comparer
		query-parser.py	# For parsing SQL queries received from the frontend
		Dockerfile	# Contains docker configuration
		requirements.txt	# For external users to get dependencies
		database	# Contains raw data and schema files for loading
			CSV Files
			database-dump.sql # Schema file to load csv tables into Postgres
			Dockerfile # Contains docker configuration
		docker-compose.yaml	# For building container

We employed the Procedural Oriented Programming model to structure this assignment into smaller components; each file has its purpose and utility functions. This makes the codes more understandable, as each file now serves to carry out a single purpose. The functional files are then called from and integrated with the views to serve requests from the clients.

The implementation of our application consists of the following steps:

- Step 1: Generate query execution plans based on input queries
- Step 2: Generate the natural language description of the QEPs
- Step 3: Separate the queries into its different clauses and predicates
- Step 4: Generate the natural language explanation for the difference in queries and QEPs
- Step 5: Present the results in the Graphical User Interface (GUI)

2. Installation guide

2.1 Database

Due to the large size of the TPC-H dataset, and the fact that you will be executing the software using different datasets, we assume that an instance of the PostgreSQL database loaded with the TPC-H dataset and other datasets will be present on the side of the user grading this project. Thus, we provide the files needed to set up the database instance, *Dockerfile*, *docker-compose.yml*, and *database-dump.sql*.

For internal collaboration, our team uses Docker to run identical instances of the TPC-H database. In the creation of the Postgres docker container, the *database-dump.sql* script will be run, automatically importing the TPC-H dataset.

To create this database dump, a new instance of PostgreSQL was loaded, and the dataset was loaded into the database by manually importing the 8 different CSV files. The entire database was then dumped into a single file, *database-dump.sql*. Doing this meant that the team could avoid performing extraneous steps such as manually running scripts to create the .tbl and .csv files.

2.2 Application

This section contains **instructions on running the application**.

The following instructions apply to Windows systems. For Mac systems, some commands may differ.

1. Download Dockerfile, docker-compose.yml and database-dump.sql, and the 8 TPC-H CSV files from [OneDrive](#).
2. Ensure all three files are located in the same container.

For testing on databases other than TPC-H: Ensure the dataset's CSV files are in the same directory, so that the database can be built.

Steps:

1. From the main directory, navigate to the app/ directory using `cd app`.
2. Create a Python virtual environment.

```
python -m venv env
```

Install Flask in the virtual environment by running the following command

```
python -m pip install flask
```

3. Activate virtual environment and install requirements.

```
env\Scripts\activate
```

```
pip install -r requirements.txt
```

4. Configure the database connection parameters to point to your local Postgres instance in `app/preprocessing.py` (Lines 32-42)

```
32     @classmethod
33     def get_conn(cls):
34         if cls._conn is None:
35             cls._conn = psycopg2.connect(
36                 host="postgres",
37                 database="TPC-H",
38                 user="postgres",
39                 password="password123",
40                 port="5432",
41             )
42         return cls._conn
```

Fig. 1 Example of our parameters

5. Run the project.

```
flask --app project run
```

6. The web app will be accessible at <http://localhost:5001/>

Alternatively, you may use the Docker version for easier setup.

Pre-requisites: Docker Engine and Docker Compose installed.

1. From the main directory, navigate to the `app/` directory using `cd app`.
2. Build and start up the container using `docker compose up --build`. During first creation, the database dump will be imported to the database automatically.
3. The web app will be accessible at <http://localhost:5001/>

3. Key Algorithms

3.1 Preprocessing

3.1.1 Database connection

Various functions in our program require a connection to the database. To avoid using a global variable or multiple instantiations of a database connection, the singleton pattern is used. By making sure that only one instance of the DatabaseConnection class exists at any one time, this pattern ensures that all calls to `get_conn()` will return the same database connection object, thus avoiding the overhead of creating a new connection each time one is needed. If none exists at the time, one is created.

The code will be tested on different datasets for the purpose of grading, so the user grading this project can also edit the value of `database` as convenient.

```
class DatabaseConnection:
    """
    Database connection singleton class.
    Call get_conn() to get a database connection.
    """

    _conn = None

    @classmethod
    def get_conn(cls):
        if cls._conn is None:
            cls._conn = psycopg2.connect(
                host="postgres",
                database="TPC-H",
                user="postgres",
                password="password123",
                port="5432",
            )
        return cls._conn
```

Fig. 2 Database Connection

3.1.2 Query Parsing

Query parsing consists of the following general stages:

1. Getting user input (the SQL query) from the front-end application
2. Creating a regular expression for each possible SQL clause
3. Searching the query for matches
4. Splitting the clauses and their predicates into key-value pairs

We define a function called `parse_sql_query` that takes an SQL query string as input and extracts the different predicates of the different clauses into a dictionary. We decided to use the regular expression package to search for patterns in the SQL query string and extract the relevant predicates. The limitation of this method is that nested SQL statements in the `from` or `where` clauses are stored as entire strings and thus, are not further parsed into their own dictionaries. Therefore, we are unable to search the nested queries to find any minute differences in these statements if the input SQL queries contain nested statements.

The regular expressions used in the function are:

- `select_pattern`: matches the "SELECT" keyword of the query
- `from_pattern`: matches the "FROM" keyword of the query
- `where_pattern`: matches the "WHERE" keyword of the query
- `group_by_pattern`: matches the "GROUP BY" keyword of the query
- `having_pattern`: matches the "HAVING" keyword of the query
- `order_by_pattern`: matches the "ORDER BY" keyword of the query
- `limit_pattern`: matches the "LIMIT" keyword of the query

An important point to note from SQL's syntax is that in a query, only the `select` and `from` clauses are mandatory for inclusion, while all other clauses are optional. We took this into consideration when creating the regular expressions as seen below.

```
select_pattern = re.compile(r'select (.+?) from', re.DOTALL)

from_pattern = re.compile(r'from (.+?) (?:where|group by|having|order by|limit|$)', re.DOTALL)

where_pattern = re.compile(r'where (.+?) (?:group by|having|order by|limit|$)', re.DOTALL)

group_by_pattern = re.compile(r'group by (.+?) (?:having|order by|limit|$)', re.DOTALL)

having_pattern = re.compile(r'having (.+?) (?:order by|limit|$)', re.DOTALL)

order_by_pattern = re.compile(r'order by (.+?) (?:limit|$)', re.DOTALL)

limit_pattern = re.compile(r'limit (.+?)$', re.DOTALL)
```

Fig. 3 Regular Expressions

The function then uses the `search` method of the regular expression object to search for matches in the SQL query string.

```
select_match = select_pattern.search(sql_query)
```

Fig. 4 Snippet of match searching for Select clause

If a match is found for a particular clause, the function extracts the relevant predicates of that clause and adds it to a dictionary as a value where the key is the associated SQL keyword, as shown below:

```
if select_match:
    select_clause = select_match.group(1).strip().split(',')
    for i in range(len(select_clause)):
        select_clause[i] = select_clause[i].strip()
    dict['SELECT'] = select_clause
```

Fig. 5 Snippet of key-value pair creation for Select clause

The function then returns the dictionary, which has SQL keywords as keys and a list of their predicates as their values. As an example, we have executed `parse_sql_query` using an example SQL query from the TPC-H file, '10.sql'.

```
select
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
from
    customer,
    orders,
    lineitem,
    nation
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= date ':1'
    and o_orderdate < date ':1' + interval '3' month
    and l_returnflag = 'R'
    and c_nationkey = n_nationkey
group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
```



```
n_name,  
c_address,  
c_comment  
order by  
revenue desc;
```

Fig. 6 Example input into parse_sql_query()

Based on the input from Fig 6, after passing it into the `parse_sql_query` function, the output dictionary object we obtain is as follows:

```
{  
  'SELECT': [  
    'c_custkey',  
    'c_name',  
    'SUM(l_extendedprice * (1 - l_discount)) AS revenue',  
    'c_acctbal',  
    'n_name',  
    'c_address',  
    'c_phone',  
    'c_comment'  
  ],  
  'FROM': [  
    'customer',  
    'orders',  
    'lineitem',  
    'nation'  
  ],  
  'WHERE': [  
    'c_custkey = o_custkey',  
    'l_orderkey = o_orderkey',  
    "o_orderdate >= date ':1'",  
    "o_orderdate < date ':1' + interval '3' month",  
    'l_returnflag = 'R'',  
    'c_nationkey = n_nationkey'  
  ],  
  'GROUP BY': [  
    'c_custkey',  
    'c_name',  
    'c_acctbal',  
    'c_phone',  
    'n_name',  
    'c_address',  
    'c_comment'  
  ],  
  'ORDER BY': [  
    'revenue desc';'  
  ]  
}
```

```
}      ]
```

Fig. 7 Output from parse_sql_query() with Fig. 6 as input

Based on Fig 7, we can see that for every SQL keyword that was present in the query, there is an associated key-value pair where the value is a list of all the predicates for that associated keyword. Each predicate is stored as a string, including predicates that contain inequalities or are formed using aggregate functions such as `SUM`.

3.1.3 Query Plan Generation

Query plan generation consists of the following stages:

1. A database connection is obtained
2. The query plans are obtained by executing the user-inputted queries after prepending the `explain` and `analyse` keywords
3. The resultant plans are parsed for the summary, plan comparison attributes and explanation information in natural language
4. The resulting information is then added to the result if they are not duplicates and returned
5. In the event of an error, an error message is returned instead

After which, the relevant information is extracted by parsing the JSON object to build the QEPTree using QEPNode class. The QEPTree instances generated are subsequently used for the comparison of the 2 query execution plans.

3.2 Explanation of Change

3.2.1 Query Plan Comparison

Comparing two different QEPs consist of the following general stages:

1. Generating the QEPs from PostgreSQL using the `explain` and `analyse` keywords when executing the user inputted SQL statements
2. The execution of the `explain` query returns the JSON object which contains the query execution tree. This JSON object is parsed using `extract_from_json(json_obj)` defined in the `qepGenerator.py`
3. The above function iteratively takes in QEP nodes, starting from the root and traverses down the QEP tree by accessing the children nodes of each node, until it reaches the leaf nodes
4. The root of the tree is returned which is used to instantiate a QEPTree class. Upon instantiation, it runs through a few preprocessing methods

- a. The most important method would be the `_setParentRecursive()` function which adds the attribute 'parent' in each QEP node. This allows us to traverse the tree bottom-up starting from the leaf nodes.
- b. Another critical method would be the `_generateIntermediatesPostOrder()` which traverses the tree, starting from the root, in a post-order manner to update the QEPTree instance's intermediates dictionary. As a result, we would get a dictionary for the QEPTree which contains all the intermediate results formed during query execution.

```
{
  "T0": {
    "formedbyRelations": [
      "nation",
      "supplier"
    ],
    "formedFromQEPidx": [
      5,
      6
    ],
    "formedbyQEP": "QEPnode(idx = '3', parent='2',
type='Nested Loop', join_type='Inner', relation_name='None')",
  },
  "T1": {
    "formedbyRelations": [
      "lineitem"
    ],
    "formedFromQEPidx": [
      12
    ],
    "formedbyQEP": "QEPnode(idx = '10', parent='8',
type='Aggregate', join_type='None', relation_name='None')",
  },
  "T2": {
    "formedbyRelations": [
      "part"
    ],
    "formedFromQEPidx": [
      14
    ],
    "formedbyQEP": "QEPnode(idx = '11', parent='8',
type='Hash', join_type='None', relation_name='None')"
```

```

    }
}

```

Fig. 8 Intermediate Dictionary

5. Subsequently, we parse this intermediate dictionary to form another dictionary that contains how two raw relations are being joined, be it directly or indirectly through intermediates. An example of how this dictionary looks like is as follows:

a.

```

{
    "supplier + nation": "QENode(idx = '3', parent='2',
type='Nested Loop', join_type='Inner', relation_name='None')",

    "lineitem + orders": "QENode(idx = '10', parent='8',
type='Nested Loop', join_type='None', relation_name='None')",

    "customer + region": "QENode(idx = '11', parent='8',
type='Hash Join', join_type='None', relation_name='None')"
}

```

Fig. 9 Join Dictionary

- b. The rationale behind this is that when a query changes, **there can be significant changes in the tree structure of the query execution plan**. Therefore, **comparing the two trees level by level may result in irrelevant comparisons** - for instance, comparing a 'sort' and a 'index join'. Hence, our comparisons mainly revolve around comparing how the raw relations are being joined together, either directly or indirectly through intermediates.
- c. It should be noted that the QENode in the above dictionary does not represent all the attributes of the QENode instance so as to improve the readability of the figure and the other attributes of the QENode. Other attributes such as join conditions are still accessible.

The above dictionaries are generated for each of the QEPTree. We then use the two dictionaries from each QEPTree to make our comparison. For example, we check if all combinations of joins (eg: "supplier + nation") are present in the second QEPTree.

- a. If the keys are different in the two trees, we first filter out the additional entries and the entries that are removed from QEPTree1 to QEPTree2.
- b. Once this is complete, for each key:value pair in the dictionary of QEPTree1, we check whether it has the same key:value pair in the dictionary of QEPTree2. If it exists, it means that, for example if supplier and nation relations are joined using nested loop join in the QEPTree1, they are also joined using nested loop join in the QEPTree2 as well.

- c. If the value in the QEPTree2's dictionary for the same key in the QEPTree1's dictionary is not the same as the value in the QEPTree1, it means the join between the relations specified in the key is joined using different type of join **or** joined with a different condition.
 - i. The way we identify direct joins or indirect joins through intermediates is to check with another dictionary which contains all the direct joins of the QEPTree.
 - ii. If there are mismatches, a natural language explanation string is added to a list of differences, which is returned at the end of the function call.
- d. Another check we perform is to see whether the scan methods remain the same. There may be cases where, for example, sequential scan is performed on relation 'nation' in QEPTree1 but index scan is performed on the same relation in the QEPTree2. These differences are also appended to the difference list.
- e. The difference list is then passed back to the frontend which is then displayed after some formatting.

3.2.2 Query Comparison

Query comparison consists of the following general stages:

1. Parsing the first SQL statement
2. Parsing the second SQL statement
3. Comparing all the predicates for each SQL clause of both statements. In our comparison, we account for the following possible cases:
 - a. Case 1: The clause exists in both queries
 - i. Case 1.1: The predicates of a clause remain unchanged
 - ii. Case 1.2: The predicates of a clause have changed
 - b. Case 2: The clause is only present in one of the queries
 - i. Case 2.1: The clause is present in query 1 but not query 2
 - ii. Case 2.2: The clause is present in query 2 but not query 1
 - c. Case 3: The clause is not used in either queries at all (logic covered implicitly)

We define a function called `query_compare` that takes 2 SQL queries as input variables. This function calls the earlier defined function `parse_sql_query`, creating the two dictionaries `query_1_dict` and `query_2_dict`. It then iterates through both dictionaries, searching for a clause from the list of possible clauses, and checking to see which case is being met for each clause. Finally, it returns `changes_list`.

```
def query_compare(query1, query2):  
  
    query_1_dict = parse_sql_query(query1)  
    query_2_dict = parse_sql_query(query2)
```

```

changes_list = []

    for key in ['SELECT', 'FROM', 'WHERE', 'GROUP BY', 'HAVING', 'ORDER BY',
'LIMIT']:

```

Fig. 10 Function call of query_compare()

The natural language explanation of the changes between Query 1 and Query 2 is generated differently for each of the cases. The statements that generate the text explanation themselves have been bolded for easy reference, as shown through the following tables:

```

# Case 1: key exists in both queries
if key in query_1_dict and key in query_2_dict:
    # Case 1.1: value in key is unchanged
    if query_1_dict[key] == query_2_dict[key]:
        changes_list.append(f"Predicates for the {key} clause remained the same")
    # Case 1.2: value in key has changed
    else:
        old_value = query_1_dict[key]
        new_value = query_2_dict[key]
        present_in_old_not_new = []
        present_in_new_not_old = []
        # Iterate over each element in the old and new values
        for i, elem in enumerate(old_value):
            # Check if the element was present in the new value
            if elem not in new_value:
                present_in_old_not_new.append(f"{elem}")
        for i, elem in enumerate(new_value):
            # Check if the element was present in the old value
            if elem not in old_value:
                present_in_new_not_old.append(f"{elem}")

```

```

changes = []

if present_in_old_not_new:
    changes.append("Was present in Query 1 but not Query 2: " + ",
".join(present_in_old_not_new))

if present_in_new_not_old:
    changes.append("Was not present in Query 1 but is now present in
Query 2: " + ", ".join(present_in_new_not_old))

changes_list.append(f"Predicates for the {key} clause have
changed:")

if changes:
    for change in changes:
        changes_list.append(change)

```

Fig. 11 Case 1: SQL clause present in both Query 1 and Query 2

```

# Case 2: keys not present in one of the two
else:
    # Case 2.1: key present in query1 but not query2
    if key in query_1_dict and key not in query_2_dict:
        changes_list.append(f"The {key} clause was present in Query 1 but is
now not present in Query 2")

    # Case 2.2: key present in query2 but not query1
    if key in query_2_dict and key not in query_1_dict:
        changes_list.append(f"The {key} clause was not present in Query 1
but is now present in Query 2")

```

Fig. 12 Case 2: SQL clause present in either Query 1 or Query 2 (but not both)

As an example, we have executed `query_compare(query1, query2)` by taking the following two sample queries as input:

```

query1 = "
select
    column1,
    column2,
    column3
from
    table1,
    table2,
    table3
where
    column1 > 10,
    and column2 = "example"
group by
    column1,
    column2
order by
    column1 desc,
    column2 asc;
"

query2 = "
select
    column1,
    column3
from
    table1,
    table3
group by
    column1,
    column2
order by
    column1 desc,
    column3 asc
limit
    100;
"

```

Fig. 13 Example inputs into `query_compare()`

Based on the input from Table 13 into the comparison function, the output we obtain is as follows. The output has been bolded for easy reference to above.

```

['Predicates for the SELECT clause have changed:', 'Was present in Query 1 but not Query 2:
column2', 'Predicates for the FROM clause have changed:', 'Was present in Query 1 but not
Query 2: table2', 'The WHERE clause was present in Query 1 but is now not present in Query
2', 'Predicates for the GROUP BY clause remained the same', 'Predicates for the ORDER BY

```

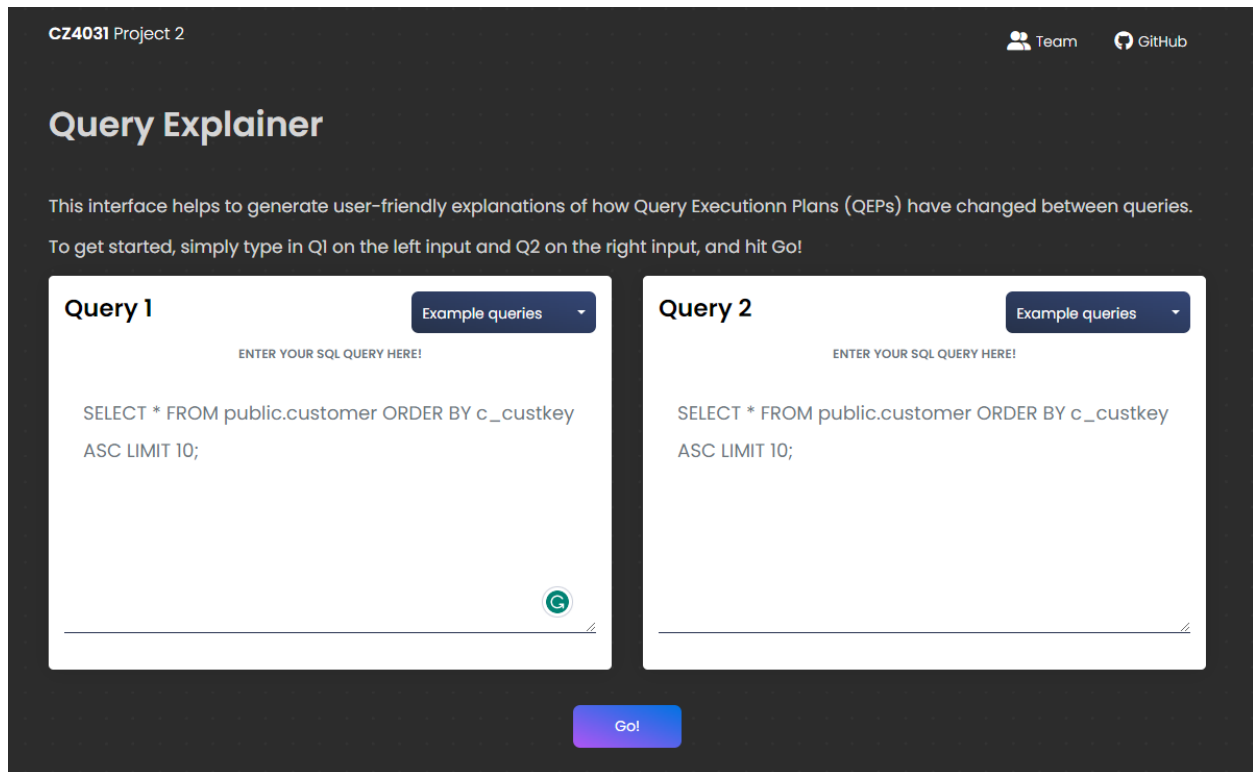

clause have changed:', 'Was present in Query 1 but not Query 2: column2 ASC', 'Was not present in Query 1 but is now present in Query 2: column3 ASC', 'The LIMIT clause was not present in Query 1 but is now present in Query 2']

Fig. 14 Output from query_compare() with Fig. 13 as inputs

4. User Interface

The interface was implemented using the Flask web framework. Flask utilizes the Jinja templating engine that performs server-side rendering, allowing us to dynamically render HTML/CSS web pages to the user. The use of a web-based front-end allowed us to be more flexible in designing more beautiful user interfaces compared to that of python-based GUI libraries.

Fig 15 shows the general layout of our interface.



The screenshot displays the 'Query Explainer' web application. At the top, the header includes 'CZ4031 Project 2' on the left and 'Team' and 'GitHub' links on the right. The main title 'Query Explainer' is prominently displayed. Below the title, a descriptive paragraph states: 'This interface helps to generate user-friendly explanations of how Query Execution Plans (QEPs) have changed between queries. To get started, simply type in Q1 on the left input and Q2 on the right input, and hit Go!'. The interface features two side-by-side input boxes for 'Query 1' and 'Query 2'. Each box has a title, a placeholder 'ENTER YOUR SQL QUERY HERE!', a dropdown menu for 'Example queries', and a text area containing the SQL query: 'SELECT * FROM public.customer ORDER BY c_custkey ASC LIMIT 10;'. A green circular icon with a 'G' is visible in the bottom right corner of the Query 1 input area. At the bottom center, there is a blue 'Go!' button.

Fig. 15 Interface Layout

4.1 Query Input Section

The query input section Figure 16 provides two text forms for users to input two queries Q1 and Q2 for comparison. It is generally assumed that Q1 and Q2 are related as Q2 must have evolved from Q1 to have a basis for comparison, hence no checks are needed on the frontend for the inputs. Example queries are provided in the dropdown list for user convenience. The placeholder in the input form also helps to guide new users on how to write their queries. Upon clicking the 'Go!' button, the 2 queries will be submitted to the backend for processing. This is done by sending a POST request containing json data of the two queries. If either of the forms are empty during submission, the interface will prompt the user where a query is missing.

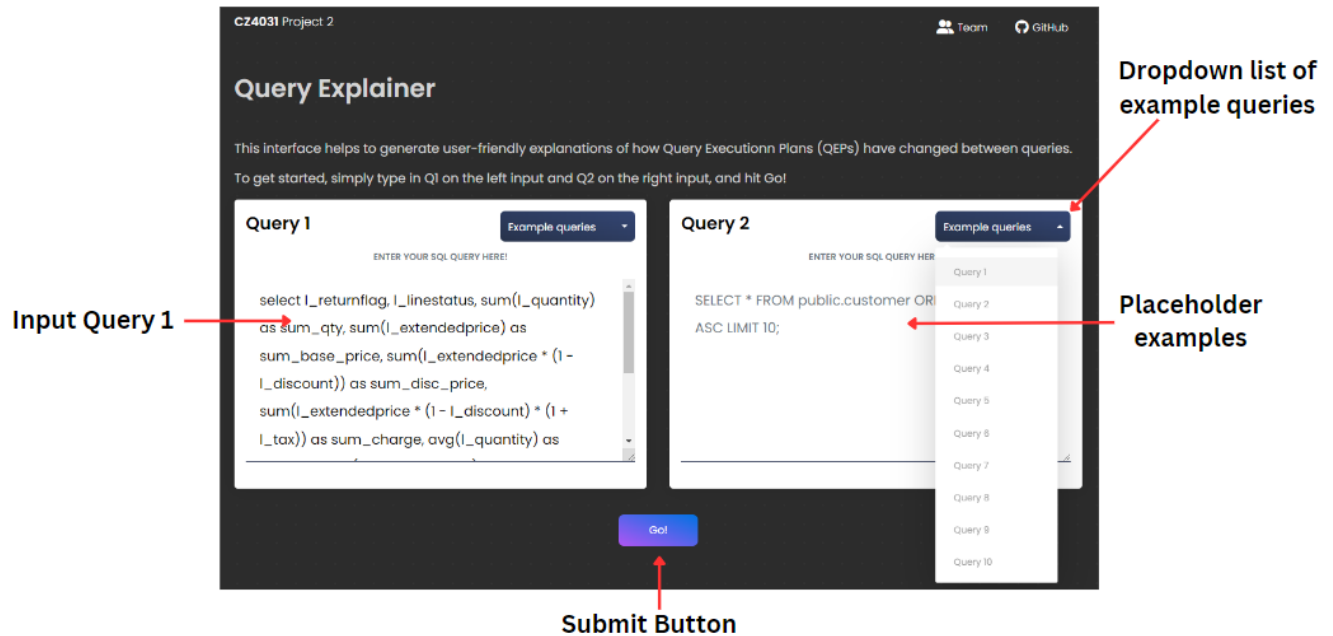


Fig. 16 Input forms with prompts to guide users on writing queries

4.2 Results Section

Upon receiving QEP data and an explanation from explain.py, Flask re-renders the template with newly received data to populate a results section on the interface below the query section (Fig 17). Two query plans P1 and P2 are generated side by side.

Each query plan contains the following information:

1. Total cost: Sum of costs of each operation executed according to the query plan.
2. Number of operations nodes in the query plan.
3. Graph: Graphical explanation of the query plan produced by *amCharts*. Due to space constraints, the user will need to click on Display Graph to view the graph.
4. Description: Step-by-step natural language explanation of the query plan.

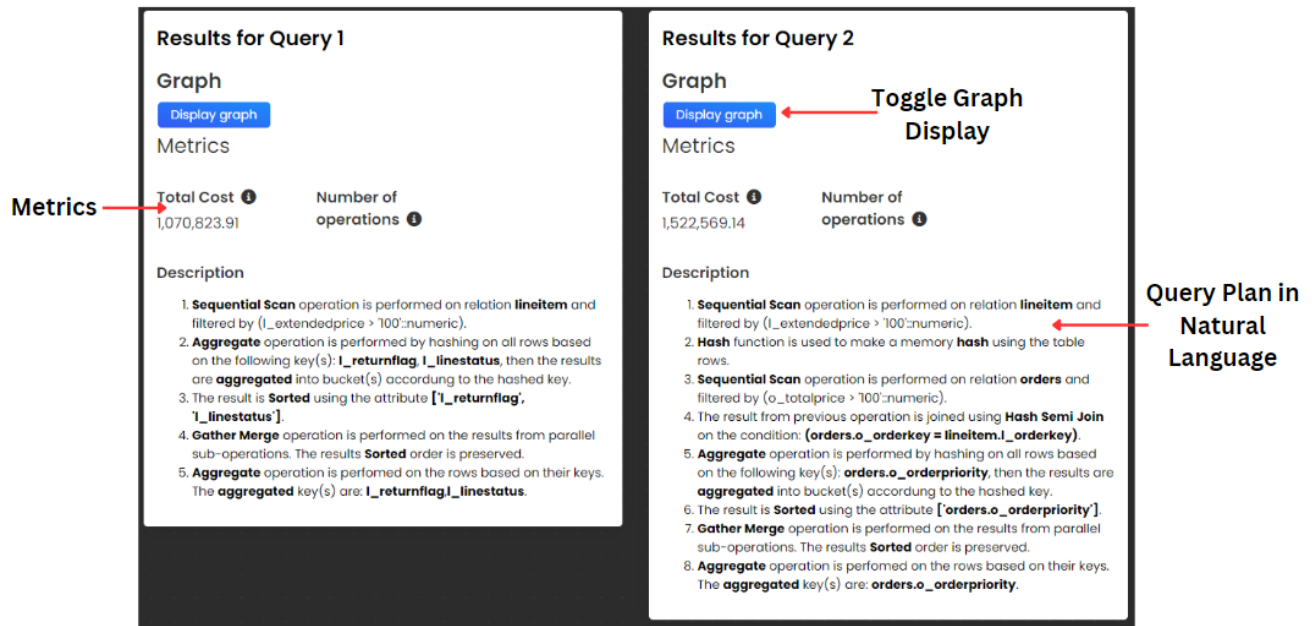


Fig. 17 Results Section containing Query Plans

On toggling the graph display (Fig. 18), a pop-up will appear showing a tree graph for the corresponding query plan. The nodes represent operators such as Sequential Scan, Hash Join, Sort and Hash. Users can mouse over nodes to view metrics such as cost of operation and join conditions.

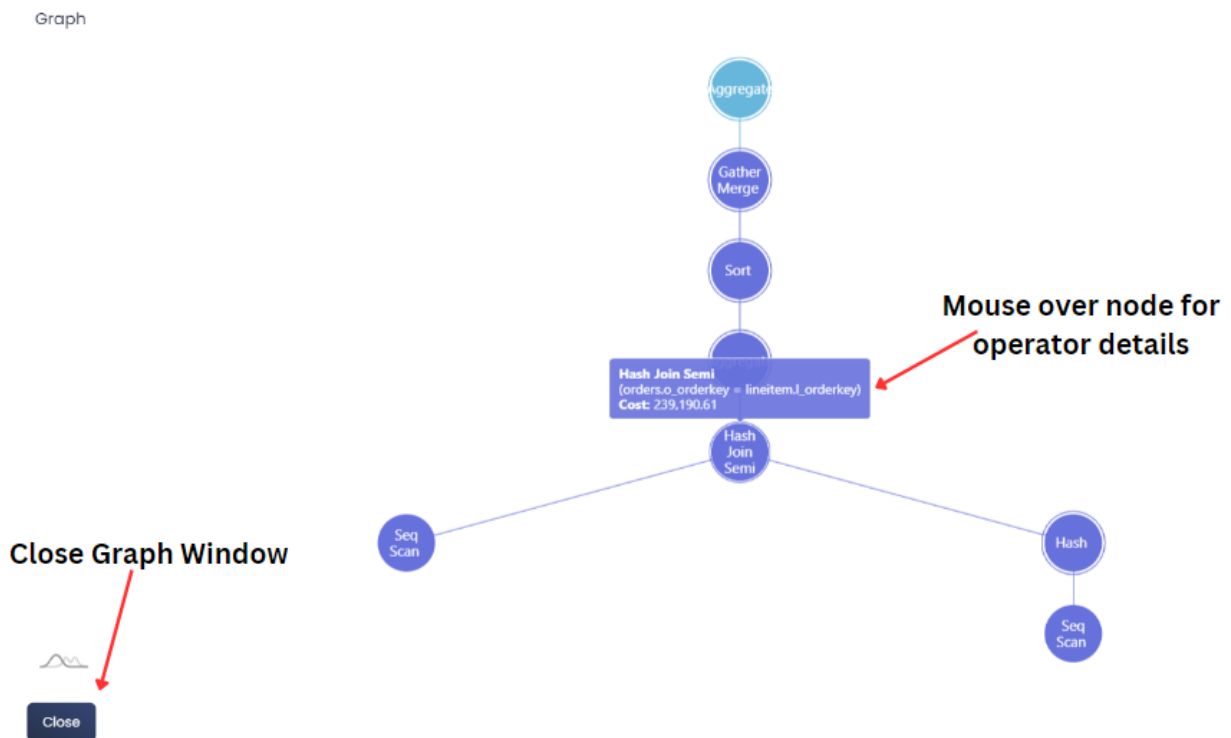


Fig. 18 Graphical Explanation of single Query Plan

Lastly, we utilise query_parser and getComparison to generate natural language explanations of query change and query plan change respectively and display them in the “Differences” section.

Consider Q1 and Q2 in the table below, with the only difference being the additional WHERE clause:

Q1	Q2
<pre>select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice * (1 - l_discount)) as sum_disc_price, sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from lineitem group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus;</pre>	<pre>select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice * (1 - l_discount)) as sum_disc_price, sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from lineitem where l_extendedprice > 100 group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus;</pre>

With Q1 and Q2 as inputs, the differences are clearly explained in bullet point form in Figure 19.

<h3>Differences in Query</h3> <ul style="list-style-type: none"> • Predicates for the SELECT clause remained the same • Predicates for the FROM clause remained the same • The WHERE clause was not present in Query 1 but is now present in Query 2 • Predicates for the GROUP BY clause remained the same
<h3>Differences in Query Execution Plan</h3> <ul style="list-style-type: none"> • Both QEPs involve same scan method for each relation

Fig. 19 Query Difference and QEP Difference interface cards

5. Assumptions and Limitations

1. Queries only use these common clauses [`select`, `from`, `where`, `group by`, `having`, `order by`, `limit`]. The choice of clauses was generated by viewing the example queries in the TPC-H dataset, and various online sources.
 - a. The code assumes that the input queries only use the common clauses that we have defined in the code. If the input queries contain other clauses, the parsing and comparison functions may not pick up on them, thus missing out some information from the query.
2. Queries are written with an accurate syntax, with the clause order being `select`, `from`, `where`, `group by`, `having`, `order by`, `limit`. Although all clauses other than `select` and `from` are optional, they have to follow this sequence if they are used.
 - a. The code assumes that the input queries are written in a specific clause order. If the input query has clauses in the wrong order, the regular expressions created to match each clause will result in flawed matches. Therefore, it is important for the user to ensure that the input queries follow an accurate SQL syntax, to obtain accurate results.