CZ4031: Database System Principles

Group 8

Project 2 Report

Name	Matriculation Number	Contributions
CHIU WEI JIE, REEVES	U2023200F	SQL Differences & Explanation, Report
GUCON NAILAH GINYLLE PABILONIA	U2021643H	Interface Implementation, Code Integration, Report
IVAN PUA JUN HAO	U2021665H	QEP Differences & Explanation, Report
NUR DILAH BINTE ZAINI	U2022478K	Interface Implementation, Code Integration, Report

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

Table of Content

Introduction

Project Methodology

Develop an application that automatically generates user-friendly explanations (e.g., natural and visual language description) of the changes to the SQL statements and query execution plans that take place during data exploration.

TPC-H Dataset

The TPC-H dataset inserted into PostgreSQL is designed for measuring the performance of complex decision-support databases and includes schemas with varying amounts of data for developing and testing SQL queries.

Below is a summary of the relations found in the generated PostgreSQL database:

Table 1: TPC-H Dataset

Table	Description	# of Tuples
CUSTOMER	Contains the list of all customers and their details	150000
LINEITEM	Contains the list of all transport lines and their details	6001215
NATION	Contains the list of available nations	25
ORDERS	Contains the list of all orders and their details	1500000
PART	Contains the list of all parts and their details	200000
PARTSUPP	Contains the list of all the suppliers of different parts	800000
REGION	Contains the list of all the region	5
SUPPLIER	Contain the list of all the suppliers and their details	10000
		Total: 8661245

We will be using PostgreSQL to optimize queries on the large and complex TPC-H dataset, which can be time-consuming and costly without a query optimizer.

Prerequisites

Do follow the steps below to be able to run our application.

1. Edit the `config` variable in explain.py based on your own configurations

```
class CursorManager(object):
    def __init__(self):
        config = {
                "host": "localhost",
                "dbname": "TPC-H",
                "user" : "postgres",
                "pwd" : "Dsp12345",
                "port" : "5432"
        }
...
```

dbname: Database's name that is created in local PostgreSQL
(Please change to your dbname according to the database you created)
pwd: Password that is set for the PostgreSQL's server
(Please change the password according to the password you set)

2. Install libraries entering the code in the terminal

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

3. Execute the program

```
python project.py
```

Explain Implementation

Explain High Level Concept

The overarching idea behind Explain.py is to serve as a backend tool that can be utilized by the interface to execute the required functions and provide the corresponding outcomes.

In essence, Explain.py can be divided into three distinct sections, each serving a specific purpose in this process.

Table 2: Explain Section Descriptions

Sections	Description
Querying postgre for QEP	SQL query is pass into explain and a function
Store the QEP in a define structure	String representing the type of operation performed at this node
Analyze and the SQL and QEP given	String representing additional details about the operation performed at this node

The `Explain` class essentially serves as a wrapper for all QEP related functions. In addition it holds the `CursorManager` which is responsible for the connection with the postgreSQL server as well as fetching the results for each query provided.

QEP Difference & Explanation

Overview

We define two classes, `QEP_Node` and `QEP_Tree`, that are used to build and represent the Query Execution Plan (QEP).

The `QEP_Node` class represents a single node in the QEP. It has the following attributes:

Table 3: QEP_Node Attributes List

Attribute Name	Attribute Description
indent_size	Number of indents in explanation to determine the depth of which the node is at
operation	String representing the type of operation performed at this node
details	String representing additional details such as the cost about the operation performed at this node
parent	Reference to the parent node of this node in the tree
children	List of references to the child nodes of this node in the tree
explanation	List of strings representing any additional explanations associated with this node

The `QEP_Tree` class represents the entire QEP tree. It has the following attributes:

Table 4: QEP_Tree Attributes List

Attribute Name	Attribute Description
root	Reference to the root node of the tree
prev_indent_size	Integer representing the indent size of the previous node processed in the tree

The `QEP_Tree` class defines the tree structure of the QEP. Its role is to maintain the structure of the QEP and the references to each node. This way, we can easily traverse the QEP tree using different tree traversal methods to achieve our desired results.

Code and examples for key functions and algorithms such as `QEP_tree().build()`, `QEP_tree().get_explanation()` and `QEP_tree().compareQEP()` are shown in **Appendices A, B and C** respectively.

Description of Key Algorithms

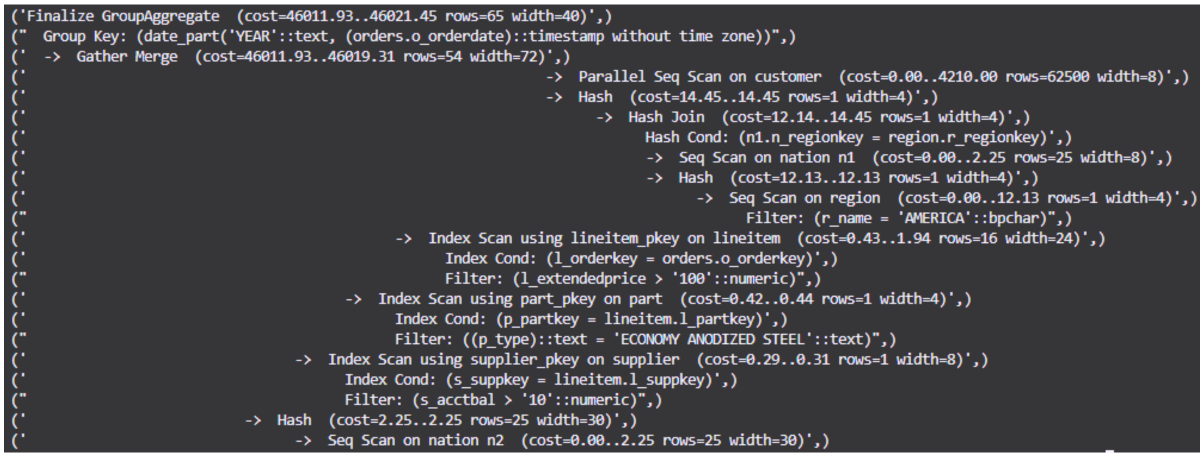
Tree Building

The `QEP_Tree` class has a `build()` method which takes the given plan consisting of a list of strings, returned by postgreSQL as an argument. It iterates through the plan and constructs a `QEP_Node` for each operation. The `QEP_Node` contains all the relevant information regarding each operation which includes the parent node, associated children, operation name, details regarding the cost of the operation, indent size to determine the level of the node, as well as the explanation for the given operation. The result of the `QEP_Tree` is a binary tree representation of operations in the QEP.

For each row in the plan, regular expressions (regex) are used to split the string into 3 different parts (indent_size, operation, details). Based on the size of the indentation in the string, we can determine the level of the node and therefore identify which parent it belongs to. For example, if 2 rows have the same level of indentation, they are both on the same level (i.e, sibling nodes). This enables us to find and attach them to the parent node easily. To determine if a row satisfies an operation, it should include "->" in the row. Otherwise it would be treated as an explanation.

In addition, we have to handle child nodes that appear further down the plan since they will not have the same indent size as the previous row. To handle this, when we encounter a row with its indent size greater than the previous row, it would suggest that this row is a child of a parent that was created further up in the tree. Hence, we have to find its sibling with the same indent size and return its parent. To find its parent, we traverse the tree iteratively using an iterative version of the depth-first search algorithm, starting with the root node. Once we encounter its sibling node (i.e, same indent size) we return its parent node. Then, we can attach the current node to the parent that we have found

Finally, the `build()` method will return the root node of the QEP once it is done constructing the tree. The figure below shows an example of a plan returned by postgreSQL which will be parsed by the `build()` method.



```
('Finalize GroupAggregate  (cost=46011.93..46021.45 rows=65 width=40)',)
("  Group Key: (date_part('YEAR'::text, (orders.o_orderdate)::timestamp without time zone))",)
('  ->  Gather Merge  (cost=46011.93..46019.31 rows=54 width=72)',)
('                                                              ->  Parallel Seq Scan on customer  (cost=0.00..4210.00 rows=62500 width=8)',)
('                                                              ->  Hash  (cost=14.45..14.45 rows=1 width=4)',)
('                                                                    ->  Hash Join  (cost=12.14..14.45 rows=1 width=4)',)
('                                                                          Hash Cond: (n1.n_regionkey = region.r_regionkey)',)
('                                                                          ->  Seq Scan on nation n1  (cost=0.00..2.25 rows=25 width=8)',)
('                                                                          ->  Hash  (cost=12.13..12.13 rows=1 width=4)',)
('                                                                                ->  Seq Scan on region  (cost=0.00..12.13 rows=1 width=4)',)
("                                                                                      Filter: (r_name = 'AMERICA'::bpchar)",)
('                                            ->  Index Scan using lineitem_pkey on lineitem  (cost=0.43..1.94 rows=16 width=24)',)
('                                                  Index Cond: (l_orderkey = orders.o_orderkey)',)
("                                                  Filter: (l_extendedprice > '100'::numeric)",)
('                                      ->  Index Scan using part_pkey on part  (cost=0.42..0.44 rows=1 width=4)',)
('                                            Index Cond: (p_partkey = lineitem.l_partkey)',)
("                                            Filter: ((p_type)::text = 'ECONOMY ANODIZED STEEL'::text)",)
('                                ->  Index Scan using supplier_pkey on supplier  (cost=0.29..0.31 rows=1 width=8)',)
('                                      Index Cond: (s_suppkey = lineitem.l_suppkey)',)
("                                      Filter: (s_acctbal > '10'::numeric)",)
('                          ->  Hash  (cost=2.25..2.25 rows=25 width=30)',)
('                                ->  Seq Scan on nation n2  (cost=0.00..2.25 rows=25 width=30)',)
```

QEP Explanation

The `QEP_Tree` has a `get_explanation()` method which parses the explanation of each `QEP_Node` in the respective QEP_Tree. Since postgreSQL uses Left Deep trees to do query planning by default, the `get_explanation()` method takes in the root node of the QEP tree that we want to get the explanation for, as well as an empty array to store the results. According to the Left Deep tree specifications, operations on the left side of the tree gets executed first. Hence, this method traverses the QEP tree in a post order fashion, to correspond to the Left Deep tree execution order.

The `get_explanation()` method transforms the explanations of each node into a more refined natural language version by parsing each explanation by adding new words depending on the operation and explanation to form a sentence.

Finally, a list of explanations is returned.

Comparing QEPs

The `QEP_Tree` has a `compareQEP()` function which takes both root nodes of the 2 QEP trees we want to compare as arguments. It first generates a queue of nodes for each tree by traversing through the tree in a post order fashion. Again, following the Left Deep tree execution order. Once these queues are generated, the queues are iterated simultaneously and each node is compared with the other from opposite queues. The comparison is then parsed based on whether the operations and explanations differ from each other. When one of the queues is “empty”, we continue iterating the queue that still has nodes as it has additional operations not found in the latter query plan due to changes in the SQL query.

Finally, a list of comparisons is returned.

SQL Difference & Explanation

Overview

We developed a feature that facilitates the analysis of SQL query differences through a systematic and comprehensive process.

Firstly, we applied a flagging mechanism to identify any changes made between the "Old SQL" and "New SQL" versions, thus providing a clear and structured representation of the differences. Subsequently, we conducted an in-depth analysis of the output, extracting valuable insights that can aid users in identifying potential typo errors or areas of change. Finally, we developed a concise and user-friendly explanation of the differences, enabling users to comprehend the modifications made to the SQL queries effectively.

This approach significantly enhances the users' ability to identify and analyze any differences between SQL queries. By providing meaningful insights and clear explanations, our feature aids users in the process of enhancing the quality of their SQL query development and optimizing the query performance.

Code and examples for key functions and algorithms such as `compare_sql()` and `explainSQL()` are shown in **Appendices D & E** respectively.

Descriptions of Key Algorithms

Comparing SQL Queries

The `compare_sql` function takes two SQL query strings as input and compares them line by line using the Differ class from the difflib module. It then stores the differences between the two queries in a list of tuples, where the first element of each tuple is a string indicating the type of difference (e.g. "SQL old removed" or "SQL new added"), and the second element is the text of the changed line. The function returns a list of lists, where each sublist contains the tuples representing the differences between two consecutive lines.

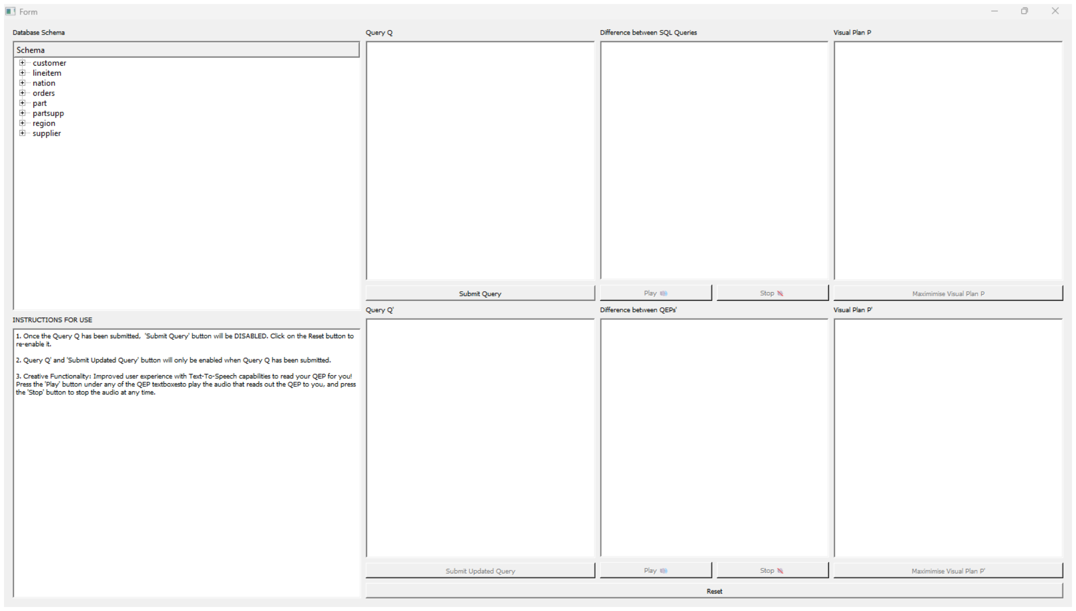
SQL Explanation

The `explainSQL` function takes the output of `compare_sql` as input and uses it to generate a list of strings explaining the differences between the two queries. It does this by iterating through the list of differences and comparing the words in the changed lines. Depending on the type of change (e.g. a change in value, a change from a value to a variable, etc.), it generates an appropriate explanation string and appends it to the output list.

Interface Implementation

Choice of Interface Framework

For our project's interface, we opted for the PyQt library, built on Qt, which allows us to use the graphical editor Qt Designer for easy implementation of components. We began by designing our interface on Qt Designer and later exported it as Python code, where we applied additional styling and functionality to make the visual appearance and experience better for the user.

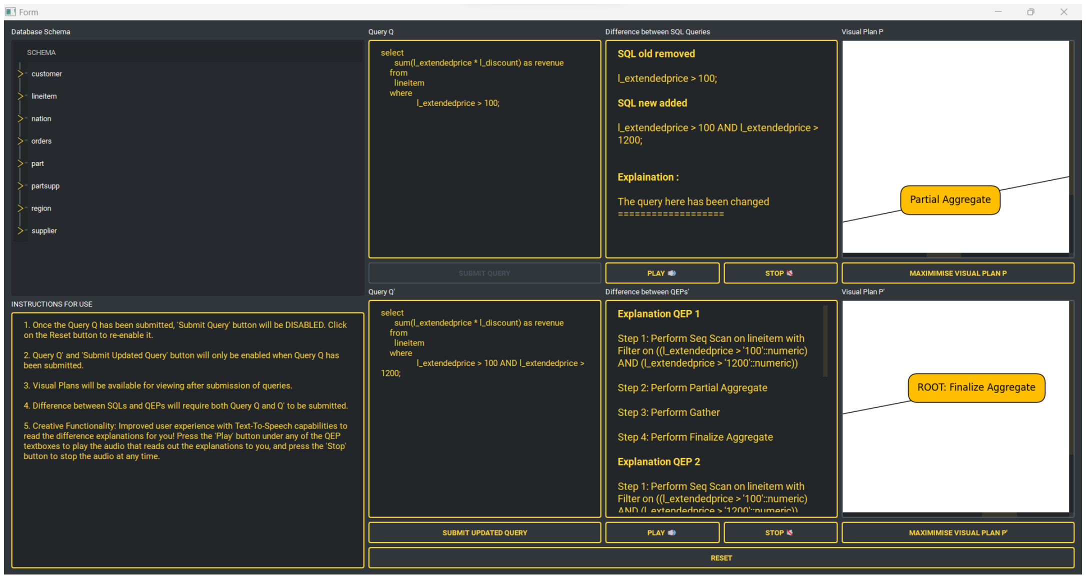


1. Once the Query Q has been submitted, 'Submit Query' button will be DISABLED. Click on the Reset button to re-enable it.

2. Query Q' and 'Submit Updated Query' button will only be enabled when Query Q has been submitted.

3. Creative Functionality: Improved user experience with Text-To-Speech capabilities to read your QEP for you! Press the 'Play' button under any of the QEP textboxesto play the audio that reads out the QEP to you, and press the 'Stop' button to stop the audio at any time.

Figure 1: Initial Design of Application



1. Once the Query Q has been submitted, 'Submit Query' button will be DISABLED. Click on the Reset button to re-enable it.

2. Query Q' and 'Submit Updated Query' button will only be enabled when Query Q has been submitted.

3. Visual Plans will be available for viewing after submission of queries.

4. Difference between SQLs and QEPs will require both Query Q and Q' to be submitted.

5. Creative Functionality: Improved user experience with Text-To-Speech capabilities to read the difference explanations for you! Press the 'Play' button under any of the QEP textboxes to play the audio that reads out the explanations to you, and press the 'Stop' button to stop the audio at any time.

```
select
   sum(l_extendedprice * l_discount) as revenue
  from
   lineitem
  where
        l_extendedprice > 100;
```

SQL old removed

l_extendedprice > 100;

SQL new added

l_extendedprice > 100 AND l_extendedprice > 1200;

Explaination :

The query here has been changed
==================

```
select
   sum(l_extendedprice * l_discount) as revenue
  from
   lineitem
  where
        l_extendedprice > 100 AND l_extendedprice >
1200;
```

Explanation QEP 1

Step 1: Perform Seq Scan on lineitem with Filter on ((l_extendedprice > '100'::numeric) AND (l_extendedprice > '1200'::numeric))

Step 2: Perform Partial Aggregate

Step 3: Perform Gather

Step 4: Perform Finalize Aggregate

Explanation QEP 2

Step 1: Perform Seq Scan on lineitem with Filter on ((l_extendedprice > '100'::numeric) AND (l_extendedprice > '1200'::numeric))

Figure 2: Final Design of Application

Query and Differences Displays

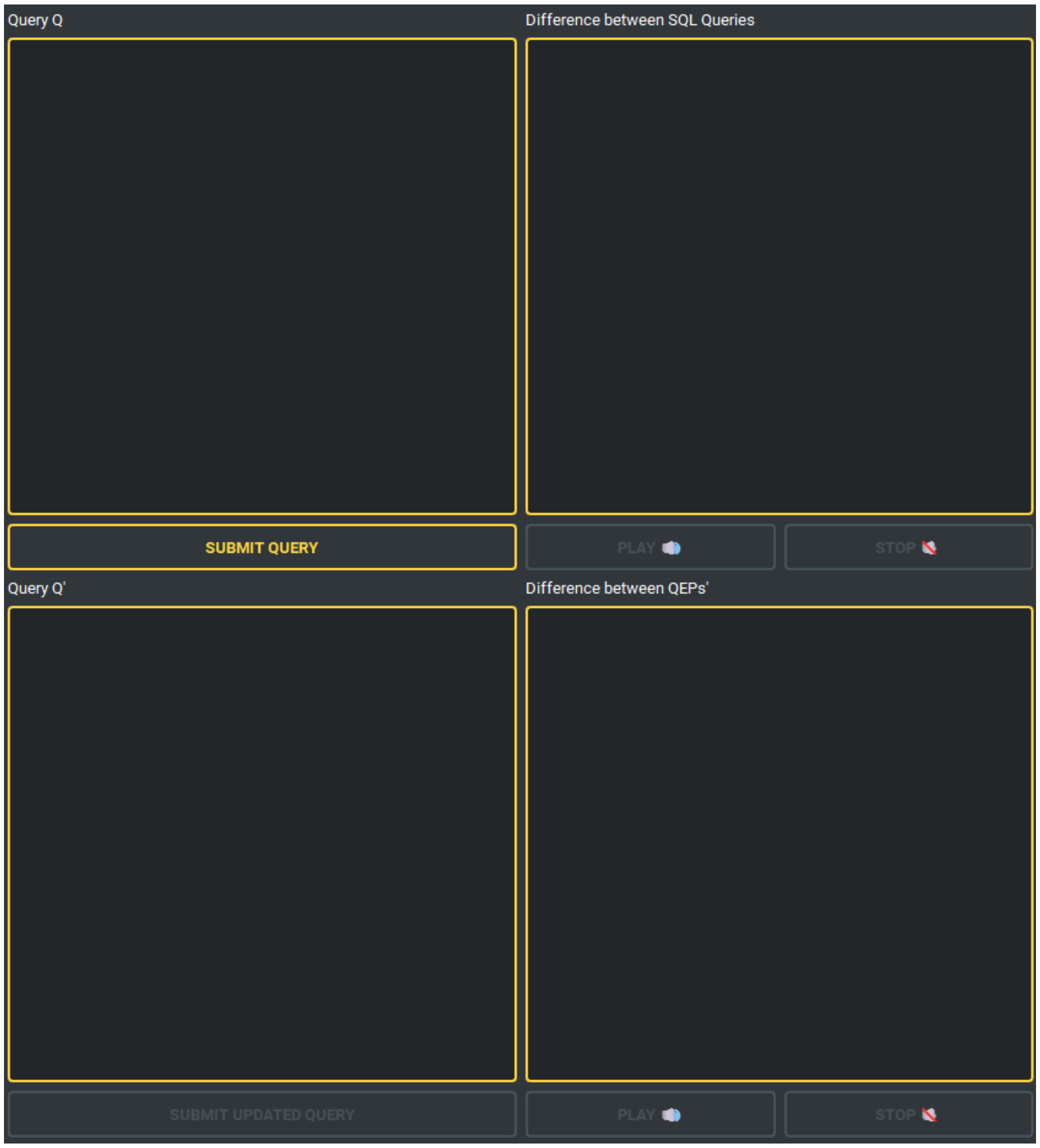


Figure 3: TextBrowsers for Queries and Difference Explanations

We have incorporated editable text browsers to facilitate user input for the queries and non-editable text browsers to display the differences between the SQL queries and QEPs.

“Creative Feature”: Text-to-Speech

One of the unique features we added to the project is the integration of text-to-speech functionality into the QEP output, which enhances the overall user experience.

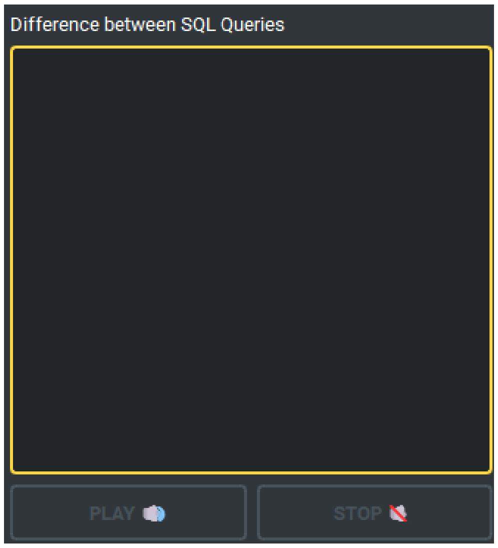


Figure 4: Text to Speech for Difference between SQL Queries

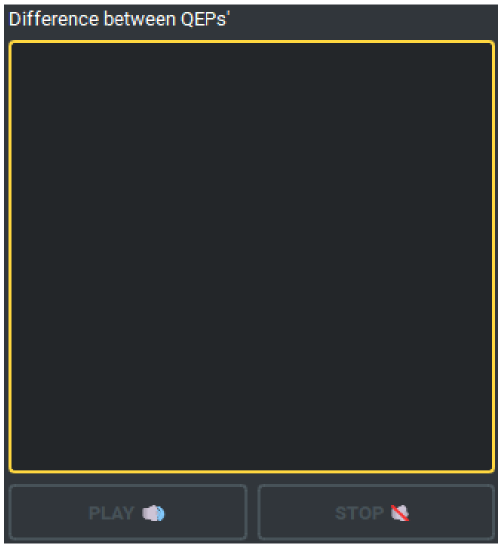


Figure 5: Text to Speech for Difference between QEPs

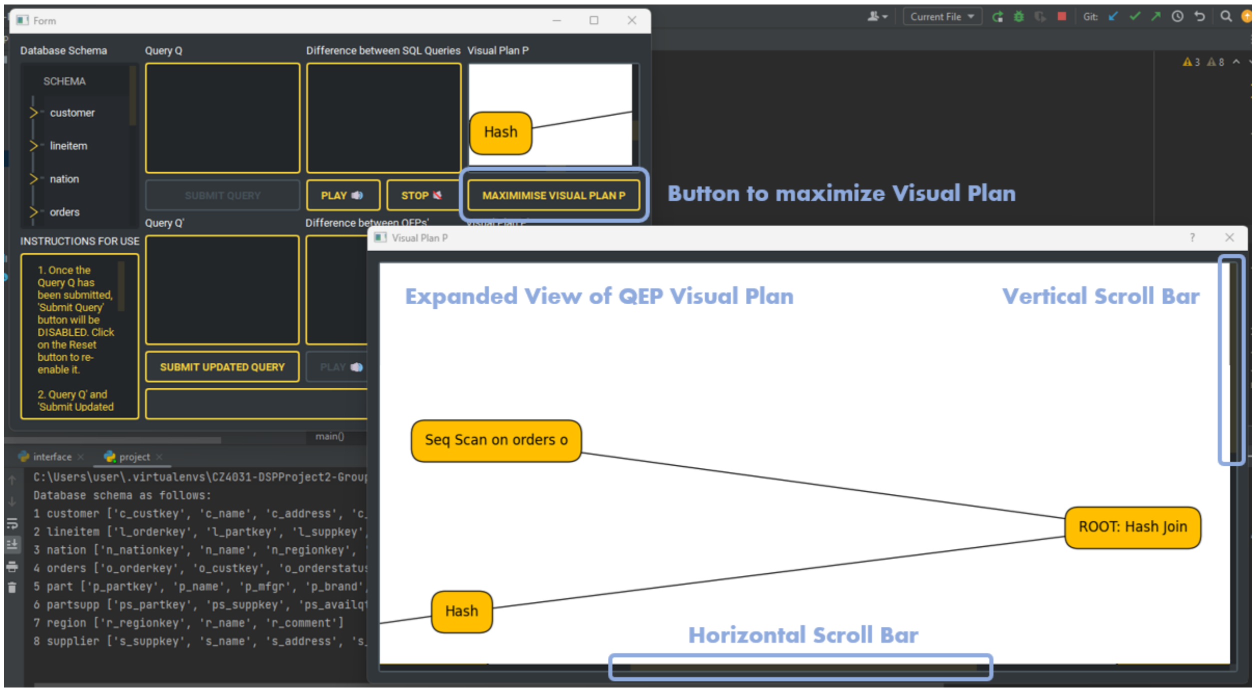
Implementing text to speech capabilities can enhance the user experience by providing an additional way for users to access information. Users can choose to listen to the output instead of reading it, which can be helpful for those with visual impairments or those who prefer auditory learning.

Visual Plan

Our application also provides visual plans of the queries in the form of graphs, which can be expanded by clicking on a button for better visualization.

We used NetworkX with Matplotlib to create the visual plans as it allows us to represent the query execution plan as a directed acyclic graph, where nodes represent the operations and edges represent the data flow between them, which can make it easier to understand the relationships between the various steps in the plan. Additionally, the ability to customize the visualizations with Matplotlib allows us to add additional information to the graph, such as colors or labels, to further aid in visualizing the execution plan.

In addition to their flexibility and ease of use, an advantage of using NetworkX with Matplotlib for creating the visual plans is their speed. They are optimized for fast graph rendering, allowing for quick and efficient visualization of even complex and large queries, making sure the user does not have to wait long before they can see the output of their query.



1. Once the Query Q has been submitted, 'Submit Query' button will be DISABLED. Click on the Reset button to re-enable it.

2. Query Q' and 'Submit Updated

```
C:\Users\user\.virtualenvs\CZ4031-DSPProject2-Grou
Database schema as follows:
1 customer ['c_custkey', 'c_name', 'c_address', 'c
2 lineitem ['l_orderkey', 'l_partkey', 'l_suppkey',
3 nation ['n_nationkey', 'n_name', 'n_regionkey',
4 orders ['o_orderkey', 'o_custkey', 'o_orderstatu
5 part ['p_partkey', 'p_name', 'p_mfgr', 'p_brand',
6 partsupp ['ps_partkey', 'ps_suppkey', 'ps_availq
7 region ['r_regionkey', 'r_name', 'r_comment']
8 supplier ['s_suppkey', 's_name', 's_address', 's
```

Figure 6: Visual Plan Functionality

Note: The starting point of the QEP is labeled with the word "ROOT" in the graph.

Reason behind Visual Plan

It enables users to visualise QEPs in a graph format. QEPs may be too complicated to be described in the text to the users. With the aid of a visual plan, users are able to understand the QEPs better.

It is built in a form of graph structure to allow for more flexible and intuitive representations of the relationships between different parts of a query. While a tree structure can represent the logical flow of a query in a hierarchical manner, it does not capture the complexity of the relationships between different parts of the query.

Schema

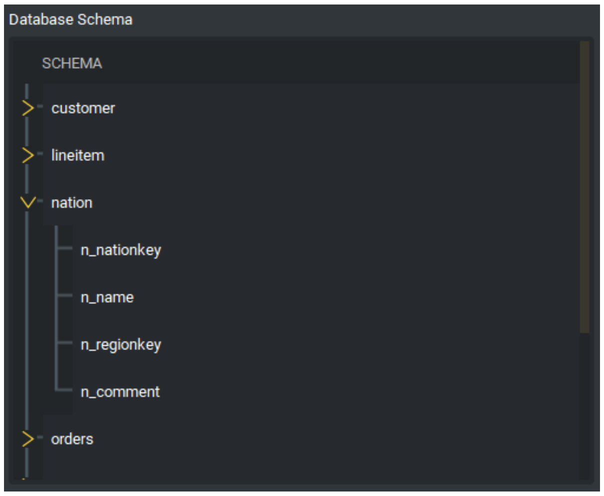
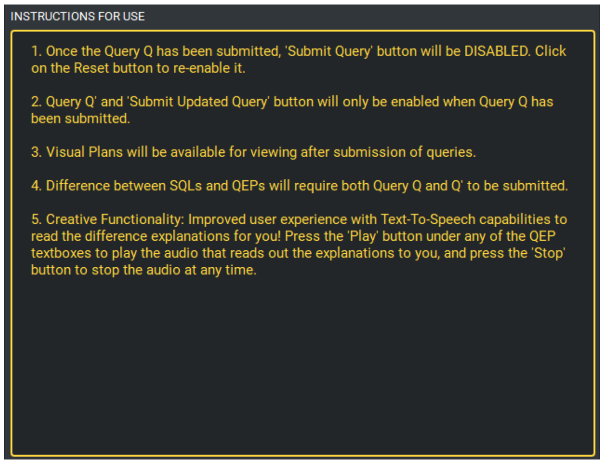


Figure 7: Schema Display

Our application offers the ability to view the schema of the PostgreSQL database, providing an organized display of the various tables and their corresponding columns.

Instructions



INSTRUCTIONS FOR USE

1. Once the Query Q has been submitted, 'Submit Query' button will be DISABLED. Click on the Reset button to re-enable it.

2. Query Q' and 'Submit Updated Query' button will only be enabled when Query Q has been submitted.

3. Visual Plans will be available for viewing after submission of queries.

4. Difference between SQLs and QEPs will require both Query Q and Q' to be submitted.

5. Creative Functionality: Improved user experience with Text-To-Speech capabilities to read the difference explanations for you! Press the 'Play' button under any of the QEP textboxes to play the audio that reads out the explanations to you, and press the 'Stop' button to stop the audio at any time.

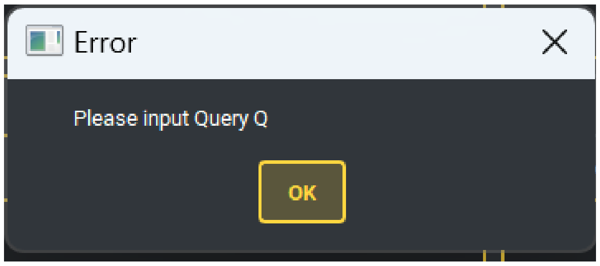
Figure 8: Instructions Display

Our application provides comprehensive instructions to guide users on the functionality of our application and its usage.

Error Checking

Our application ensures that the SQL Queries are valid before generating differences between SQL Queries and QEPs. The SQL queries given by the users are checked against 2 requirements which are as follows:

1) Users are not allowed to submit a blank query



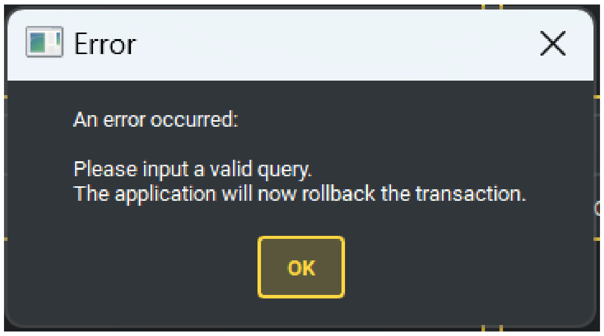
Please input Query Q

Figure 9: Error Pop Up for empty inputs

Empty strings submitted by the users in the inputs will be rejected. An error message in pop up will be shown if the user submits an empty string as the SQL query.

This prevents the users from sending an empty string to PostgreSQL which will raise an error and crash the application.

2) Users are only allowed to input valid SQL queries



An error occurred:

Please input a valid query.
The application will now rollback the transaction.

Figure 10: Error Pop Up for invalid SQL queries

The best method to ensure that the SQL queries are valid is to verify if the queries are accepted by PostgreSQL. The application will expect an input from the user and will then submit the query to PostgreSQL. If the input is an invalid query, PostgreSQL will raise an error and the application will display an error pop up message.

The pop up states that "The application will now rollback the transaction", as the application is required to rollback the transaction in PostgreSQL to enable the user to input a new SQL query and to make sure that the data remains accurate and consistent.

Reason behind Error Checking

Error checking functionalities that our application offers ensures that the system will not crash as the user navigates through the app. This ensures a user-friendly experience for the users.

Application Walkthrough

This section will demonstrate the recommended process of inputting queries, generating the differences in QEPs & SQLs explanations and their respective visual plans.

Step 1: Database Verification

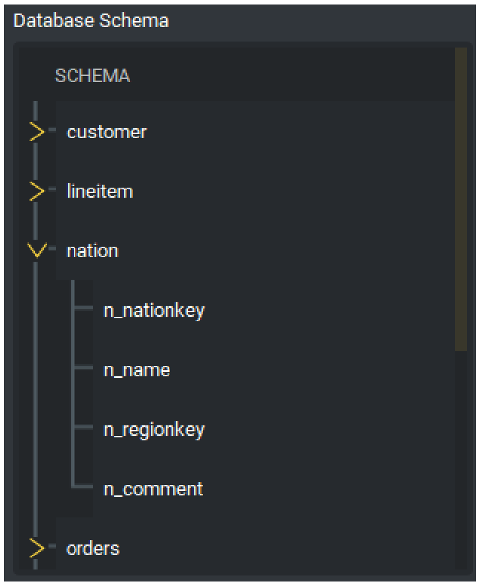
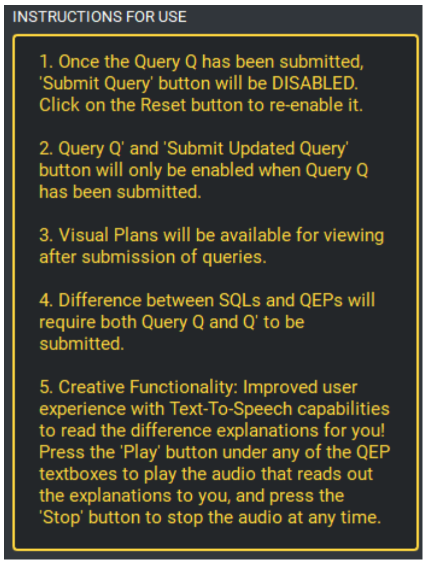


Figure 10: Verify Database Connection with Schema

Before proceeding, it is important to confirm that your connection to the intended database server, and that the required relations are stored in it. To do so, you can inspect the database schema.

The database schema shown is the TPCH dataset you should have imported into our server named "TPC-H". Ensure that the database is correctly loaded before proceeding to the next step.

Step 2: Familiarise yourself with the application's functionalities



INSTRUCTIONS FOR USE

1. Once the Query Q has been submitted, 'Submit Query' button will be DISABLED. Click on the Reset button to re-enable it.

2. Query Q' and 'Submit Updated Query' button will only be enabled when Query Q has been submitted.

3. Visual Plans will be available for viewing after submission of queries.

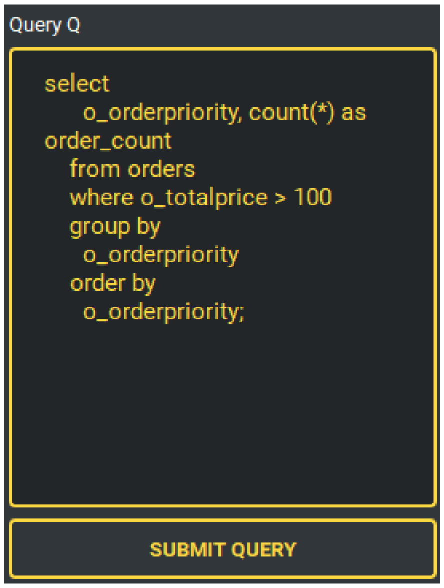
4. Difference between SQLs and QEPs will require both Query Q and Q' to be submitted.

5. Creative Functionality: Improved user experience with Text-To-Speech capabilities to read the difference explanations for you! Press the 'Play' button under any of the QEP textboxes to play the audio that reads out the explanations to you, and press the 'Stop' button to stop the audio at any time.

Figure 10: Application Instructions

We recommend going through the "Instructions for Use" to familiarise yourself with the application's functionalities for a smooth user experience. Feel free to refer back to it in the subsequent steps as well.

Step 3: Input Query Q

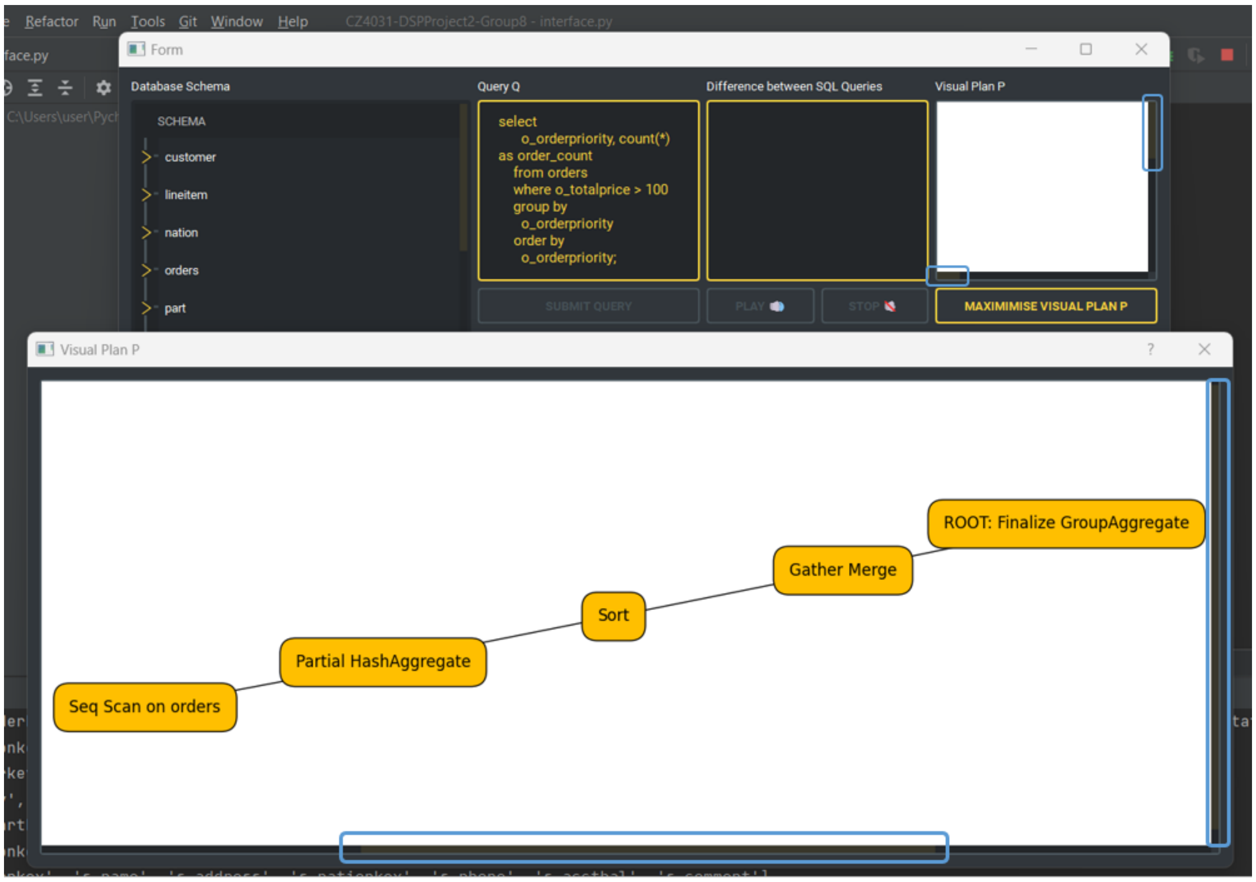


```
select
      o_orderpriority, count(*) as
order_count
    from orders
    where o_totalprice > 100
    group by
      o_orderpriority
    order by
      o_orderpriority;
```

Figure 12: Input Query Q

Input your chosen query to be analyzed into the QTextBrowser labeled “Query Q”, and click on the “Submit Query” button.

Step 4: Observe the Visual Plans of Query Q

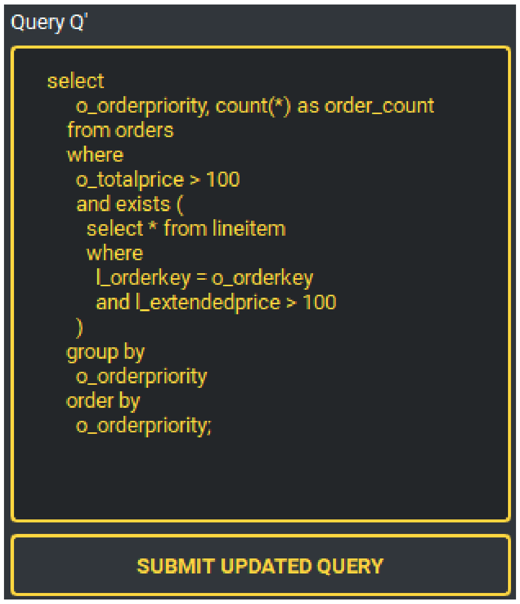


```
select
    o_orderpriority, count(*)
as order_count
  from orders
  where o_totalprice > 100
  group by
    o_orderpriority
  order by
    o_orderpriority;
```

Figure 13: Visual Plan of Query Q

After clicking on the "Submit Query" button, the visual plan for Query Q will be made available . We recommended clicking on the "Maximise Visual Plan P" button to view the visual plan in a bigger window for a clearer view. Use the scroll bars highlighted in blue rounded rectangles if needed.

Step 5: Input Query Q’



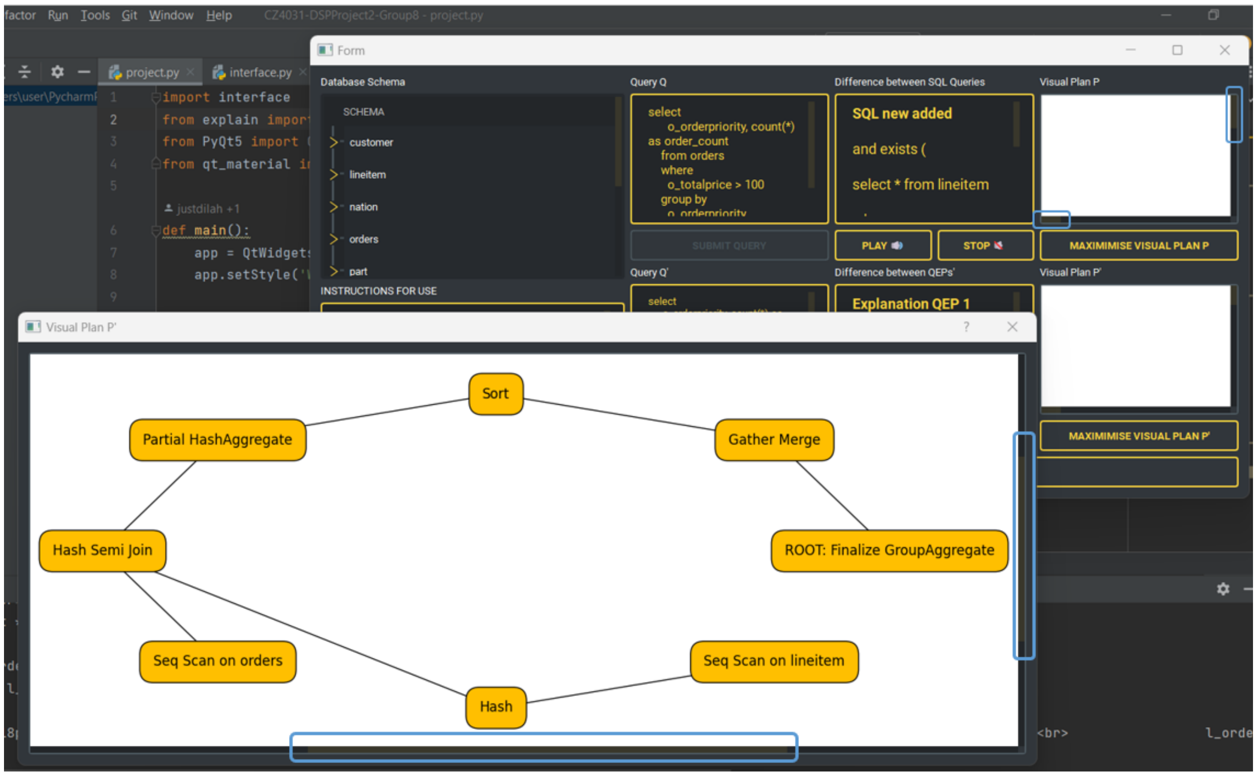
```
select
   o_orderpriority, count(*) as order_count
  from orders
  where
   o_totalprice > 100
   and exists (
    select * from lineitem
    where
     l_orderkey = o_orderkey
     and l_extendedprice > 100
   )
  group by
   o_orderpriority
  order by
   o_orderpriority;
```

Figure 14: Input Query Q’

Input your next chosen query to be analyzed into the QTextBrowser labeled “Query Q’”, and click on the “Submit Updated Query” button.

Do note it will take some time to generate the explanations for both the difference in SQL queries and QEP.

Step 6: Observe the Visual Plans of Query Q’



```
import interface
from explain import
from PyQt5 import
from qt_material i
justdilah +1
def main():
    app = QtWidget
    app.setStyle('
```

```
select
    o_orderpriority, count(*)
as order_count
  from orders
  where
    o_totalprice > 100
  group by
    o_orderpriority
```

SQL new added

and exists (

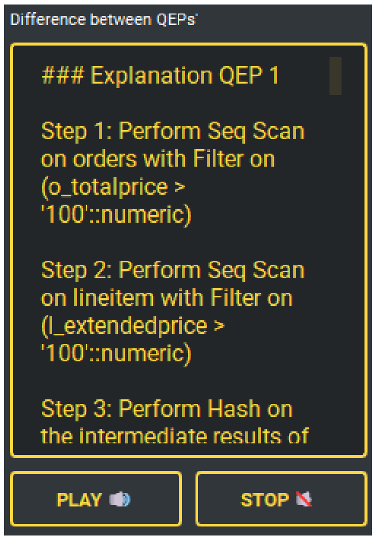
select * from lineitem

Explanation QEP 1

Figure 15: Visual Plan of Query Q’

After clicking on the “Submit Updated Query” button, the visual plan for Query Q’ will be made available. We recommended clicking on the “Maximise Visual Plan P’” button to view the visual plan in a bigger window for a clearer view. Use the scroll bars highlighted in blue rounded rectangles if needed.

Step 7: Observe the differences between the SQL Queries


Difference between QEPs'

Explanation QEP 1

Step 1: Perform Seq Scan on orders with Filter on (o_totalprice > '100'::numeric)

Step 2: Perform Seq Scan on lineitem with Filter on (l_extendedprice > '100'::numeric)

Step 3: Perform Hash on the intermediate results of

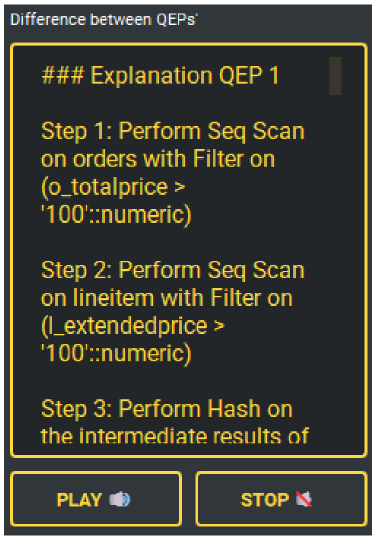
Figure 15: Differences in SQL Queries

After clicking on the “Submit Updated Query” button, it will also generate the “Difference between SQL Queries” explanation in its assigned QTextBrowser.

Comparing Query Q (Figure 12) and Query Q’ (Figure 14), the output difference is correct, as shown in Figure 15.

You can also make use of our application’s text-to-speech capabilities to play the audio to read out the text for your convenience.

Step 8: Observe the differences between the QEPs


Difference between QEPs'

Explanation QEP 1

Step 1: Perform Seq Scan on orders with Filter on (o_totalprice > '100'::numeric)

Step 2: Perform Seq Scan on lineitem with Filter on (l_extendedprice > '100'::numeric)

Step 3: Perform Hash on the intermediate results of

Figure 15: Differences in QEPs

After clicking on the “Submit Updated Query” button, it will also generate the “Difference between QEPs” explanation in its assigned QTextBrowser.

You can also make use of our application’s text-to-speech capabilities to play the audio to read out the text for your convenience.

Appendix A

Code for `QEP_Tree().build()`

```
    """
    Builds the QEP tree and returns the root node
    """

    def build(self, plan) -> QEP_Node:
        cur_node = None
        node = None
        indent_size = 0
        operation = ""
        details = None
        i = 0
        for row in plan:
            cur_row = row[0]

            # if this condition satisfies then this is the root node
            if "->" not in cur_row and "cost=" in cur_row:
                if self.root == None:
                    match = re.match(r"^(.+)\s\s(.+)$", cur_row)
                    node = QEP_Node(0, match.group(1).strip(), match.group(2))
                    self.root = node
                    self.root.parent = self.root
                    cur_node = self.root
                    self.prev_indent_size = 0

            # If the row has '->' in it, then it is considered an operation in the
QEP
            # Process the rows with '->' and extract relevant information such as
the
            # operation, depth, and details (explanation)
            # Indent size is used to determine if 2 nodes are on the same level
            if "->" in cur_row:
                match = re.match(r"(\s*->)?\s*(\w.*)\s+\((.*)\)$", cur_row)
                indent_size = len(match.group(1))
                operation = match.group(2).replace("Parallel", "")
                details = match.group(3)

                node = QEP_Node(indent_size, operation.strip(), details)

                if self.root == None:
                    self.root = node
                    self.root.parent = self.root
                    cur_node = self.root
                    self.prev_indent_size = indent_size
                    i += 1
                    continue

                # node is on the same level
                # get the parent node and attach it as its child
                if self.prev_indent_size == indent_size:
                    parent = cur_node.parent
                    node.parent = parent
                    parent.children.append(node)

                # further down the tree there are child nodes
                # so have to find its parent and
                # attach the child (this node) to it
                elif self.prev_indent_size > indent_size:
                    p = self.findParent(self.root, indent_size)
                    if p:
                        if len(p.children) < 2:
                            p.children.append(node)
                            node.parent = p
                else:
                    node.parent = cur_node
                    if len(cur_node.children) < 2:
                        cur_node.children.append(node)
            # Row is an explanation row (without '->')
            # Attach the explanations to the explanation list in the node
            else:
                if "Workers Planned" not in cur_row:
                    explain_results =
self.transformRawExplainToNaturalLanguage(cur_row)
                else:
                    explain_results = ""

                if explain_results == None or explain_results == "":
                    i += 1
                    continue

                node.explanation.append(explain_results)

            # Update the current node pointer
            self.prev_indent_size = indent_size
            cur_node = node
            i += 1

        return self.root
```

Appendix B

Code for `QEP_Tree().get_explanation()`

```
    """
    Traverses the QEP tree by specifying the root node of the particular tree
    recursively using post order traversal to get the steps and explanation
    for the tree
    Returns a list of explanations of each step
    """

    def get_explanation(self, node: QEP_Node, resultList: List[str]) -> List[str]:
        if node == None:
            return []

        for child in node.children:
            self.get_explanation(child, resultList)

        operation = node.operation
        explanation = (
            node.explanation
            if node.explanation != [] and node.explanation != None
            else ""
        )

        step = len(resultList) + 1
        if explanation == "" and (
            "join" not in operation.lower()
            and "hash" not in operation.lower()
            and "nested loop" not in operation.lower()
        ):
            result = f"Step {step}: Perform {operation}"
        elif "join" in operation.lower() or "nested loop" in operation.lower():
            result = f"Step {step}: Perform {operation} on the intermediate results
 of step {node.children[0].step} and {node.children[1].step}"
        elif "hash" in operation.lower() and "join" not in operation.lower():
            result = f"Step {step}: Perform {operation} on the intermediate results
 of step {node.children[0].step}"
        else:
            result = f"Step {step}: Perform {operation} with
 {self.joinExplanationList(explanation)}"

        node.step = step

        resultList.append(result)

        return resultList
```

Example Query Plan from postgreSQL

```
Finalize GroupAggregate
│
└── Gather Merge
    │
    └── Partial GroupAggregate
        │
        └── Sort
            │
            └── Hash Join
                │
                ├── Nested Loop
                │   │
                │   ├── Nested Loop
                │   │   │
                │   │   ├── Nested Loop
                │   │   │   │
                │   │   │   ├── Hash Join
                │   │   │   │   │
                │   │   │   │   ├── Seq Scan on orders
                │   │   │   │   └── Hash
                │   │   │   │       │
                │   │   │   │       ├── Hash Join
                │   │   │   │       │   │
                │   │   │   │       │   ├── Seq Scan on customer
                │   │   │   │       │   └── Hash
                │   │   │   │       │       │
                │   │   │   │       │       ├── Hash Join
                │   │   │   │       │       │   │
                │   │   │   │       │       │   ├── Seq Scan on nation n1
                │   │   │   │       │       │   └── Hash
                │   │   │   │       │       │       │
                │   │   │   │       │       │       ├── Seq Scan on region
                │   │   │   │       │       │       └── Filter
                │   │   │   │       │       └── Hash
                │   │   │   │       │           │
                │   │   │   │       │           └── Seq Scan on nation n1
                │   │   │   │       └── Filter
                │   │   │   └── Index Scan using lineitem_pkey on lineitem
                │   │
                │   └── Index Scan using part_pkey on part
                │
                └── Index Scan using supplier_pkey on supplier
                    │
                    └── Hash
                        │
                        └── Seq Scan on nation n2
```

Example Query

```
select
      o_year,
      sum(case
        when nation = 'BRAZIL' then volume
        else 0
      end) / sum(volume) as mkt_share
    from
      (
        select
          DATE_PART('YEAR',o_orderdate) as o_year,
          l_extendedprice * (1 - l_discount) as volume,
          n2.n_name as nation
        from
          part,
          supplier,
          lineitem,
          orders,
          customer,
          nation n1,
          nation n2,
          region
        where
          p_partkey = l_partkey
          and s_suppkey = l_suppkey
          and l_orderkey = o_orderkey
          and o_custkey = c_custkey
          and c_nationkey = n1.n_nationkey
          and n1.n_regionkey = r_regionkey
          and r_name = 'AMERICA'
          and s_nationkey = n2.n_nationkey
          and o_orderdate between '1995-01-01' and '1996-12-31'
          and p_type = 'ECONOMY ANODIZED STEEL'
          and s_acctbal > 10
          and l_extendedprice > 100
      ) as all_nations
    group by
      o_year
    order by
      o_year;
```

Example Result

Step	Explanation
Step 1	Perform Seq Scan on orders with Filter on ((o_orderdate >= '1995-01-01'::date) AND (o_orderdate <= '1996-12-31'::date))
Step 2	Perform Seq Scan on customer
Step 3	Perform Seq Scan on nation n1
Step 4	Perform Seq Scan on region with Filter on (r_name = 'AMERICA'::bpchar)
Step 5	Perform Hash on the intermediate results of step 4
Step 6	Perform Hash Join on the intermediate results of step 3 and 5
Step 7	Perform Hash on the intermediate results of step 6
Step 8	Perform Hash Join on the intermediate results of step 2 and 7
Step 9	Perform Hash on the intermediate results of step 8
Step 10	Perform Hash Join on the intermediate results of step 1 and 9
Step 11	Perform Index Scan using lineitem_pkey on lineitem with Index Cond on (l_orderkey = orders.o_orderkey) and Filter on (l_extendedprice > '100'::numeric)
Step 12	Perform Nested Loop on the intermediate results of step 10 and 11
Step 13	Perform Index Scan using part_pkey on part with Index Cond on (p_partkey = lineitem.l_partkey) and Filter on ((p_type)::text = 'ECONOMY ANODIZED STEEL'::text)
Step 14	Perform Nested Loop on the intermediate results of step 12 and 13
Step 15	Perform Index Scan using supplier_pkey on supplier with Index Cond on (s_suppkey = lineitem.l_suppkey) and Filter on (s_acctbal > '10'::numeric)
Step 16	Perform Nested Loop on the intermediate results of step 14 and 15

Step 17	Perform Seq Scan on nation n2
Step 18	Perform Hash on the intermediate results of step 17
Step 19	Perform Hash Join on the intermediate results of step 16 and 18
Step 20	Perform Sort with Sort Key on (date_part('YEAR'::text, (orders.o_orderdate)::timestamp without time zone))
Step 21	Perform Partial GroupAggregate with Group Key on (date_part('YEAR'::text, (orders.o_orderdate)::timestamp without time zone))
Step 22	Perform Gather Merge
Step 23	Perform Finalize GroupAggregate with Group Key on (date_part('YEAR'::text, (orders.o_orderdate)::timestamp without time zone))

Appendix C

Code for `QEP_Tree().compareQEP()`

```
"""
    Compares 2 QEP trees by taking each tree's root node as argument
    Traverses both trees to generate a queue of steps for each tree
    Uses these queues to compare the actions taken by each tree
    Returns a list of comparisons
    """

    def compareQEP(self, node1: QEP_Node, node2: QEP_Node) -> List[str]:
        def get_node_queue(node: QEP_Node, queue: List[QEP_Node]) -> List[QEP_Node]:
            if node == None:
                return

            for child in node.children:
                get_node_queue(child, queue)

            queue.append(node)

            return queue

        q1Queue = get_node_queue(node1, [])
        q2Queue = get_node_queue(node2, [])

        compareList = []
        q1Pointer = q2Pointer = 0
        while q1Pointer < len(q1Queue) and q2Pointer < len(q2Queue):
            operationQ1 = q1Queue[q1Pointer].operation
            explanationQ1 = q1Queue[q1Pointer].explanation
            operationQ2 = q2Queue[q2Pointer].operation
            explanationQ2 = q2Queue[q2Pointer].explanation
            step = min(q1Pointer + 1, q2Pointer + 1)
            if operationQ1 == operationQ2 and explanationQ1 == explanationQ2:
                if not explanationQ1:
                    compareExplanation = f"Step {step}: Both queries perform the same
operations at this step executing a {operationQ1}."
                else:
                    compareExplanation = f"Step {step}: Both queries perform the same
operations at this step executing a {operationQ1} with
{self.joinExplanationList(explanationQ1)}."
            elif operationQ1 == operationQ2 and explanationQ1 != explanationQ2:
                if explanationQ1 and explanationQ2:
                    compareExplanation = f"Step {step}: Both queries perform the same
operations at this step executing a {operationQ1}. However, Q1 and Q2 have different
conditions on the operations with Q1 to {self.joinExplanationList(explanationQ1)} and
Q2 to {self.joinExplanationList(explanationQ2)}."
                elif explanationQ1 and not explanationQ2:
                    compareExplanation = f"Step {step}: Both queries perform the same
operations at this step executing a {operationQ1}. However, Q2 has an additional
condition to {self.joinExplanationList(explanationQ2)} while Q1 is just performing a
basic {operationQ1}."
                else:
                    compareExplanation = f"Step {step}: Both queries perform the same
operations at this step executing a {operationQ1}. However, Q1 has an additional
condition to {self.joinExplanationList(explanationQ1)} while Q2 is just performing a
basic {operationQ2}."
            elif operationQ1 != operationQ2:
                if explanationQ1 and explanationQ2:
                    compareExplanation = f"Step {step}: Both queries are performing
different operations at this step, with Q1 executing a {operationQ1} with
{self.joinExplanationList(explanationQ1)} and Q2 executing a {operationQ2} with
{self.joinExplanationList(explanationQ2)}."
                elif explanationQ1 and not explanationQ2:
                    compareExplanation = f"Step {step}: Both queries are performing
different operations at this step, with Q1 executing a {operationQ1} with
{self.joinExplanationList(explanationQ1)} and Q2 executing a {operationQ2}."
                elif not explanationQ1 and explanationQ2:
                    compareExplanation = f"Step {step}: Both queries are performing
different operations at this step, with Q1 executing a {operationQ1} and Q2 executing a
{operationQ2} with {self.joinExplanationList(explanationQ2)}."
                else:
                    compareExplanation = f"Step {step}: Both queries are performing
different operations at this step, with Q1 executing a {operationQ1} and Q2 executing a
{operationQ2}."

            compareList.append(compareExplanation)
            compareExplanation = ""
            q1Pointer += 1
            q2Pointer += 1

        line = (
            "Q1 has additional steps taken due to the change in query.\nAdditional
steps taken by Q1:"
            if q1Pointer < len(q1Queue)
            else "Q2 has additional steps taken due to the change in query.\nAdditional
steps taken by Q2:"
        )
        compareList.append(line)

        while q1Pointer < len(q1Queue):
            operation = q1Queue[q1Pointer].operation
            explanation = q1Queue[q1Pointer].explanation
            step = q1Pointer + 1
            if explanation:
                compareExplanation = f"Step {step}: Q1 executes {operation} due to
{self.joinExplanationList(explanation)} which doesn't exist in Q2"
```

```
            else:
                compareExplanation = f"Step {step}: Q1 executes {operation}"
            compareList.append(compareExplanation)
            q1Pointer += 1

        while q2Pointer < len(q2Queue):
            operation = q2Queue[q2Pointer].operation
            explanation = q2Queue[q2Pointer].explanation
            step = q2Pointer + 1
            if explanation:
                compareExplanation = f"Step {step}: Q2 executes {operation} due to
{self.joinExplanationList(explanation)} which doesn't exist in Q1"
            else:
                compareExplanation = f"Step {step}: Q2 executes {operation}"
            compareList.append(compareExplanation)
            q2Pointer += 1

        return compareList
```

Example Query 1

```
select
      o_year,
      sum(case
        when nation = 'BRAZIL' then volume
        else 0
      end) / sum(volume) as mkt_share
    from
      (
        select
          DATE_PART('YEAR',o_orderdate) as o_year,
          l_extendedprice * (1 - l_discount) as volume,
          n2.n_name as nation
        from
          part,
          supplier,
          lineitem,
          orders,
          customer,
          nation n1,
          nation n2,
          region
        where
          p_partkey = l_partkey
          and s_suppkey = l_suppkey
          and l_orderkey = o_orderkey
          and o_custkey = c_custkey
          and c_nationkey = n1.n_nationkey
          and n1.n_regionkey = r_regionkey
          and r_name = 'AMERICA'
          and s_nationkey = n2.n_nationkey
          and o_orderdate between '1995-01-01' and '1996-12-31'
          and p_type = 'ECONOMY ANODIZED STEEL'
          and s_acctbal > 10
          and l_extendedprice > 100
      ) as all_nations
    group by
      o_year
    order by
      o_year;
```

Example Query 2

```
    select
      n_name,
      o_year,
      sum(amount) as sum_profit
    from
      (
        select
          n_name,
          DATE_PART('YEAR',o_orderdate) as o_year,
          l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as
amount
        from
          part,
          supplier,
          lineitem,
          partsupp,
          orders,
          nation
        where
          s_suppkey = l_suppkey
          and ps_suppkey = l_suppkey
          and ps_partkey = l_partkey
          and p_partkey = l_partkey
          and o_orderkey = l_orderkey
          and s_nationkey = n_nationkey
          and p_name like '%green%'
          and s_acctbal > 10
          and ps_supplycost > 100
      ) as profit
    group by
      n_name,
      o_year
    order by
      n_name,
      o_year desc;
```

Example Query Plan for Example Query 1 from postgreSQL

```
Finalize GroupAggregate
│
└─── Gather Merge
    │
    └─── Partial GroupAggregate
        │
        └─── Sort
            │
            └─── Hash Join
                │
                ├─── Nested Loop
                │   │
                │   ├─── Nested Loop
                │   │   │
                │   │   ├─── Nested Loop
                │   │   │   │
                │   │   │   ├─── Hash Join
                │   │   │   │   │
                │   │   │   │   ├─── Seq Scan on orders
                │   │   │   │   └─── Hash
                │   │   │   │       │
                │   │   │   │       ├─── Hash Join
                │   │   │   │       │   │
                │   │   │   │       │   ├─── Seq Scan on customer
                │   │   │   │       │   └─── Hash
                │   │   │   │       │       │
                │   │   │   │       │       ├─── Hash Join
                │   │   │   │       │       │   │
                │   │   │   │       │       │   ├─── Seq Scan on nation n1
                │   │   │   │       │       │   └─── Hash
                │   │   │   │       │       │       │
                │   │   │   │       │       │       ├─── Seq Scan on region
                │   │   │   │       │       │       └─── Filter
                │   │   │   │       │       └─── Hash
                │   │   │   │       │           │
                │   │   │   │       │           └─── Seq Scan on nation n1
                │   │   │   │       └─── Filter
                │   │   │   └─── Index Scan using lineitem_pkey on lineitem
                │   │
                │   └─── Index Scan using part_pkey on part
                │
                └─── Index Scan using supplier_pkey on supplier
                    │
                    └─── Hash
                        │
                        └─── Seq Scan on nation n2
```

Example Query Plan for Example Query 2 from postgreSQL

```
Finalize GroupAggregate
│
└── Gather Merge
    │
    └── Partial GroupAggregate
        │
        └── Sort
            │
            └── Hash
                │
                └── Nested Loop
                    │
                    ├── Hash Join
                    │   │
                    │   ├── Seq Scan on partsupp
                    │   │
                    │   └── Hash
                    │       │
                    │       └── Hash Join
                    │           │
                    │           ├── Seq Scan on lineitem
                    │           │
                    │           └── Hash
                    │               │
                    │               └── Seq Scan on part
                    │
                    └── Index Scan on supplier using supplier_pkey
```

Example Result for Comparison:

Step	Explanation
1	Both queries perform different operations. Q1 executes a Seq Scan on orders with Filter on ((o_orderdate >= '1995-01-01'::date) AND (o_orderdate <= '1996-12-31'::date)) and Q2 executes a Seq Scan on partsupp with Filter on (ps_supplycost > '100'::numeric).
2	Both queries perform different operations. Q1 executes a Seq Scan on customer and Q2 executes a Seq Scan on lineitem.
3	Both queries perform different operations. Q1 executes a Seq Scan on nation n1 and Q2 executes a Seq Scan on part with Filter on ((p_name)::text ~~ '%green%'::text).
4	Both queries perform different operations. Q1 executes a Seq Scan on region with Filter on (r_name = 'AMERICA'::bpchar) and Q2 executes a Hash.
5	Both queries perform different operations. Q1 executes a Hash and Q2 executes a Hash Join with Hash Cond on (lineitem.l_partkey = part.p_partkey).
6	Both queries perform different operations. Q1 executes a Hash Join with Hash Cond on (n1.n_regionkey = region.r_regionkey) and Q2 executes a Hash.
7	Both queries perform different operations. Q1 executes a Hash and Q2 executes a Hash Join with Hash Cond on ((partsupp.ps_suppkey = lineitem.l_suppkey) AND (partsupp.ps_partkey = lineitem.l_partkey)).
8	Both queries perform different operations. Q1 executes a Hash Join with Hash Cond on (customer.c_nationkey = n1.n_nationkey) and Q2 executes a Index Scan using supplier_pkey on supplier with Index Cond on (s_suppkey = lineitem.l_suppkey) and Filter on (s_acctbal > '10'::numeric).
9	Both queries perform different operations. Q1 executes a Hash and Q2 executes a Nested Loop.
10	Both queries perform different operations. Q1 executes a Hash Join with Hash Cond on (orders.o_custkey = customer.c_custkey) and Q2 executes a Index Scan using orders_pkey on orders with Index Cond on (o_orderkey = lineitem.l_orderkey).

11	Both queries perform different operations. Q1 executes a Index Scan using lineitem_pkey on lineitem with Index Cond on (l_orderkey = orders.o_orderkey) and Filter on (l_extendedprice > '100'::numeric) and Q2 executes a Nested Loop.
12	Both queries perform different operations. Q1 executes a Nested Loop and Q2 executes a Seq Scan on nation.
13	Both queries perform different operations. Q1 executes a Index Scan using part_pkey on part with Index Cond on (p_partkey = lineitem.l_partkey) and Filter on ((p_type)::text = 'ECONOMY ANODIZED STEEL'::text) and Q2 executes a Hash.
	ADDITIONAL STEPS TAKEN BY Q1:
14	Both queries perform different operations. Q1 executes a Nested Loop and Q2 executes a Hash Join with Hash Cond on (supplier.s_nationkey = nation.n_nationkey).
15	Q1: Index Scan using supplier_pkey on supplier with Index Cond on (s_suppkey = lineitem.l_suppkey) and Filter on (s_acctbal > '10'::numeric) Q2: Sort with Sort Key on nation.n_name
16	Q1: Nested Loop Q2: Partial GroupAggregate with Group Key on nation.n_name
17	Q1: Seq Scan on nation n2 Q2: Gather Merge
18	Q1: Hash Q2: Finalize GroupAggregate with Group Key on nation.n_name
19	Q1: Hash Join due to Hash Cond on (supplier.s_nationkey = n2.n_nationkey) which doesn't exist in Q2
20	Q1: Sort due to Sort Key on (date_part('YEAR'::text, (orders.o_orderdate)::timestamp without time zone)) which doesn't exist in Q2
21	Q1: Partial GroupAggregate due to Group Key on (date_part('YEAR'::text, (orders.o_orderdate)::timestamp without time zone)) which doesn't exist in Q2
22	Q1: Gather Merge
23	Q1: Finalize GroupAggregate due to Group Key on (date_part('YEAR'::text, (orders.o_orderdate)::timestamp without time zone)) which doesn't exist in Q2

Appendix D

Code for `compare_sql()`

```
def compare_sql(self, string1, string2):
    lines1 = Explain.stripString(string1)
    lines2 = Explain.stripString(string2)

    # Compare the lines using the Differ class
    differ = difflib.Differ()
    diff = list(differ.compare(lines1, lines2))

    # Store the differences in a list
    differences = []
    temp = []
    sign = ""
    pre = ""
    for line in diff:
        text = line[2:]
        if (sign.startswith("-") and line.startswith("+")) or (
            sign.startswith("+") and line.startswith("-")
        ):
            differences.append(temp)
            temp = []
        if len(temp) == 0 and line.startswith("-"):
            temp.append(("#### SQL old removed #### \n", text))
            sign = "-"
        elif len(temp) == 0 and line.startswith("+"):
            temp.append(("#### SQL new removed #### \n", text))
            sign = "+"
        elif line.startswith("+") and pre.startswith("+"):
            temp.append(("", text)) # SQL 2
        elif line.startswith("-") and pre.startswith("-"):
            temp.append(("", text)) # SQL 1
        elif line.startswith("?") :
            continue

        pre = line

    if len(temp) != 0:
        differences.append(temp)

    if differences[-1][0][0] == "SQL old removed\n":
        temp = []
        temp.append(("SQL new added\n", " "))
        differences.append(temp)
    return differences
```

Code for `explainSQL()`

```
def explainSQL(self,diffArray):
    line1 = ""
    line2 = ""
    pretype = ""
    change = []
    explanation = []

    for i in diffArray:
        for diff_type, line in i:
            if diff_type == "#### SQL new removed #### \n" or diff_type == "####
SQL old removed #### \n":
                pretype = diff_type

            if pretype == "#### SQL old removed #### \n":
                if line1 and line2:
                    string1_words = re.split(r"[ _-]+", line1)
                    string2_words = re.split(r"[ _-]+", line2)

                    if len(string1_words) == len(string2_words):
                        for i in range(min(len(string1_words),
len(string2_words))):
                            if string1_words[i] != string2_words[i]:
                                change.append([string1_words[i], string2_words[i]])
                    else :
                        change.append([' '.join(string1_words),'
'.join(string2_words)])

                    line1 = ""
                    line2 = ""

            if line1 == "" or pretype == "[[[ SQL old removed #### \n":
                line1 += line
                continue
            elif line2 == "" or pretype == "#### SQL old removed #### \n":
                line2 += line
                continue

    if line1 and line2:
        string1_words = re.split(r"[ _-]+", line1)
        string2_words = re.split(r"[ _-]+", line2)

        if len(string1_words) == len(string2_words):
            for i in range(min(len(string1_words), len(string2_words))):
                if string1_words[i] != string2_words[i]:
                       change.append([string1_words[i], string2_words[i]])
        else :
```

```
        change.append([' '.join(string1_words),' '.join(string2_words)])


for i in change:
    If i[1] == " ":
        explanation.append("The old query is removed")
    elif i[0].isnumeric() and i[1].isnumeric():
        explanation.append("There is a change in value")
    elif i[0].isnumeric() and (i[1].isalnum() or i[1].isalpha()):
        explanation.append("There is a change from a value to a variable")
    elif i[1].isnumeric() and (i[0].isalnum() or i[0].isalpha()):
        explanation.append("There is a change from a variable to a value")
    elif (i[0].isalnum() or i[0].isalpha()) and (
        i[1].isalnum() or i[1].isalpha()
    ):
        explanation.append("There is a change in variable")
    else:
        explanation.append("The query here has been changed")

return explanation
```

Appendix E

Short Query: Example Query Q, Query Q' and difference between SQL Queries output

Query Q

```
Select sum(l_extendedprice * l_discount) as revenue from lineitem
    where
         l_extendedprice > 100;
```

Query Q'

```
Select sum(l_extendedprice * l_discount) as revenue from lineitem
    where
         l_extendedprice > 100 AND l_extendedprice < 1200;
```

Difference between SQL Queries

```
#### SQL old removed ####
where l_extendedprice > 100;

#### SQL new added ####
where l_extendedprice > 100 and l_extendedprice < 1200

Explanation:
The query here has been changed
===================
```

Medium Query: Example Query Q, Query Q' and difference between SQL Queries output

Query Q

```
Select
    o_orderpriority, count(*) as order_count from orders
    where
      o_totalprice > 100
    group by
      o_orderpriority
    order by
      o_orderpriority;
```

Query Q'

```
Select
    o_orderpriority, count(*) as order_count from orders
    where
      o_totalprice > 100
      and exists (select * from lineitem
        where
          l_orderkey = o_orderkey and l_extendedprice > 100
      )
    group by
      o_orderpriority
    order by
      o_orderpriority;
```

Difference between SQL Queries

```
#### SQL old removed ####
orderpriority, count(*) as order_count from orders where o_totalprice > 100

#### SQL new added ####
orderpriority, count(*) as order_count from orders where o_totalprice > 100 and
exists ( select * from lineitem where l_orderkey = o_orderkey and l_extendedprice >
100 )

Explanation:
The query here has been changed
===================
```

Large Query: Example Query Q, Query Q’ and difference between SQL Queries output

Query Q

```
Select supp_nation, cust_nation, l_year, sum(volume) as revenue from
      (
        select n1.n_name as supp_nation, n2.n_name as cust_nation,
DATE_PART('YEAR',l_shipdate) as l_year, l_extendedprice * (1 - l_discount) as
volume
      from supplier, lineitem, orders, customer, nation n1, nation n2
      where
        s_suppkey = l_suppkey and o_orderkey = l_orderkey and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey and c_nationkey = n2.n_nationkey) as shipping
    group by supp_nation, cust_nation, l_year;
```

Query Q’

```
Select supp_nation, cust_nation, l_year, sum(volume) as revenue from
      (
        select n1.n_name as supp_nation, n2.n_name as cust_nation,
DATE_PART('YEAR',l_shipdate) as l_year, l_extendedprice * (1 - l_discount) as
volume
        from supplier, lineitem, orders, customer, nation n1, nation n2
        where
          s_suppkey = l_suppkey and o_orderkey = l_orderkey and c_custkey =
o_custkey and s_nationkey = n1.n_nationkey and c_nationkey = n2.n_nationkey
          and (
            (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
            or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE')
          )
          and l_shipdate between '1995-01-01' and '1996-12-31'
          and o_totalprice > 100 and c_acctbal > 10
      ) as shipping
    group by supp_nation, cust_nation, l_year
    order by supp_nation, cust_nation, l_year;
```

Difference between SQL Queries

```
#### SQL old removed ####
orders, customer, nation n1, nation n2 where s_suppkey = l_suppkey and o_orderkey =
l_orderkey and c_custkey = o_custkey and s_nationkey = n1.n_nationkey and
c_nationkey = n2.n_nationkey) as shipping

#### SQL new added ####
orders, customer, nation n1, nation n2 where s_suppkey = l_suppkey and o_orderkey =
l_orderkey and c_custkey = o_custkey and s_nationkey = n1.n_nationkey and
c_nationkey = n2.n_nationkey and ( (n1.n_name = 'france' and n2.n_name = 'germany')
or (n1.n_name = 'germany' and n2.n_name = 'france') ) and l_shipdate between
'1995-01-01' and '1996-12-31' and o_totalprice > 100 and c_acctbal > 10 ) as
shipping

Explanation :
The query here has been changed
 ===================

#### SQL old removed ####
group by supp_nation, cust_nation, l_year;

#### SQL new added ####
group by supp_nation, cust_nation, l_year
order by supp_nation, cust_nation, l_year;

Explanation :
There is a change in variable
===================
```