NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

CE4031/CZ4031 DATABASE SYSTEM PRINCIPLES

PROJECT 2: NATURAL LANGUAGE DESCRIPTION OF QUERY PLANS DURING DATA EXPLORATION

Group 2

Group Members	**Matriculation Number**
ANDY NG CHIN KUAN	U1621910C
LIM ZHI EN	U1622411D
MUHAMMAD AMIRUL AFIQ BIN JA'AFAR	U1620684A
THA TOE OO @JODI	U1621448G

Content Page

Introduction

This assignment is part of the CZ/CE4031 Database System Principles course from Nanyang Technological University (NTU). This report will provide a detailed description of our implementation of this second project assignment for this course. There are two parts to this assignment. Our first task is to write a program that automatically generates natural language description of the changes to the query plans that take place during data exploration. It is noted that the queries are related as they have evolved from the original query Q1. The generated query plans may also share common content among themselves. As an example, we are tasked to generate natural language description of the way the plans have evolved during data exploration (e.g., a hash join in P1 has now evolved to sort-merge join in P2 due to changes in the WHERE clause in Q2).

Our second task is to create a user-friendly, graphical user interface (GUI) to enable the aforementioned task. This project is developed in Python and uses PostgreSQL as its database and the TPC-H dataset as mentioned in the lab manual.

File Tree Structure

```
CE/CZ4032:.
├───sample_query (The directory contains some sample query found from the Internet)
├───School Sample Query (The directory contains some sample query that were used for
Neuron)
├───sql_commands (The directory contains sql commands that were used to import data to
Postgresql database.)
├───src (The directory contains the source code)
│   │   main.py (The file that needs to be run in command line.)
│   ├───postgres_interface (The directory contains code related to interacting with postgres)
│   │   │   postgres_wrapper.py  (The file that is used to interact with postgres)
│   │   │   __init__.py
│   ├───qt_parser (The directory contains code related to parsing the query tree)
│   │   │   find_difference.py  (The backend code to find the different between two query trees)
│   │   │   main_parser.py (Code that will be used to connect the backend with the frontend GUI)
│   │   │   node_utils.py (Code that contains the Node object and all the other codes related to
Node Object)
│   │   │   __init__.py
│   └───utils
│       │   singleton.py (Code that contains a singleton class that will be used to create singleton
classes)
│       │   __init__.py
└───tests (The directory contains test cases)
        context.py
        test_difference_in_natural_language.py
        __init__.py
```

Installation & Execution

Instructions

Run the following command to install the required libraries needed to run our program.

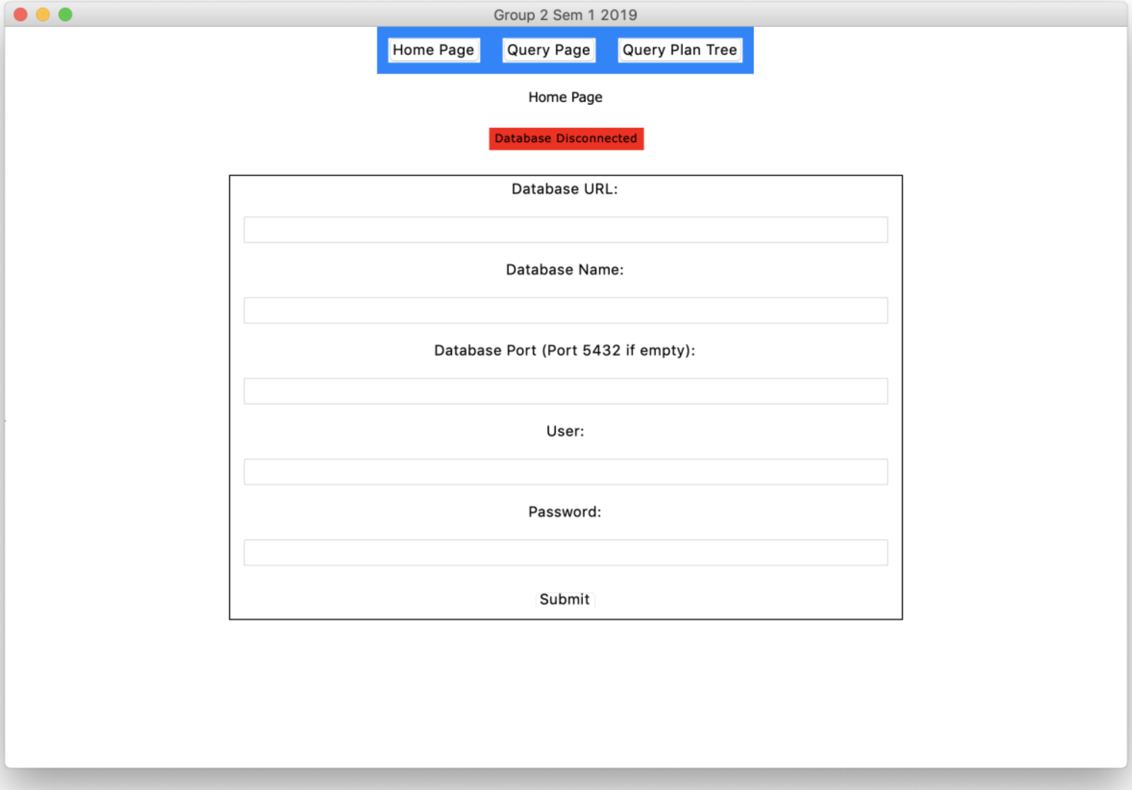
```
python -m pip install -r requirements.txt
```

Please navigate to the src directory and run the following command:

```
python main.py
```

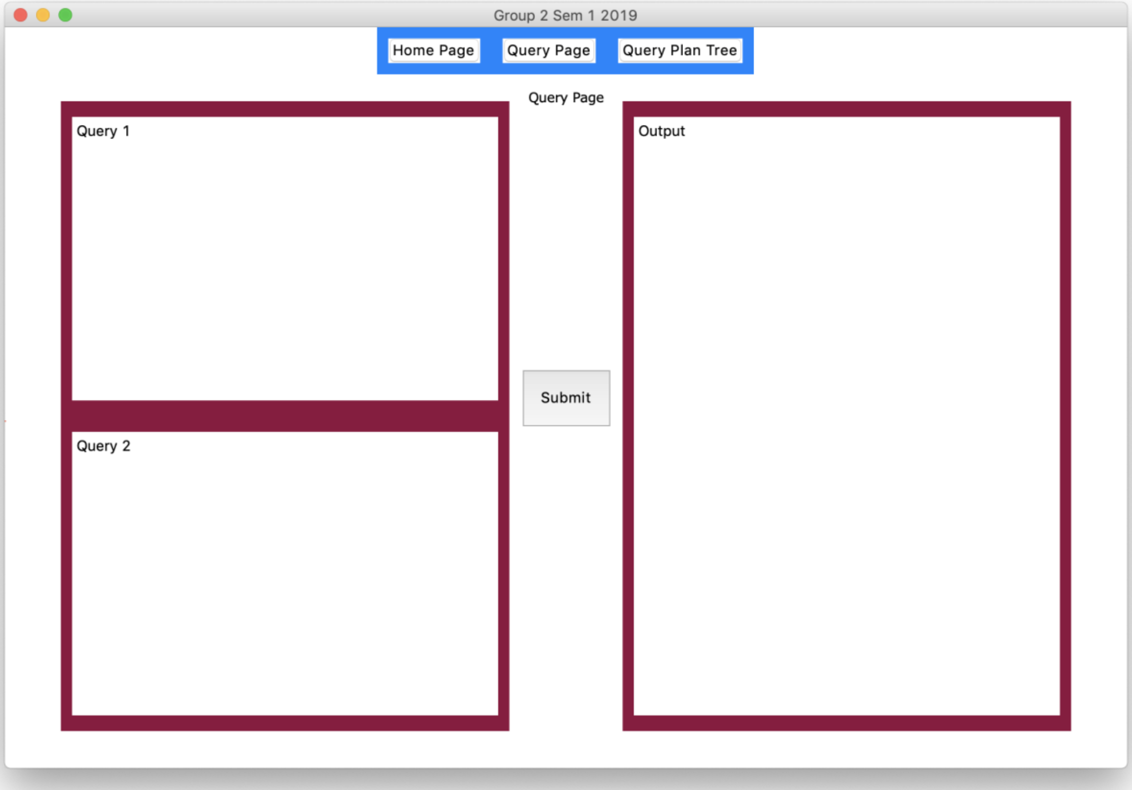
Home Page

Execute the main.py file to start up our program. You will be greeted with the following screen requesting for your details to the database that you'll be using. If you are unable to see the submit button on your screen, resize the application larger so that the button appears.



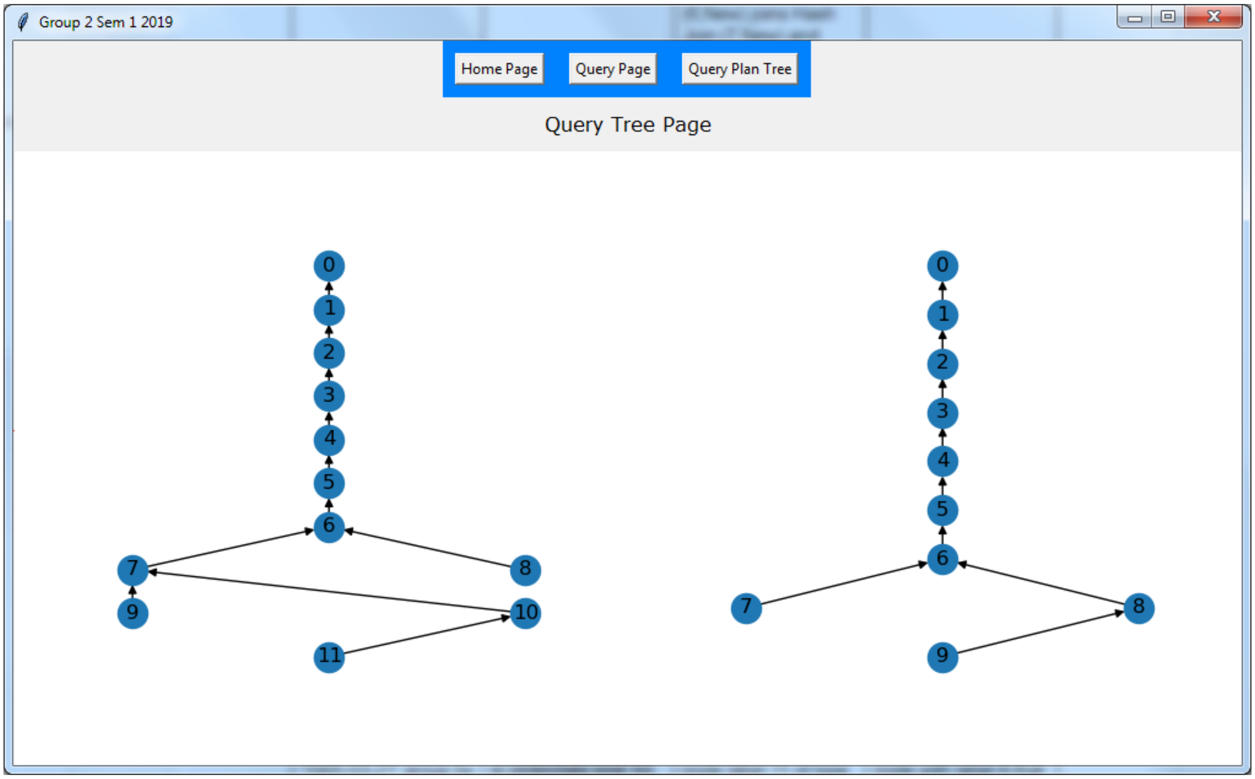
You will be informed that you have successfully logged in to the database when the red textbox changes from "Database Disconnected" to a green textbox with "Database Connected".

Query Page



In the Query page, you will be greeted by 2 textboxes on the left and an Output box on the right. Text Box with query 1 will represent the old query and Text Box with query 2 will represent the new query to be input by the user. The output box will display the differences in the analyzed text query inputs on the left into the output box on the right. If invalid queries are inputted into the query textboxes, the output will request for a valid SQL query to be inputted.

Query Plan Tree Page



The Query Plan Tree page will display the graph output of the two queries. The left graph will display the tree graph of the old query and the right for the new query. These output graphs are based on the inputs from the Query Page. Nodes on the tree graph are labelled by numbers and can be identified from the Query Page.

Algorithm to find differences between two sql queries

Differences in Projections

One of the simplest ways that a query can change is for the projections to change. In order to find the difference in projections between two queries, parsing the query plan tree alone won't be enough. This is because Postgresql does not include the projections when returning the output of the query plan.

As a result, in order to find the difference between the projections of two SQL queries, regex is used. The following regex pattern will be used: "select(.*?)from" with case ignored. From the regex pattern, we can obtain a string that consists of the projections. In order to separate out the projections properly, the result can be separated by commas and converted to a list. For each element in the list, we will be checking for any unbalanced brackets and removing the extra brackets. This is necessary because for queries such as "select (c_nationkey, c_name) from customer;" the resulting list will be ["(c_nationkey", "c_name)"] so it is necessary to remove the unbalanced brackets so that the resulting list can be converted to ["c_nationkey", "c_name"].

An example of the two queries that we used to test the differences in projections is the following:

Query 1: select * from customer;
Query 2: select (c_nationkey, c_name) from customer;
Output: Query projections has changed from ['*'] in the old query to ['c_name', 'c_nationkey'] in the new query.

This shows that in the case above, the differences in projections have been identified correctly.

Differences in query plan trees

To find the differences in the query plan trees, we will try to find what is called the minimum edit distance. The main idea of using the minimum edit distance is to find the smallest change that we can make to one graph to turn it into another graph. In our case, we will be finding a way to turn the query tree produced by the first query into the query tree produced by the second query.

We will be making use of a library called "NetworkX" which is a python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. We will first convert our query plan trees into a directed network x graph. Each node in the networkx graph will be labeled with a number, and each node will also possess a node object. The node object will possess the following attributes:

Attribute name	Representation Of
node_type	Node Type of the node in the query plan tree
relation_name	Relation Name of the node in the query plan tree
schema	Schema of the node in the query plan tree
alias	Alias of the node in the query plan tree
group_key	Group key of the node in the query plan tree
sort_key	Sort key of the node in the query plan tree
join_type	Join type of the node in the query plan tree
index_name	Index name of the node in the query plan tree
hash_cond	Hash Condition of the node in the query plan tree
table_filter	Table Filter of the node in the query plan tree
index_cond	Index Condition of the node in the query plan tree
merge_cond	Merge Condition of the node in the query plan tree
recheck_cond	Recheck Condition of the node in the query plan tree
join_filter	Join filter of the node in the query plan tree
subplan_name	Subplan name of the node in the query plan tree
plan_rows	Plan rows of the node in the query plan tree
output_name	Output name that will be used to prettify the output of the node in the query plan tree

After converting them into a network x graph, we will make use of the function in networkx called the “optimize_edit_paths”. The function will accept the two graphs and will return the node edit path: the nodes mappings from the first graph to the second graph, the edge edit path: the edge mappings from the first graph to the other, and also the cost to convert from one graph to the other.

To improve the performance of the function, we provided a few heuristics for both the nodes and the edges.

For node heuristics, we did the following:

If we identify the node in the old graph to be exactly the same as the node in the new graph: the cost of swapping the nodes from the old graph to the new graph will be 0.

If we identify the nodes to be the same node type and that the critical attributes are the same, the cost of swapping the nodes will be 0.25. To identify the critical attributes, we group the node types into different families. Different families and their critical attributes are shown below:

Nodes that belong to the same family	Critical attribute
Nodes that has the word ‘Scan’ in their node type	Relation name represented as relation_name in the node object
Nodes that has the word ‘Aggregate’ in their node type	Group key represented as group_key in the node object
Nodes that has the word ‘Sort’ in their node type	Sort key represented as sort_key in the node object
Nodes that has the word ‘Join’ or ‘Nested Loop’ in their node type	Hash condition for hash join, represented as hash_cond in node object Merge condition for merge join, represented as merge_cond in node object

If the nodes have the same node type, the cost of swapping the nodes from the old graph to the new graph will be 0.5. If the nodes belong to the same family, the cost of swapping the nodes from the old graph to the new graph will be 1.0. Otherwise, if the nodes are of different families, the cost of swapping the nodes will be 9223372036854775807, which is a very high cost to deter the nodes from different families from being swapped together.

For the edge heuristics, we did the following:

If the parent nodes and the child nodes of the edge in the old graph are mapped to an edge with a parent node of the same type and children node of the same type, the cost of swapping the edges will be 0. Otherwise, the cost of swapping the edges will be 1.

With the help of the heuristics, we can use the output from “optimize_edit_paths” to find the changes between the query tree plans. Specifically, we just have to use the node edit path to identify the changes. The output of the node edit path are tuples that can be categorized into different categories: swapped nodes, inserted nodes, and deleted nodes. Swapped nodes are nodes that have been swapped from the old graph to the new graph. All of the swapped nodes are in the following format (Old node label, New node label), where we can see how the node with the old node label in the old graph gets mapped to the node with the new node label in the new graph. All of the inserted nodes are in the following format (None, New node label), where we can see how none of the nodes in the old graph gets mapped to the node with the new node label in the new graph. All of the inserted nodes are in the following format (Old node label, None), where we can see how node with the old node label in the old graph cannot be mapped to any node in the new graph. After we have categorized the node edit path into different types of nodes, we can proceed to identify how the query plan tree has changed.

Swapped nodes

For swapped nodes, we have to iterate through all the tuples related to the swapped nodes and compare them to see the difference. If the node label has changed, we need to identify the new node label in the new graph. If any attributes of the node object have changed, we need to identify how the attributes have changed. The code snippet that we use to compare differences is shown below:

```
def compare_differences(self, other, original_label, current_label):
        """
        This function is used to compare the differences between two
nodes
        """
        differences = []
        if not(self.node_type == other.node_type or ("Scan" in
self.node_type and "Scan" in other.node_type) or \
            ("Aggregate" in self.node_type and "Aggregate" in
other.node_type) or ((self.node_type == "Nested Loop" or "Join" in
self.node_type)\
                and (other.node_type == "Nested Loop" or "Join" in
other.node_type))):
            difference = "The node with node label " +
str(original_label) + " of type " + str(self.node_type) + " has
evolved into " + str(current_label) + " of type " +
str(other.node_type)
            differences.append(difference)
        else:
            if (original_label != current_label or self.node_type !=
other.node_type):
                difference = "The node with node label " +
str(original_label) + " of type " + str(self.node_type) + " gets
mapped to new node label " + str(current_label) + " of type " +
str(other.node_type)
                differences.append(difference)
            if self.relation_name != other.relation_name:
                difference =  "relation name " +
str(self.relation_name) + " has changed into " +  "relation name " +
str(other.relation_name)
```

```
                differences.append(difference)
            if self.schema != other.schema:
                difference = "schema has changed from " +
str(self.schema) + " into " + str(other.schema)
                differences.append(difference)
            if self.alias != other.alias:
                difference = "alias has changed from " +
str(self.alias) + " into " + str(other.alias)
                differences.append(difference)
            if self.group_key != other.group_key:
                difference = "group key has changed from " +
str(self.group_key) + " into " + str(other.group_key)
                differences.append(difference)
            if self.sort_key != other.sort_key:
                difference = "sort key has changed from " +
str(self.sort_key) + " into " + str(other.sort_key)
                differences.append(difference)
            if self.join_type != other.join_type:
                difference = "join type has changed from " +
str(self.join_type) + " into " + str(other.join_type)
                differences.append(difference)
            if self.index_name != other.index_name:
                difference = "index name has changed from " +
str(self.index_name) + " into " + str(other.index_name)
                differences.append(difference)
            if self.hash_cond != other.hash_cond:
                difference = "hash condition has changed from " +
str(self.hash_cond) + " into " + str(other.hash_cond)
                differences.append(difference)
            if self.table_filter != other.table_filter:
                difference =  "table filter has changed from " +
str(self.table_filter) + " into " + str(other.table_filter)
                differences.append(difference)
            if self.index_cond != other.index_cond:
                difference =  "index condition has changed from " +
str(self.index_cond) + " into " + str(other.index_cond)
                differences.append(difference)
            if self.merge_cond != other.merge_cond:
                difference =  "merge condition has changed from " +
str(self.merge_cond) + " into " + str(other.merge_cond)
                differences.append(difference)
            if self.recheck_cond != other.recheck_cond:
                difference =  "recheck condition has changed from " +
str(self.recheck_cond) + " into " + str(other.recheck_cond)
                differences.append(difference)
            if self.join_filter != other.join_filter:
                difference =  "join filter has changed from " +
str(self.join_filter) + " into " + str(other.join_filter)
                differences.append(difference)
            if self.subplan_name != other.subplan_name:
                difference =  "subplan name has changed from " +
str(self.subplan_name) + " into " + str(other.subplan_name)
                differences.append(difference)
            if self.output_name != other.output_name:
                difference =  "output name has changed from " +
str(self.output_name) + " into " + str(other.output_name)
                differences.append(difference)
        if len(differences) == 0:
            return "N.A."
        if len(differences) == 1:
            difference = differences[0]
            if (original_label == current_label and self.node_type ==
other.node_type):
                return "The node with node label " +
str(original_label) + " of type " + str(self.node_type) + " has the
following changes: " \
                    + difference[0] +  difference [1:] + ".\n"
            else:
                return difference[0].upper() +  difference [1:] +
".\n"
        else:
            last_difference = differences[-1]
            differences_up_to_last = differences[:-1]
            difference_string = ", ".join(differences_up_to_last)
            difference_string += " and " + last_difference + ".\n"
            if (original_label == current_label and self.node_type ==
other.node_type):
                return "The node with node label " +
str(original_label) + " of type " + str(self.node_type) + " has the
following changes: " \
                    + difference_string[0] +  difference_string[1:]
            else:
                return difference_string[0].upper() +
difference_string[1:]
```

We first identified if the nodes belong to the same family. If they belong to the same family, we proceed to find out if the node in the new graph still has the same label as the node in the old graph. Afterward, we proceed to find out the difference of all the attributes in the node objects between the node in the old graph and the node in the new graph. Any differences between the two nodes will then be identified and returned.

Inserted nodes

To identify where the nodes are inserted, the following algorithm is used:
Perform a post-order traversal with all the nodes in the new graph so that node traversal will start from the leaves towards the roots to represent the order in which the nodes will run. If the current node is a swapped node, proceed to ignore it. Else if the current node belongs to the family used to join different relations together, proceed to find out the children of the node, which corresponds to the relations that will be joined together. Else add it to the array that will be used to store the current chain of inserted nodes. If the parent of the current node is a swapped node or belongs to the join type or if the current node is the root, then proceed to find out the children of the first element in the array. With the parent and the children, we will be able to find out the nodes between which the current chain of inserted nodes will be inserted. We will then proceed with the traversal until there are no more nodes to traverse. The code snippet that we used to identify the inserted nodes is shown below:

```
def get_natural_language_output_for_the_inserted_nodes(G2,
inserted_nodes, substitued_nodes, join_nodes):
"""This function is used to get the natural language output for any
inserted nodes"""
    difference_list = []
    inserted_nodes_list = []
    inserted_nodes_in_G2 = [x[1] for x in inserted_nodes]
    differences = []
    substitued_nodes_in_G2 = [x[1] for x in substitued_nodes]
    for node in list(nx.dfs_postorder_nodes(G2, source=0)):
        if node in substitued_nodes_in_G2:
            continue
        if node in join_nodes:

differences.append(get_natural_language_ouput_for_join_queries(G2,
node, new_flag))
        else:
            inserted_nodes_list.append(node)
            successors_list =
list(G2.successors(inserted_nodes_list[0]))
            if len(successors_list) == 0:
                successor = None
            else:
                successor = successors_list[0]
            if node == 0:
                parent = None
```

```
differences.append(get_natural_language_ouput_between_successor_and_pa
rent_for_insertion(G2, successor, parent, inserted_nodes_list))
                inserted_nodes_list.clear()
            else:
                parent = list(G2.predecessors(node))[0]
                if parent in substitued_nodes_in_G2 or parent in
join_nodes:

differences.append(get_natural_language_ouput_between_successor_and_pa
rent_for_insertion(G2, successor, parent, inserted_nodes_list))
                    inserted_nodes_list.clear()
    return differences
```

Deleted nodes

To identify where the nodes are deleted, the following algorithm is used:
Perform a post-order traversal with all the nodes in the old graph so that node traversal will start from the leaves towards the roots to represent the order in which the nodes will run. If the current node is a swapped node, proceed to ignore it. Else add it to the array that will be used to store the current chain of deleted nodes. If the parent of the current node is a swapped node or belongs to the join type or if the current node is the root, then proceed to find out the children of the first element in the array. With the parent and the children, we will be able to find out the nodes between which the current chain of deleted nodes will be deleted. We will then proceed with the traversal until there are no more nodes to traverse. The code snippet that we used to identify the deleted nodes is shown below:

```
def get_natural_language_output_for_the_deleted_nodes(G1,
deleted_nodes, substitued_nodes, join_nodes):
    """
    This function is used to get the natural language output for any
deleted nodes
    """
    difference_list = []
    deleted_nodes_list = []
    deleted_nodes_in_G1 = [x[0] for x in deleted_nodes]
    differences = []
    substitued_nodes_in_G1 = [x[0] for x in substitued_nodes]
    for node in list(nx.dfs_postorder_nodes(G1, source=0)):
        if node in substitued_nodes_in_G1:
            continue
        else:
            deleted_nodes_list.append(node)
            successors_list = 
list(G1.successors(deleted_nodes_list[0]))
            if len(successors_list) == 0:
                successor = None
            else:
                successor = successors_list[0]
            if node == 0:
                parent = None

differences.append(get_natural_language_ouput_between_successor_and_pa
rent_for_deletion(G1, successor, parent, deleted_nodes_list))
                deleted_nodes_list.clear()
            else:
                parent = list(G1.predecessors(node))[0]
                if parent in substitued_nodes_in_G1 or parent in 
join_nodes:

differences.append(get_natural_language_ouput_between_successor_and_pa
rent_for_deletion(G1, successor, parent, deleted_nodes_list))
                    deleted_nodes_list.clear()
    return differences
```

Sample Test Cases

Query 1	New Query 2	Output	Explanation
select * from lineitem;	select * from customer;	The node with node label 0 of type Seq Scan has the following changes: relation name lineitem has changed into relation name customer, alias has changed from lineitem into customer and output name has changed from lineitem into customer.	Node with label 0 retains the same node label from the old graph to the new graph. However there has been some changes in the following attributes: relation name, alias and output name.
select * from customer;	select * from customer where c_nationkey = 15;	The node with node label 0 of type Seq Scan has the following changes: table filter has changed from None into (c_nationkey = 15).	Node with label 0 retains the same node label from the old graph to the new graph. However there has been some changes in the following attributes: table filter.
select * from customer where c_nationkey > 0;	select * from customer where c_nationkey = 15;	The node with node label 0 of type Seq Scan has the following changes: table filter has changed from (c_nationkey > 0) into (c_nationkey = 15).	Node with label 0 retains the same node label from the old graph to the new graph. However there has been some changes in the following attributes: table filter.
select * from lineitem;	select * from customer where c_nationkey > 0;	The node with node label 0 of type Seq Scan has the following changes: relation name lineitem has changed into relation name customer, alias has changed from	Node with label 0 retains the same node label from the old graph to the new graph. However there has been some changes in the following attributes: relation name, alias,

		lineitem into customer, table filter has changed from None into (c_nationkey > 0) and output name has changed from lineitem into customer.	table filter and output name.
select * from customer;	select (c_nationkey, c_name) from customer;	Query projections has changed from ['*'] in the old query to ['c_name', 'c_nationkey'] in the new query.	There has been a change in projection from ['*'] in the old query to ['c_name', 'c_nationkey'] in the new query.
select * from customer;	select (c_nationkey, c_name) from customer where c_nationkey = 15;	Query projections has changed from ['*'] in the old query to ['c_name', 'c_nationkey'] in the new query.The node with node label 0 of type Seq Scan has the following changes: table filter has changed from None into (c_nationkey = 15).	There has been a change in projection from ['*'] in the old query to ['c_name', 'c_nationkey'] in the new query. Node with label 0 retains the same node label from the old graph to the new graph. However there has been some changes in the following attributes: table filter.
select * from lineitem;	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	Query projections has changed from ['*'] in the old query to ['avg(l_discount) as avg_disc', 'avg(l_extendedprice) as avg_price', 'avg(l_quantity) as avg_qty', 'count(*) as count_order', 'l_linestatus', 'l_returnflag', 'sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge', 'sum(l_extendedprice	There has been a change in projection from ['*'] in the old query to ['avg(l_discount) as avg_disc', 'avg(l_extendedprice) as avg_price', 'avg(l_quantity) as avg_qty', 'count(*) as count_order', 'l_linestatus', 'l_returnflag', 'sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge',

	avg_disc, count(*) as count_order from lineitem where l_shipdate <= '1998-09-16' group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus;	* (1 - l_discount)) as sum_disc_price', 'sum(l_extendedprice) as sum_base_price', 'sum(l_quantity) as sum_qty'] in the new query.The node with node label 0 of type Seq Scan gets mapped to new node label 4 of type Seq Scan and table filter has changed from None into (l_shipdate <= '1998-09-16'::date).Aggregate (3,New), Sort (2,New), Gather Merge (1,New) and Aggregate (0,New) gets inserted after Seq Scan (4,New).	'sum(l_extendedprice * (1 - l_discount)) as sum_disc_price', 'sum(l_extendedprice) as sum_base_price', 'sum(l_quantity) as sum_qty'] in the new query. The node with node label 0 in the old graph gets mapped to node with label 4 in new graph. In the new graph, new nodes with node label 3, 2, 1, and 0 gets inserted after the node with label 4.
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_shipdate <= '1998-09-16' group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus;	select * from lineitem;	Query projections has changed from ['avg(l_discount) as avg_disc', 'avg(l_extendedprice) as avg_price', 'avg(l_quantity) as avg_qty', 'count(*) as count_order', 'l_linestatus', 'l_returnflag', 'sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge', 'sum(l_extendedprice * (1 - l_discount)) as sum_disc_price', 'sum(l_extendedprice) as sum_base_price', 'sum(l_quantity) as sum_qty'] in the old query to ['*'] in the new query.The node with node label 4 of	There has been a change in projection from ['avg(l_discount) as avg_disc', 'avg(l_extendedprice) as avg_price', 'avg(l_quantity) as avg_qty', 'count(*) as count_order', 'l_linestatus', 'l_returnflag', 'sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge', 'sum(l_extendedprice * (1 - l_discount)) as sum_disc_price', 'sum(l_extendedprice) as sum_base_price', 'sum(l_quantity) as sum_qty'] in the old query to ['*'] in the new query. The node with node label 4 in

		type Seq Scan gets mapped to new node label 0 of type Seq Scan and table filter has changed from (l_shipdate <= '1998-09-16'::date) into None.Aggregate (3,Old), Sort (2,Old), Gather Merge (1,Old) and Aggregate (0,Old) that is after Seq Scan (4,Old) gets deleted.	the old graph gets mapped to node with label 0 in new graph. In the old graph, old nodes with the label of 3, 2, 1, and 0 that are after node label 4 in the post traversal order gets deleted.
select * from orders;	select o_orderpriority, count(*) as order_count from orders as o where o_orderdate >= '1996-05-01' and o_orderdate < '1996-08-01' and exists (select * from lineitem where l_orderkey = o.o_orderkey and l_commitdate < l_receiptdate) group by o_orderpriority order by o_orderpriority;	Query projections has changed from ['*'] in the old query to ['count(*) as order_count', 'o_orderpriority'] in the new query.The node with node label 0 of type Seq Scan gets mapped to new node label 5 of type Seq Scan, alias has changed from orders into o and table filter has changed from None into ((o_orderdate >= '1996-05-01'::date) AND (o_orderdate < '1996-08-01'::date)).Index Scan (6,New) gets inserted before Nested Loop (4,New).Nested Loop (4,New) joins Seq Scan (5,New) and Index Scan (6,New) and gets inserted.Sort (3,New), Aggregate (2,New), Gather Merge (1,New) and Aggregate (0,New) gets inserted after	There has been a change in projection from ['*'] in the old query to ['count(*) as order_count', 'o_orderpriority'] in the new query. Node with node label 0 in the old graph gets mapped to node label 5 in the new graph. There is a new node of node label 6 that gets inserted before node label 4. There is a new node of node label 4 that joins nodes with label 5 and 6 together. Nodes with label 3, 2, 1 and 0 gets inserted in front of node with label 4 in the new graph.

		Nested Loop (4,New).	
select o_orderpriority, count(*) as order_count from orders as o where o_orderdate >= '1996-05-01' and o_orderdate < '1996-08-01' and exists (select * from lineitem where l_orderkey = o.o_orderkey and l_commitdate < l_receiptdate) group by o_orderpriority order by o_orderpriority;	select * from orders;	Query projections has changed from ['count(*) as order_count', 'o_orderpriority'] in the old query to ['*'] in the new query.The node with node label 5 of type Seq Scan gets mapped to new node label 0 of type Seq Scan, alias has changed from o into orders and table filter has changed from ((o_orderdate >= '1996-05-01'::date) AND (o_orderdate < '1996-08-01'::date)) into None.Index Scan (6,Old) that is before Nested Loop (4,Old) gets deleted.Nested Loop (4,Old), Sort (3,Old), Aggregate (2,Old), Gather Merge (1,Old) and Aggregate (0,Old) that is after Seq Scan (5,Old) gets deleted.	There has been a change in projection from ['count(*) as order_count', 'o_orderpriority'] in the old query to [*] in the new query. Node with node label 5 in the old graph gets mapped to node label 0 in the new graph. In the old graph, old nodes with the label of 6, 4, 3, 2, 1 and 0 gets deleted. Node with labels 4, 3, 2 and 1 are after node with label 5 in the post traversal order.
select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from orders, customer, lineitem where c_mktsegment = 'BUILDING' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < '1995-03-22' and l_shipdate	select * from orders;	Query projections has changed from ['l_orderkey', 'o_orderdate', 'o_shippriority', 'sum(l_extendedprice * (1 - l_discount)) as revenue'] in the old query to ['*'] in the new query.The node with node label 9 of type Seq Scan gets mapped to new node label 0 of type Seq Scan and table filter has changed from	There has been a change in projection from ['l_orderkey', 'o_orderdate', 'o_shippriority', 'sum(l_extendedprice * (1 - l_discount)) as revenue'] in the old query to [‘*’] in the new query. Node with node label 9 in the old graph gets mapped to node label 0 in the new graph. In the old graph, old nodes with labels 11,

> '1995-03-22' group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate limit 10;		(o_orderdate < '1995-03-22'::date) into None.Seq Scan (11,Old) and Hash (10,Old) that is before Hash Join (7,Old) gets deleted.Hash Join (7,Old) that is in between Seq Scan (9,Old) and Nested Loop (6,Old) gets deleted.Index Scan (8,Old) that is before Nested Loop (6,Old) gets deleted.Nested Loop (6,Old), Sort (5,Old), Aggregate (4,Old), Gather Merge (3,Old), Aggregate (2,Old), Sort (1,Old) and Limit (0,Old) that is after Hash Join (7,Old) gets deleted.	10, 7, 8, 6, 5, 4, 3, 2, 1, 0 gets deleted in post traversal order.
select p_brand, p_type, p_size, count(distinct ps_suppkey) as supplier_cnt from partsupp, part where p_partkey = ps_partkey and p_brand <> 'Brand#34' and p_type not like 'ECONOMY BRUSHED%' and p_size in (22, 14, 27, 49, 21, 33, 35, 28) and partsupp.ps_suppkey not in (select s_suppkey from supplier where s_comment like '%Customer%Complaints%') group by p_brand, p_type, p_size order by	select * from orders;	Query projections has changed from ['l_orderkey', 'o_orderdate', 'o_shippriority', 'sum(l_extendedprice * (1 - l_discount)) as revenue'] in the old query to ['*'] in the new query.The node with node label 9 of type Seq Scan gets mapped to new node label 0 of type Seq Scan and table filter has changed from (o_orderdate < '1995-03-22'::date) into None.Seq Scan (11,Old) and Hash (10,Old) that is before Hash Join (7,Old) gets deleted.Hash Join (7,Old) that is in between Seq Scan	There has been a change in projection from['count(distinct ps_suppkey) as supplier_cnt', 'p_brand', 'p_size', 'p_type'] in the old query to ['*'] in the new query. Node with node label 9 in the old graph gets mapped to node label 0 in the new graph. In the old graph, old nodes with the label of 11,10, 8, 7, 6, 5, 4, 3, 2, 1 gets deleted in post traversal order.

supplier_cnt desc, p_brand, p_type, p_size;		(9,Old) and Nested Loop (6,Old) gets deleted.Index Scan (8,Old) that is before Nested Loop (6,Old) gets deleted.Nested Loop (6,Old), Sort (5,Old), Aggregate (4,Old), Gather Merge (3,Old), Aggregate (2,Old), Sort (1,Old) and Limit (0,Old) that is after Hash Join (7,Old) gets deleted.	
select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from orders, lineitem where l_orderkey = o_orderkey and o_orderdate < date '1995-03-21' and l_shipdate > date '1995-03-21' group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate limit 10;	select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from customer, orders, lineitem where c_mktsegment = 'HOUSEHOLD' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-21' and l_shipdate > date '1995-03-21' group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate limit 10;	The node with node label 6 of type Hash Join gets mapped to new node label 7 of type Hash Join and hash condition has changed from (lineitem.l_orderkey = orders.o_orderkey) into (orders.o_custkey = customer.c_custkey). The node with node label 7 of type Seq Scan gets mapped to new node label 11 of type Seq Scan, relation name lineitem has changed into relation name customer, alias has changed from lineitem into customer, table filter has changed from (l_shipdate > '1995-03-21'::date) into (c_mktsegment = 'HOUSEHOLD'::bpchar) and output name has changed from lineitem into customer.The node with node label 8 of	Node with label 6 in the old graph gets mapped to node with label 7 in the new graph. Node with label 7 in the old graph gets mapped to node with label 11 in the new graph. Node with label 8 in the old graph gets mapped to node with label 10 in the new graph. The node with label 8 that has the node type Index Scan gets inserted before node with label 6 that has the node type Nested Loop in the new graph. There is a node with label 6 in the new graph that gets inserted and joins the following two nodes together: node with label 7 and node with label 8.

		type Hash gets mapped to new node label 10 of type Hash.Index Scan (8,New) gets inserted before Nested Loop (6,New).Nested Loop (6,New) joins Hash Join (7,New) and Index Scan (8,New) and gets inserted.	
select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from customer, orders, lineitem where c_mktsegment = 'HOUSEHOLD' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-21' and l_shipdate > date '1995-03-21' group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate limit 10;	select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from orders, lineitem where l_orderkey = o_orderkey and o_orderdate < date '1995-03-21' and l_shipdate > date '1995-03-21' group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate limit 10;	The node with node label 7 of type Hash Join gets mapped to new node label 6 of type Hash Join and hash condition has changed from (orders.o_custkey = customer.c_custkey) into (lineitem.l_orderkey = orders.o_orderkey).The node with node label 10 of type Hash gets mapped to new node label 8 of type Hash.The node with node label 11 of type Seq Scan gets mapped to new node label 7 of type Seq Scan, relation name customer has changed into relation name lineitem, alias has changed from customer into lineitem, table filter has changed from (c_mktsegment = 'HOUSEHOLD'::bpchar) into (l_shipdate > '1995-03-21'::date) and output name has changed from customer into lineitem.Index Scan (8,Old) that is before	Node with label 7 in the old graph gets mapped to node with label 6 in the new graph.Node with label 10 in the old graph gets mapped to node with label 8 in the new graph.Node with label 11 in the old graph gets mapped to node with label 7 in the new graph. There is a node of type Index Scan with the label 8 in the old graph that is before node with label 6 that gets deleted. There is a node with label 6 in the old graph that is in between node with label 7 and node with label 5 in the old graph that gets deleted.

		Nested Loop (6,Old) gets deleted.Nested Loop (6,Old) that is in between Hash Join (7,Old) and Sort (5,Old) gets deleted.	

find_difference.py

```
import queue
import networkx as nx
from .node_utils import set_output_name, Node
from networkx.algorithms.similarity import optimize_edit_paths
import re
import sys

new_flag = "New"
old_flag = "Old"

def node_match(node_1, node_2):
    """
    This function is used to test if two nodes are equal in network x
    """
    return node_1['custom_object'] == node_2['custom_object']

def node_substitude_cost(node_1, node_2):
    """
    This function is used to define the heuristics for node
substitution in network x
    """
    node_1_object = node_1['custom_object']
    node_2_object = node_2['custom_object']
    if node_1_object == node_2_object:
        return 0
    elif node_1_object.node_type == node_2_object.node_type and 'Scan'
in node_1_object.node_type and node_1_object.relation_name ==
node_2_object.relation_name:
        return 0.25
    elif node_1_object.node_type == node_2_object.node_type and
'Aggregate' in node_1_object.node_type and node_1_object.group_key ==
node_2_object.group_key:
        return 0.25
    elif node_1_object.node_type == node_2_object.node_type and "Sort"
in node_1_object.node_type and node_1_object.sort_key ==
node_2_object.sort_key:
        return 0.25
    elif node_1_object.node_type == node_2_object.node_type and ('Hash
Join' in node_1_object.node_type) and node_1_object.hash_cond ==
node_2_object.hash_cond:
        return 0.25
    elif node_1_object.node_type == node_2_object.node_type and
('Merge Join' in node_1_object.node_type) and node_1_object.merge_cond
== node_2_object.merge_cond:
        return 0.25
    elif node_1_object.node_type == node_2_object.node_type:
        return 0.5
    elif 'Sort' in node_1_object.node_type and 'Sort' in
node_2_object.node_type:
        return 1.0
    elif 'Scan' in node_1_object.node_type and 'Scan' in
node_2_object.node_type:
        return 1.0
    elif 'Aggregate' in node_1_object.node_type and 'Aggregate' in
node_2_object.node_type:
        return 1.0
    elif ('Join' in node_1_object.node_type or node_1_object.node_type
== 'Nested Loop') and ('Join' in node_2_object.node_type or
node_2_object.node_type == 'Nested Loop'):
        return 1.0
    return 9223372036854775807

def edge_subt_cost(old_edge_dict, new_edge_dict):
    """
    This function is used to define the heuristics for edge
substitution in network x
    """
    old_edge_parent_type = old_edge_dict['parent_type']
    old_edge_children_type = old_edge_dict['children_type']
    new_edge_parent_type = new_edge_dict['parent_type']
    new_ege_children_type = new_edge_dict['children_type']
    if old_edge_parent_type == new_edge_parent_type and
old_edge_children_type == new_ege_children_type:
        return 0
    return 1.0



def get_graph_from_query_plan(query_plan):
    """
    This function is used to create a network x graph from network x
    """
    G = nx.DiGraph()
    q = queue.Queue()
    q_node = queue.Queue()
    q.put(query_plan)
    q_node.put(None)
    current_index = 0

    while not q.empty():
        current_plan = q.get()
        parent_index = q_node.get()

        node_type = relation_name = schema = alias = group_key =
sort_key = join_type = index_name = hash_cond = table_filter \
            = index_cond = merge_cond = recheck_cond = join_filter =
subplan_name = plan_rows = output_name = None

        node_type = current_plan['Node Type']

        if 'Relation Name' in current_plan:
            relation_name = current_plan['Relation Name']
        if 'Schema' in current_plan:
            schema = current_plan['Schema']
        if 'Alias' in current_plan:
            alias = current_plan['Alias']
        if 'Group Key' in current_plan:
            group_key = current_plan['Group Key']
        if 'Sort Key' in current_plan:
            sort_key = current_plan['Sort Key']
        if 'Join Type' in current_plan:
            join_type = current_plan['Join Type']
        if 'Index Name' in current_plan:
            index_name = current_plan['Index Name']
        if 'Hash Cond' in current_plan:
            hash_cond = current_plan['Hash Cond']
        if 'Filter' in current_plan:
            table_filter = current_plan['Filter']
        if 'Index Cond' in current_plan:
            index_cond = current_plan['Index Cond']
        if 'Merge Cond' in current_plan:
            merge_cond = current_plan['Merge Cond']
        if 'Recheck Cond' in current_plan:
            recheck_cond = current_plan['Recheck Cond']
        if 'Join Filter' in current_plan:
            join_filter = current_plan['Join Filter']
        if 'Subplan Name' in current_plan:
            if "returns" in current_plan['Subplan Name']:
                name = current_plan['Subplan Name']
                subplan_name = name[name.index("$"):-1]
            else:
                subplan_name = current_plan['Subplan Name']
        if "Limit" == node_type:
            plan_rows = current_plan['Plan Rows']

        if "Scan" in node_type:
            if "Index" in node_type:
                if relation_name:
                    output_name = set_output_name(relation_name + "
with index " + index_name)
            elif "Subquery" in node_type:
                output_name = set_output_name(alias)
            else:
                output_name = set_output_name(relation_name)

        current_node = Node(current_plan['Node Type'], relation_name,
schema, alias, group_key, sort_key, join_type,
                            index_name, hash_cond, table_filter,
index_cond, merge_cond, recheck_cond, join_filter,
                            subplan_name, plan_rows, output_name)

        G.add_node(current_index, custom_object=current_node)

        if parent_index is not None:
            parent_type =
G.nodes[parent_index]['custom_object'].node_type
            children_type = current_node.node_type

            G.add_edge(parent_index, current_index, **{'parent_type':
str(parent_type), 'children_type': str(children_type)})

        if 'Plans' in current_plan:
            for item in current_plan['Plans']:
                # push child plans into queue
                q.put(item)
                # push parent for each child into queue
                q_node.put(current_index)
        current_index += 1

    return G

def find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan):
    """
    This function is used to get the difference between two query
plans in networkx.
```

```
    Will output in natural language.
    """
    result = re.search('select(.*?)from', old_query, re.IGNORECASE)
    old_query_projections = result.group(1)
    old_query_projections_list = [x.strip() for x in
old_query_projections.split(',')]
    for i in range(len(old_query_projections_list)):
        projection = old_query_projections_list[i]
        open_bracket_count = projection.count("(")
        closed_bracket_count = projection.count(")")
        while open_bracket_count > closed_bracket_count:
            projection = projection.replace('(', '', 1)
            open_bracket_count = open_bracket_count - 1
        while closed_bracket_count > open_bracket_count:
            projection = projection.replace(')', '', 1)
            closed_bracket_count = closed_bracket_count - 1
        old_query_projections_list[i] = projection
    old_query_projections_list.sort()
    result = re.search('select(.*?)from', new_query, re.IGNORECASE)
    new_query_projections = result.group(1)
    new_query_projections_list = [x.strip() for x in
new_query_projections.split(',')]
    for i in range(len(new_query_projections_list)):
        projection = new_query_projections_list[i]
        open_bracket_count = projection.count("(")
        closed_bracket_count = projection.count(")")
        while open_bracket_count > closed_bracket_count:
            projection = projection.replace('(', '', 1)
            open_bracket_count = open_bracket_count - 1
        while closed_bracket_count > open_bracket_count:
            projection = projection.replace(')', '', 1)
            closed_bracket_count = closed_bracket_count - 1
        new_query_projections_list[i] = projection
    new_query_projections_list.sort()
    G1 = get_graph_from_query_plan(old_query_plan)
    G2 = get_graph_from_query_plan(new_query_plan)
    generator = optimize_edit_paths(G1, G2, node_match=node_match,
node_subst_cost=node_substitude_cost, edge_subst_cost=edge_subt_cost)
    node_edit_path, edge_edit_path, cost = list(generator)[0]
    if old_query_projections_list == new_query_projections_list:
        return get_the_difference_in_natural_language(G1, G2,
node_edit_path, edge_edit_path, cost)
    else:
        old_query_projections_list.sort()
        new_query_projections_list.sort()
        query_difference_string = "Query projections has changed from
" + str(old_query_projections_list) + " in the old query to " +
str(new_query_projections_list) + " in the new query."
        natural_language_difference_string =
get_the_difference_in_natural_language(G1, G2, node_edit_path,
edge_edit_path, cost)
        if  natural_language_difference_string == "Nothing has
changed!":
            return query_difference_string
        else:
            return query_difference_string + "" +
natural_language_difference_string

def get_the_difference_in_natural_language(G1, G2, node_edit_path,
edge_edit_path, cost):
    """
    This function is used to get natural language output of all the
changes between
    two graphs in network x
    """
    if cost == 0:
        return "Nothing has changed!"
    node_difference_strings = get_node_differences(G1, G2,
node_edit_path)
    node_difference_strings = [node_difference_string for
node_difference_string in node_difference_strings if
node_difference_string != "N.A."]
    if len(node_difference_strings) == 0:
        return "Nothing has changed!"
    return "".join(node_difference_strings)


def get_node_differences(G1, G2, node_edit_path):
    """
    This function is used to get the differences between two graphs
using network x
    """
    node_differences = []
    substitued_nodes = [x for x in node_edit_path if x[0] is not None
and x[1] is not None]
    inserted_nodes = [x for x in node_edit_path if x[0] is None and
x[1] is not None]
    deleted_nodes = [x for x in node_edit_path if x[0] is not None and
x[1] is None]
```

```
    join_nodes_for_G2 = [x for x in G2.nodes() if
G2.nodes[x]['custom_object'].node_type == "Nested Loop" or "Join" in
G2.nodes[x]['custom_object'].node_type]
    join_nodes_for_G1 = [x for x in G1.nodes() if
G1.nodes[x]['custom_object'].node_type == "Nested Loop" or "Join" in
G1.nodes[x]['custom_object'].node_type]
    for node in substitued_nodes:
        node_difference =
find_difference_between_two_nodes(G1.nodes[node[0]]['custom_object'],
G2.nodes[node[1]]['custom_object'], node[0], node[1])
        node_differences.append(node_difference)
    if len(inserted_nodes) != 0:

node_differences.extend(get_natural_language_output_for_the_inserted_n
odes(G2, inserted_nodes, substitued_nodes, join_nodes_for_G2))
    if len(deleted_nodes) != 0:

node_differences.extend(get_natural_language_output_for_the_deleted_no
des(G1, deleted_nodes, substitued_nodes, join_nodes_for_G1))
    return node_differences

def find_difference_between_two_nodes(node_1, node_2, node_1_label,
node_2_label):
    """
    This function is used to get the differences between two nodes
    """
    return node_1.compare_differences(node_2, node_1_label,
node_2_label)

def get_natural_language_output_for_the_inserted_nodes(G2,
inserted_nodes, substitued_nodes, join_nodes):
    """
    This function is used to get the natural language output for any
inserted nodes
    """
    difference_list = []
    inserted_nodes_list = []
    inserted_nodes_in_G2 = [x[1] for x in inserted_nodes]
    differences = []
    substitued_nodes_in_G2 = [x[1] for x in substitued_nodes]
    for node in list(nx.dfs_postorder_nodes(G2, source=0)):
        if node in substitued_nodes_in_G2:
            continue
        if node in join_nodes:
```

```
differences.append(get_natural_language_ouput_for_join_queries(G2,
node, new_flag))
        else:
            inserted_nodes_list.append(node)
            successors_list =
list(G2.successors(inserted_nodes_list[0]))
            if len(successors_list) == 0:
                successor = None
            else:
                successor = successors_list[0]
            if node == 0:
                parent = None

differences.append(get_natural_language_ouput_between_successor_and_pa
rent_for_insertion(G2, successor, parent, inserted_nodes_list))
                inserted_nodes_list.clear()
            else:
                parent = list(G2.predecessors(node))[0]
                if parent in substitued_nodes_in_G2 or parent in
join_nodes:

differences.append(get_natural_language_ouput_between_successor_and_pa
rent_for_insertion(G2, successor, parent, inserted_nodes_list))
                    inserted_nodes_list.clear()
    return differences


def get_natural_language_output_for_the_deleted_nodes(G1,
deleted_nodes, substitued_nodes, join_nodes):
    """
    This function is used to get the natural language output for any
deleted nodes
    """
    difference_list = []
    deleted_nodes_list = []
    deleted_nodes_in_G1 = [x[0] for x in deleted_nodes]
    differences = []
    substitued_nodes_in_G1 = [x[0] for x in substitued_nodes]
    for node in list(nx.dfs_postorder_nodes(G1, source=0)):
        if node in substitued_nodes_in_G1:
            continue
        else:
            deleted_nodes_list.append(node)
            successors_list = 
list(G1.successors(deleted_nodes_list[0]))
            if len(successors_list) == 0:
                successor = None
            else:
                successor = successors_list[0]
            if node == 0:
                parent = None

differences.append(get_natural_language_ouput_between_successor_and_pa
rent_for_deletion(G1, successor, parent, deleted_nodes_list))
                deleted_nodes_list.clear()
            else:
                parent = list(G1.predecessors(node))[0]
                if parent in substitued_nodes_in_G1 or parent in 
join_nodes:

differences.append(get_natural_language_ouput_between_successor_and_pa
rent_for_deletion(G1, successor, parent, deleted_nodes_list))
                    deleted_nodes_list.clear()
    return differences

def get_natural_language_output_with_node_type_from_node_index(G, 
index, flag):
    """
    This function is used to get the natual language output with node 
type and its index
    along with information that identifies whether it belongs to the 
old graph or the new
    graph
    """
    node_type = G.nodes[index]['custom_object'].node_type
    return str(node_type) + " (" + str(index) + "," + flag + ")"


def get_natural_language_ouput_for_join_queries(G, join_node_index, 
flag):
    """
    This function is used to get the natual language output for join 
queries
    """
    successors = list(G.successors(join_node_index))
    successors_with_node_type = 
[get_natural_language_output_with_node_type_from_node_index(G, 
successor, flag) for successor in successors]
```

```
    return
get_natural_language_output_with_node_type_from_node_index(G,
join_node_index, flag) + " joins " +
get_natural_language_connection_between_objects_in_list(successors_wit
h_node_type) + \
        " and gets inserted.\n"



def
get_natural_language_ouput_between_successor_and_parent_for_insertion(
G2, successor, parent, inserted_nodes):
    """
    This function is used to get the natural language output that
identifies the successor and
    the parent of inserted nodes
    """
    inserted_nodes_with_nodes_type =
[get_natural_language_output_with_node_type_from_node_index(G2, index,
new_flag) for index in inserted_nodes]
    if successor != None and parent != None:
        return
str(get_natural_language_connection_between_objects_in_list(inserted_n
odes_with_nodes_type)) + " gets inserted in between " +
get_natural_language_output_with_node_type_from_node_index(G2,
successor, new_flag) + " and " +
get_natural_language_output_with_node_type_from_node_index(G2, parent,
new_flag) +".\n"
    if successor == None and parent == None:
        return
str(get_natural_language_connection_between_objects_in_list(inserted_n
odes_with_nodes_type)) + " gets inserted.\n"
    if successor == None:
        return
str(get_natural_language_connection_between_objects_in_list(inserted_n
odes_with_nodes_type)) + " gets inserted before " +
get_natural_language_output_with_node_type_from_node_index(G2, parent,
new_flag) + ".\n"
    return
str(get_natural_language_connection_between_objects_in_list(inserted_n
odes_with_nodes_type)) + " gets inserted after " +
get_natural_language_output_with_node_type_from_node_index(G2,
successor, new_flag) + ".\n"
```

```
def
get_natural_language_ouput_between_successor_and_parent_for_deletion(G
1, successor, parent, deleted_nodes):
    """
    This function is used to get the natural language output that
identifies the successor and
    the parent of deleted nodes
    """
    deleted_nodes_with_nodes_type =
[get_natural_language_output_with_node_type_from_node_index(G1, index,
old_flag) for index in deleted_nodes]
    if successor != None and parent != None:
        return
str(get_natural_language_connection_between_objects_in_list(deleted_no
des_with_nodes_type)) + " that is in between " +
get_natural_language_output_with_node_type_from_node_index(G1,
successor, old_flag) + " and " +
get_natural_language_output_with_node_type_from_node_index(G1, parent,
old_flag) + " gets deleted" + ".\n"
    if successor == None and parent == None:
        return
str(get_natural_language_connection_between_objects_in_list(deleted_no
des_with_nodes_type)) + " gets deleted.\n"
    if successor == None:
        return
str(get_natural_language_connection_between_objects_in_list(deleted_no
des_with_nodes_type)) + " that is before " +
get_natural_language_output_with_node_type_from_node_index(G1, parent,
old_flag) + " gets deleted" + ".\n"
    return
str(get_natural_language_connection_between_objects_in_list(deleted_no
des_with_nodes_type)) + " that is after " +
get_natural_language_output_with_node_type_from_node_index(G1,
successor, old_flag) + " gets deleted" +  ".\n"

def get_natural_language_connection_between_objects_in_list(objects):
    """
    This function is used to get natural language connection between
objects in a list
    so that objects inside the list will be joined with commas and
ands correctly
    """
    if len(objects) == 1:
        return objects[0]
    else:
        last_object = objects[-1]
        objects_up_to_last_object = objects[:-1]
        natural_language_connection = ",
".join(objects_up_to_last_object)
        natural_language_connection += " and " + last_object
        return natural_language_connection
```

main_parser.py

```
import networkx as nx
import queue
import re
try:
    from utils.singleton import Singleton
except:
    from src.utils.singleton import Singleton
from .find_difference import node_substitude_cost, edge_subt_cost,
get_the_difference_in_natural_language, node_match
from .node_utils import set_output_name, Node
from networkx.algorithms.similarity import optimize_edit_paths

class Parser(metaclass=Singleton):
    """
    Define an object used to store graphs and find the differences
between the graphs
    """

    old_graph = nx.DiGraph()
    new_graph = nx.DiGraph()

    def update_graphs_with_new_query_plans(self, query_plan_1,
query_plan_2):
        """
        This function is used to update the graphs by regenearting
them from
        new query plan
        """
        if query_plan_1 is None:
            self.old_graph.clear()
        else:
            self.update_graph_from_query_plan(self.old_graph,
query_plan_1)
        if query_plan_2 is None:
            self.new_graph.clear()
        else:
            self.update_graph_from_query_plan(self.new_graph,
query_plan_2)

    def get_graphs_for_visualizations(self):
        """
        This function is used to generate graphs that will be used for
visualization
        """
        return self.old_graph.reverse(), self.new_graph.reverse()

    def get_difference_between_old_and_new_graphs(self, old_query,
new_query):
        """
        This function is used to get the difference between old and
new graphs
        """
        result = re.search('select(.*?)from', old_query,
re.IGNORECASE)
        old_query_projections = result.group(1)
        old_query_projections_list = [x.strip() for x in
old_query_projections.split(',')]
        for i in range(len(old_query_projections_list)):
            projection = old_query_projections_list[i]
            open_bracket_count = projection.count("(")
            closed_bracket_count = projection.count(")")
            while open_bracket_count > closed_bracket_count:
                projection = projection.replace('(', '', 1)
                open_bracket_count = open_bracket_count - 1
            while closed_bracket_count > open_bracket_count:
                projection = projection.replace(')', '', 1)
                closed_bracket_count = closed_bracket_count - 1
            old_query_projections_list[i] = projection
        old_query_projections_list.sort()
        result = re.search('select(.*?)from', new_query,
re.IGNORECASE)
        new_query_projections = result.group(1)
        new_query_projections_list = [x.strip() for x in
new_query_projections.split(',')]
        for i in range(len(new_query_projections_list)):
            projection = new_query_projections_list[i]
            open_bracket_count = projection.count("(")
            closed_bracket_count = projection.count(")")
            while open_bracket_count > closed_bracket_count:
                projection = projection.replace('(', '', 1)
                open_bracket_count = open_bracket_count - 1
            while closed_bracket_count > open_bracket_count:
                projection = projection.replace(')', '', 1)
                closed_bracket_count = closed_bracket_count - 1
            new_query_projections_list[i] = projection
        new_query_projections_list.sort()
        generator = optimize_edit_paths(self.old_graph, 
self.new_graph, node_match=node_match, 
node_subst_cost=node_substitude_cost, edge_subst_cost=edge_subt_cost)
        node_edit_path, edge_edit_path, cost = list(generator)[0]
        if old_query_projections_list == new_query_projections_list:
            return 
get_the_difference_in_natural_language(self.old_graph, self.new_graph, 
node_edit_path, edge_edit_path, cost)
        else:
            old_query_projections_list.sort()
            new_query_projections_list.sort()
            query_difference_string = "Query projections has changed 
from " + str(old_query_projections_list) + " in the old query to " + 
str(new_query_projections_list) + " in the new query."
            natural_language_difference_string = 
get_the_difference_in_natural_language(self.old_graph, self.new_graph, 
node_edit_path, edge_edit_path, cost)
            if  natural_language_difference_string == "Nothing has 
changed!":
                return query_difference_string
            else:
                return query_difference_string + "\n" + 
natural_language_difference_string

    def update_graph_from_query_plan(self, G, query_plan):
        """
        This function is used to update the current graph with 
information
        from the query plan
        """
        G.clear()
        q = queue.Queue()
        q_node = queue.Queue()
        q.put(query_plan)
        q_node.put(None)
        current_index = 0

        while not q.empty():
            current_plan = q.get()
            parent_index = q_node.get()

            node_type = relation_name = schema = alias = group_key = 
sort_key = join_type = index_name = hash_cond = table_filter \
                = index_cond = merge_cond = recheck_cond = join_filter 
= subplan_name = plan_rows = output_name = None
```

```
            node_type = current_plan['Node Type']

            if 'Relation Name' in current_plan:
                relation_name = current_plan['Relation Name']
            if 'Schema' in current_plan:
                schema = current_plan['Schema']
            if 'Alias' in current_plan:
                alias = current_plan['Alias']
            if 'Group Key' in current_plan:
                group_key = current_plan['Group Key']
            if 'Sort Key' in current_plan:
                sort_key = current_plan['Sort Key']
            if 'Join Type' in current_plan:
                join_type = current_plan['Join Type']
            if 'Index Name' in current_plan:
                index_name = current_plan['Index Name']
            if 'Hash Cond' in current_plan:
                hash_cond = current_plan['Hash Cond']
            if 'Filter' in current_plan:
                table_filter = current_plan['Filter']
            if 'Index Cond' in current_plan:
                index_cond = current_plan['Index Cond']
            if 'Merge Cond' in current_plan:
                merge_cond = current_plan['Merge Cond']
            if 'Recheck Cond' in current_plan:
                recheck_cond = current_plan['Recheck Cond']
            if 'Join Filter' in current_plan:
                join_filter = current_plan['Join Filter']
            if 'Subplan Name' in current_plan:
                if "returns" in current_plan['Subplan Name']:
                    name = current_plan['Subplan Name']
                    subplan_name = name[name.index("$"):-1]
                else:
                    subplan_name = current_plan['Subplan Name']
            if "Limit" == node_type:
                plan_rows = current_plan['Plan Rows']

            if "Scan" in node_type:
                if "Index" in node_type:
                    if relation_name:
                        output_name = set_output_name(relation_name +
" with index " + index_name)
                elif "Subquery" in node_type:
                    output_name = set_output_name(alias)
                else:
                    output_name = set_output_name(relation_name)

            current_node = Node(current_plan['Node Type'],
relation_name, schema, alias, group_key, sort_key, join_type,
                                index_name, hash_cond, table_filter,
index_cond, merge_cond, recheck_cond, join_filter,
                                subplan_name, plan_rows, output_name)

            G.add_node(current_index, custom_object=current_node)

            if parent_index is not None:
                parent_type =
G.nodes[parent_index]['custom_object'].node_type
                children_type = current_node.node_type

                G.add_edge(parent_index, current_index,
**{'parent_type': str(parent_type), 'children_type':
str(children_type)})

            if 'Plans' in current_plan:
                for item in current_plan['Plans']:
                    # push child plans into queue
                    q.put(item)
                    # push parent for each child into queue
                    q_node.put(current_index)
            current_index += 1
```

node_utils.py

```
def set_output_name(output_name):
    """
    This function is used to define the output name
    """
    try:
        if "T" == output_name[0] and output_name[1:].isdigit():
            output_name = int(output_name[1:])
        else:
            output_name = output_name
    except:
        output_name = None
    return output_name

class Node(object):
    """
    Define an object used to represent a node
    """
    def __init__(self, node_type, relation_name, schema, alias,
group_key, sort_key, join_type, index_name,
            hash_cond, table_filter, index_cond, merge_cond,
recheck_cond, join_filter, subplan_name,
            plan_rows, output_name):
        self.node_type = node_type
        self.relation_name = relation_name
        self.schema = schema
        self.alias = alias
        self.group_key = group_key
        self.sort_key = sort_key
        self.join_type = join_type
        self.index_name = index_name
        self.hash_cond = hash_cond
        self.table_filter = table_filter
        self.index_cond = index_cond
        self.merge_cond = merge_cond
        self.recheck_cond = recheck_cond
        self.join_filter = join_filter
        self.subplan_name = subplan_name
        self.plan_rows = plan_rows
        self.output_name = output_name

    def __eq__(self, other):
        """
        This function is used to check if two node types are equal
        """
        if not isinstance(other, Node):
            # don't attempt to compare against unrelated types
            return NotImplemented

        return self.node_type == other.node_type and
self.relation_name == other.relation_name and \
        self.schema == other.schema and self.alias == other.alias and
self.group_key == other.group_key and \
        self.sort_key == other.sort_key and self.join_type ==
other.join_type and self.index_name == other.index_name \
        and self.hash_cond == other.hash_cond and self.table_filter ==
other.table_filter and self.index_cond == other.index_name  \
        and self.merge_cond == other.merge_cond and self.recheck_cond
== other.recheck_cond and self.join_filter == other.join_filter \
        and self.subplan_name == other.subplan_name and self.plan_rows
== other.plan_rows and self.output_name == other.output_name

    def compare_differences(self, other, original_label,
current_label):
        """
        This function is used to compare the differences between two
nodes
        """
        differences = []
        if not(self.node_type == other.node_type or ("Scan" in
self.node_type and "Scan" in other.node_type) or \
            ("Aggregate" in self.node_type and "Aggregate" in
other.node_type) or ((self.node_type == "Nested Loop" or "Join" in
self.node_type)\
                and (other.node_type == "Nested Loop" or "Join" in
other.node_type))):
            difference = "The node with node label " +
str(original_label) + " of type " + str(self.node_type) + " has
evolved into " + str(current_label) + " of type " +
str(other.node_type)
            differences.append(difference)
        else:
            if (original_label != current_label or self.node_type !=
other.node_type):
                difference = "The node with node label " +
str(original_label) + " of type " + str(self.node_type) + " gets
mapped to new node label " + str(current_label) + " of type " +
str(other.node_type)
                differences.append(difference)
            if self.relation_name != other.relation_name:
                difference =  "relation name " +
str(self.relation_name) + " has changed into " +  "relation name " +
str(other.relation_name)
                differences.append(difference)
            if self.schema != other.schema:
                difference = "schema has changed from " +
str(self.schema) + " into " + str(other.schema)
                differences.append(difference)
            if self.alias != other.alias:
                difference = "alias has changed from " +
str(self.alias) + " into " + str(other.alias)
                differences.append(difference)
            if self.group_key != other.group_key:
                difference = "group key has changed from " +
str(self.group_key) + " into " + str(other.group_key)
                differences.append(difference)
            if self.sort_key != other.sort_key:
                difference = "sort key has changed from " +
str(self.sort_key) + " into " + str(other.sort_key)
                differences.append(difference)
            if self.join_type != other.join_type:
                difference = "join type has changed from " +
str(self.join_type) + " into " + str(other.join_type)
                differences.append(difference)
            if self.index_name != other.index_name:
                difference = "index name has changed from " +
str(self.index_name) + " into " + str(other.index_name)
                differences.append(difference)
            if self.hash_cond != other.hash_cond:
                difference = "hash condition has changed from " +
str(self.hash_cond) + " into " + str(other.hash_cond)
                differences.append(difference)
            if self.table_filter != other.table_filter:
                difference =  "table filter has changed from " +
str(self.table_filter) + " into " + str(other.table_filter)
                differences.append(difference)
            if self.index_cond != other.index_cond:
                difference =  "index condition has changed from " +
str(self.index_cond) + " into " + str(other.index_cond)
                differences.append(difference)
            if self.merge_cond != other.merge_cond:
                difference =  "merge condition has changed from " +
str(self.merge_cond) + " into " + str(other.merge_cond)
                differences.append(difference)
            if self.recheck_cond != other.recheck_cond:
                difference =  "recheck condition has changed from " +
str(self.recheck_cond) + " into " + str(other.recheck_cond)
                differences.append(difference)
            if self.join_filter != other.join_filter:
                difference =  "join filter has changed from " +
str(self.join_filter) + " into " + str(other.join_filter)
                differences.append(difference)
            if self.subplan_name != other.subplan_name:
                difference =  "subplan name has changed from " +
str(self.subplan_name) + " into " + str(other.subplan_name)
                differences.append(difference)
            if self.output_name != other.output_name:
                difference =  "output name has changed from " +
str(self.output_name) + " into " + str(other.output_name)
                differences.append(difference)
        if len(differences) == 0:
            return "N.A."
        if len(differences) == 1:
            difference = differences[0]
            if (original_label == current_label and self.node_type ==
other.node_type):
                return "The node with node label " +
str(original_label) + " of type " + str(self.node_type) + " has the
following changes: " \
                    + difference[0] +  difference [1:] + ".\n"
            else:
                return difference[0].upper() +  difference [1:] +
".\n"
        else:
            last_difference = differences[-1]
            differences_up_to_last = differences[:-1]
            difference_string = ", ".join(differences_up_to_last)
            difference_string += " and " + last_difference + ".\n"
            if (original_label == current_label and self.node_type ==
other.node_type):
                return "The node with node label " +
str(original_label) + " of type " + str(self.node_type) + " has the
following changes: " \
                    + difference_string[0] +  difference_string[1:]
            else:
                return difference_string[0].upper() +
difference_string[1:]
```

postgres_wrapper.py

```
import psycopg2
import json
try:
    from utils.singleton import Singleton
except:
    from src.utils.singleton import Singleton

class PostgresWrapper(metaclass=Singleton):
    """
    Define an object used to wrap around postgres
    """

    def connect_to_postgres_db(self, host, dbname, user, password,
port=5432):
        """
        This function is used to connect to postgres db.
        """
        try:
            conn = psycopg2.connect(
                host = host,
                dbname = dbname,
                user = user,
                password = password,
                port = port
            )
            self.conn = conn
            return conn, True
        except Exception as e:
            return str(e), False

    def get_query_plan_of_query(self, query):
        """
        This function is used to get query plan of a query from
postgres db.
        """
        try:
            cursor = self.conn.cursor()
            cursor.execute("explain (format json) " + query)
            result = cursor.fetchall()[0][0][0]['Plan']
            cursor.close()
            return result, True
        except Exception as e:
            self.conn.rollback()
            return str(e), False
```

singleton.py

```
class Singleton(type):
    """
    Define an Instance operation that lets clients access its unique
    instance.
    """

    def __init__(cls, name, bases, attrs, **kwargs):
        super().__init__(name, bases, attrs)
        cls._instance = None

    def __call__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__call__(*args, **kwargs)
        return cls._instance
```

main.py

```
from postgres_interface.postgres_wrapper import PostgresWrapper
from qt_parser.main_parser import Parser
```

```
import tkinter as tk
import networkx as nx
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.pyplot as plt
import random
import re


LARGE_FONT = ("Verdana",12)
HEIGHT = 500
WIDTH = 600
postgres_wrapper = PostgresWrapper()
newParser = Parser()
conn = None
G1 = None
G2 = None

def makeEntry(parent, caption, width=None, **options):
    tk.Label(parent, text=caption).pack(side="top")
    entry = tk.Entry(parent)
    if width:
        entry.config(width=width)
    entry.pack(side="top", **options)
    return entry

def set_input(textbox, value):
    textbox.config(state='normal')
    textbox.delete('1.0', tk.END)
    textbox.insert(tk.END, value)
    textbox.config(state='disabled')

def handleDBStatus(connected, db_status):
    if(connected):
        db_status.config(text = 'Database Connected',bg = 'green',
font = ("Verdana",10))
    else:
        tk.messagebox.showerror("Error","Connection failed")
        db_status.config(text= "Database Disconnected", bg='red', font
= ("Verdana",10))

def submitLogin(host, dbname, user, password, port, db_status):
    empty = False
    connected = False
    inputs = {'Database URL' : host, 'Database Name' : dbname, "User":
user, "Password" : password}
```

```
    err_msg = ''
    for key,value in inputs.items():
        if len(value.strip()) == 0:
            empty = True
            err_msg += key + "\n"
    if(empty):
        tk.messagebox.showerror("Please fill", err_msg)

    else:
        try:
            port_no = int(port)
        except:
            port_no = None
        if not port_no:
            conn, connected = 
postgres_wrapper.connect_to_postgres_db(host, dbname, user, password)
            handleDBStatus(connected,db_status)
        else:
            conn, connected = 
postgres_wrapper.connect_to_postgres_db(host, dbname, user, password, 
port_no)
            handleDBStatus(connected,db_status)

def getQueryPlan(q1,q2,r1):
    old_query = q1
    new_query = q2

    old_query = re.sub("\s+" , " ", old_query)
    new_query = re.sub("\s+" , " ", new_query)

    result_1, success_1 = 
postgres_wrapper.get_query_plan_of_query(old_query)
    result_2, success_2 = 
postgres_wrapper.get_query_plan_of_query(new_query)

    if not success_2 and not success_1:
        newParser.update_graphs_with_new_query_plans(None, None)
        plan = "Both inputs are invalid. Please input valid SQL 
queries in the textbox."
        set_input(r1,plan)
    elif not success_2:
        newParser.update_graphs_with_new_query_plans(result_1, None)
        plan = "Invalid new query. Please input a valid SQL query."
        set_input(r1,plan)
    elif not success_1 :
        newParser.update_graphs_with_new_query_plans(None, result_2)
        plan = "Invalid old query. Please input a valid SQL query."
        set_input(r1, plan)
    else:
        newParser.update_graphs_with_new_query_plans(result_1,
result_2)
        plan =
newParser.get_difference_between_old_and_new_graphs(old_query,
new_query)
        set_input(r1,plan)

def hierarchy_pos(G, root, levels=None, width=1., height=1.):
    '''If there is a cycle that is reachable from root, then this will
see infinite recursion.
       G: the graph
       root: the root node
       levels: a dictionary
               key: level number (starting from 0)
               value: number of nodes in this level
       width: horizontal space allocated for drawing
       height: vertical space allocated for drawing'''
    TOTAL = "total"
    CURRENT = "current"
    def make_levels(levels, node=root, currentLevel=0, parent=None):
        """Compute the number of nodes for each level
        """
        if not currentLevel in levels:
            levels[currentLevel] = {TOTAL : 0, CURRENT : 0}
        levels[currentLevel][TOTAL] += 1
        neighbors = G.neighbors(node)
        for neighbor in neighbors:
            if not neighbor == parent:
                levels =  make_levels(levels, neighbor, currentLevel +
1, node)
        return levels

    def make_pos(pos, node=root, currentLevel=0, parent=None,
vert_loc=0):
        dx = 1/levels[currentLevel][TOTAL]
        left = dx/2
        pos[node] = ((left + dx*levels[currentLevel][CURRENT])*width,
vert_loc)
        levels[currentLevel][CURRENT] += 1
        neighbors = G.neighbors(node)
        for neighbor in neighbors:
            if not neighbor == parent:
                pos = make_pos(pos, neighbor, currentLevel + 1, node,
vert_loc-vert_gap)
        return pos
    if levels is None:
        levels = make_levels({})
    else:
        levels = {l:{TOTAL: levels[l], CURRENT:0} for l in levels}
    vert_gap = height / (max([l for l in levels])+1)
    return make_pos({})

class SeaofFrames(tk.Tk):
    def __init__(self,*args,**kwargs):
        tk.Tk.__init__(self,*args,**kwargs)
        container = tk.Frame(self,width=100, height=100,
background="bisque")
        container.pack(side = "top", fill = "both", expand = True)
        container.grid_rowconfigure(0,weight=1)
        container.grid_columnconfigure(0,weight=1)

        self.frames = {}
        for F in (HomePage, QueryPage, QPTPage):
            frame = F(container,self)
            self.frames[F] = frame
            frame.grid(row = 0, column = 0,sticky ="nsew")

        self.show_frame(HomePage)


    def show_frame(self, cont):
        frame = self.frames[cont]
        frame.tkraise()
        if isinstance(frame, QPTPage):
            frame.refresh(frame.empty_label1, frame.empty_label2)
        frame.tkraise()

class BasePage(tk.Frame):
    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)

        menu_frame = tk.Frame(self,  bg='#0082FF')
        menu_frame.pack()
        title_frame = tk.Frame(self,  bg='#FF2E00')
        title_frame.pack(side ='left')
        button = tk.Button(menu_frame, text = "Home Page",

command=lambda:controller.show_frame(HomePage))
        button.pack(side = 'left', pady=10,padx=10, fill ='both')

        button2 = tk.Button(menu_frame, text = "Query Page",

command=lambda:controller.show_frame(QueryPage),)
        button2.pack(side = 'left',pady=10,padx=10,fill='both')
        button3 = tk.Button(menu_frame, text = "Query Plan Tree",

command=lambda:controller.show_frame(QPTPage))
        button3.pack(side = 'left',pady=10,padx=10,fill='both')

#Home Page
class HomePage(BasePage):
    title = 'Home Page'

    def __init__(self, parent, controller):
        BasePage.__init__(self, parent, controller)
        tk.Label(self, text= self.title, font =
LARGE_FONT).pack(pady=10,padx=10)


        db_status = tk.Label(self, text= "Database Disconnected",
bg='red', font = ("Verdana",10))
        db_status.pack(pady = 5, padx = 10)

        entry_frame = tk.Frame(self, bd=1, relief="solid")
        entry_frame.place(relx = 0.2, rely =0.2, relwidth = 0.6,
relheight=0.6)


        url = makeEntry(entry_frame, "Database URL:",padx = 10,
pady=10,fill ='both')

        dbname = makeEntry(entry_frame, "Database Name:",padx = 10,
pady=10,fill ='both')

        port = makeEntry(entry_frame, "Database Port (Port 5432 if
empty):",padx = 10, pady=10,fill ='both')

        user = makeEntry(entry_frame, "User:",padx = 10, pady=10,fill
='both')
```

```
        password = makeEntry(entry_frame, "Password:",padx = 10,
pady=10,fill ='both')

        submit_button = tk.Button(entry_frame, text = "Submit",
        command = lambda:submitLogin(url.get(), dbname.get(),
user.get(),password.get(), port.get(), db_status))
        submit_button.pack(side = 'bottom', pady=10,padx=10)

class QueryPage(BasePage):
    title = "Query Page"

    def __init__(self,parent,controller):
        BasePage.__init__(self,parent,controller)
        tk.Label(self, text= self.title, font =
LARGE_FONT).pack(pady=10,padx=10)


        frame1_relx = 0.05
        frame1_rely = 0.1
        frame1_relwidth = 0.40
        frame1_relheight = 0.85
        frame = tk.Frame(self,bg='#900C3F')
        frame.place(relx = frame1_relx, rely=frame1_rely,
relwidth=frame1_relwidth, relheight=frame1_relheight)

        frame2_relx = 0.55
        frame2_rely = 0.1
        frame2_relwidth = 0.40
        frame2_relheight = 0.85
        frame2 = tk.Frame(self,bg='#900C3F')
        frame2.place(relx= frame2_relx,rely=frame2_rely,
relwidth=frame2_relwidth, relheight=frame2_relheight)

        #query textbox
        q1_relx = 0.025
        q1_rely = 0.025
        q2_relx = 0.025
        q2_rely  = 0.525
        q_relwidth = 0.95
        q_relheight = 0.45

        textbox = tk.Text(frame,font=40)
        textbox.insert(tk.END,'Query 1')
        textbox.place(relx = q1_relx,rely= q1_rely,
relwidth=q_relwidth, relheight=q_relheight)
```

```
        #
        textbox2 = tk.Text(frame, font=40)
        textbox2.insert(tk.END,'Query 2')
        textbox2.place(relx = q2_relx,rely=q2_rely,
relwidth=q_relwidth, relheight=q_relheight)

        #query plan display box
        bg_color = 'white'
        r_relx = 0.025
        r_rely = 0.025
        r_relwidth = 0.95
        r_relheight = 0.95


        results = tk.Text(frame2)
        results.insert(tk.END,'Output')
        results.config(font=40, bg=bg_color, state='disabled')
        results.place(relx=r_relx,rely =r_rely,relwidth= r_relwidth,
relheight=r_relheight)

        submit = tk.Button(self,text='Submit',
command=lambda:getQueryPlan(textbox.get("1.0","end-
1c"),textbox2.get("1.0","end-1c"),

results))
        submit.place(relx=0.5, rely=0.5, relwidth = 0.08,
relheight=0.08, anchor='center')

#Query PLan Tree Page
class QPTPage(BasePage):
    title = "Query Tree Page"

    def __init__(self,parent,controller):
        BasePage.__init__(self,parent,controller)
        tk.Label(self, text= self.title, font =
LARGE_FONT).pack(pady=10,padx=10)

        G1, G2 = newParser.get_graphs_for_visualizations()

        #networkx graph1
        self.f1 = plt.figure(figsize=(5,5))
        self.a1 = self.f1.add_subplot(111)
        nx.draw_networkx(G1,ax=self.a1)
        self.canvas1 = FigureCanvasTkAgg(self.f1,self)
        self.canvas1.get_tk_widget().pack(side='left', fill =tk.BOTH,
expand = True)

        self.empty_label1 = tk.Label(self, text = "no query plan to
show")
        self.empty_label1.place(relx = 0.15, rely = 0.5, relwidth =
0.2, relheight=0.1)


        #networkx graph2
        self.f2 = plt.figure(figsize=(5,5))
        self.a2 = self.f2.add_subplot(111)
        nx.draw_networkx(G2,ax=self.a2)
        self.canvas2 = FigureCanvasTkAgg(self.f2, self)
        self.canvas2.get_tk_widget().pack(side='left', fill =tk.BOTH,
expand = True)

        self.empty_label2 = tk.Label(self, text = "no query plan to
show")
        self.empty_label2.place(relx = 0.65, rely = 0.5, relwidth =
0.2, relheight=0.1)

    def refresh(self,empty_label1,empty_label2):
        self.a1.clear()
        self.a2.clear()
        self.a1.axis('off')
        self.a2.axis('off')

        G1, G2 = newParser.get_graphs_for_visualizations()


        if (len(G1.nodes()) !=0 ):
            empty_label1.place_forget()
            pos_1 = hierarchy_pos(G1.reverse(),0)
            nx.draw(G1, ax=self.a1, pos=pos_1, with_labels=True)
            self.canvas1.draw()
        else:
            self.canvas1.draw()
            empty_label1.place(relx = 0.15, rely = 0.5, relwidth =
0.2, relheight=0.1)

        if(len(G2.nodes()) !=0):
            empty_label2.place_forget()
            pos_2 = hierarchy_pos(G2.reverse(),0)
            nx.draw(G2, ax=self.a2, pos=pos_2, with_labels=True)
            self.canvas2.draw()

        else:
            self.canvas2.draw()
            empty_label2.place(relx = 0.65, rely = 0.5, relwidth =
0.2, relheight=0.1)


app = SeaofFrames()
app.minsize(width = WIDTH, height = HEIGHT)
app.title("Group 2 Sem 1 2019")

app.mainloop()
```

Test Cases

In this section of the report, we will be exploring the different test cases that we have created to ensure that our algorithm is working as expected. The test cases cover a broad series of changes in the queries, for example changes in the table schema, changes in projection, etc. We have also experimented our algorithm on cases where a combination of changes are working as expected.

Instructions to run test cases

1. Ensure that you have pytest installed.
2. Navigate to the test directory.
3. Update your database information in the global variables: host, DB_NAME, user, password, port in the “test_difference_in_natural_language.py”.
4. Run the following command to test whether the test cases are working.

```
python -m pytest test_difference_in_natural_language.py
```

5. If everything works correctly as intended, the following output should be received in the command line.

```
===============================================================
                 8 passed, 1 warnings in 0.41s
================================================================
```

Test Case 1

The following test case explores the change in the table schema. The assertion ensures that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def test_node_changes_for_table_changes_are_reflected_correctly():
    old_query = "select * from lineitem;"
    new_query = "select * from customer;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "The node with
node label 0 of type Seq Scan has the following changes: relation name
lineitem has changed into relation name customer, alias has changed
```

```
from lineitem into customer and output name has changed from lineitem
into customer."
```

Test Case 2

The following test case explores the change in the filter. We explored changes where there are no filters vs when both queries have filters involved. The assertions ensure that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def test_node_changes_for_filter_changes_are_reflected_correctly():
    old_query = "select * from customer;"
    new_query = "select * from customer where c_nationkey = 15;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "The node with
node label 0 of type Seq Scan has the following changes: table filter
has changed from None into (c_nationkey = 15)."
```

Test Case 3

The following test case explores the combination of the first two test cases. The assertion ensures that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def
test_node_changes_for_table_changes_and_filter_changes_are_reflected_c
orrectly():
    old_query = "select * from lineitem;"
    new_query = "select * from customer where c_nationkey > 0;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "The node with
node label 0 of type Seq Scan has the following changes: relation name
lineitem has changed into relation name customer, alias has changed
from lineitem into customer, table filter has changed from None into
```

```
(c_nationkey > 0) and output name has changed from lineitem into
customer."
```

Test Case 4

The following test case explores the changes in the projection. We also explored a test case where the projection and filter were adjusted. The assertion ensures that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def test_projection_changes_are_reflected_correctly():
    old_query = "select * from customer;"
    new_query = "select (c_nationkey, c_name) from customer;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "Query
projections has changed from ['*'] in the old query to ['c_name',
'c_nationkey'] in the new query."
######################################################################
    old_query = "select * from customer;"
    new_query = "select (c_nationkey, c_name) from customer where
c_nationkey = 15;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "Query
projections has changed from ['*'] in the old query to ['c_name',
'c_nationkey'] in the new query.The node with node label 0 of type Seq
Scan has the following changes: table filter has changed from None
into (c_nationkey = 15)."
```

Test Case 5

The following test case explores the changes when new nodes are added to the graph. The assertion ensures that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def test_insertions_are_reflected_correctly():
    old_query = "select * from lineitem;"
    new_query = "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_shipdate <= '1998-09-16'
group by l_returnflag, l_linestatus order by l_returnflag,
l_linestatus;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "Query
projections has changed from ['*'] in the old query to
['avg(l_discount) as avg_disc', 'avg(l_extendedprice) as avg_price',
'avg(l_quantity) as avg_qty', 'count(*) as count_order',
'l_linestatus', 'l_returnflag', 'sum(l_extendedprice * (1 -
l_discount) * (1 + l_tax)) as sum_charge', 'sum(l_extendedprice * (1 -
l_discount)) as sum_disc_price', 'sum(l_extendedprice) as
sum_base_price', 'sum(l_quantity) as sum_qty'] in the new query.The
node with node label 0 of type Seq Scan gets mapped to new node label
4 of type Seq Scan and table filter has changed from None into
(l_shipdate <= '1998-09-16'::date).Aggregate (3,New), Sort (2,New),
Gather Merge (1,New) and Aggregate (0,New) gets inserted after Seq
Scan (4,New)."
```

Test Case 6

The following test case explores the changes when nodes are deleted from the graph. The assertion ensures that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def test_deletions_are_reflected_correctly():
    old_query = "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_shipdate <= '1998-09-16'
group by l_returnflag, l_linestatus order by l_returnflag,
l_linestatus;"
    new_query = "select * from lineitem;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "Query
projections has changed from ['avg(l_discount) as avg_disc',
'avg(l_extendedprice) as avg_price', 'avg(l_quantity) as avg_qty',
'count(*) as count_order', 'l_linestatus', 'l_returnflag',
'sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge',
'sum(l_extendedprice * (1 - l_discount)) as sum_disc_price',
'sum(l_extendedprice) as sum_base_price', 'sum(l_quantity) as
sum_qty'] in the old query to ['*'] in the new query.The node with
node label 4 of type Seq Scan gets mapped to new node label 0 of type
Seq Scan and table filter has changed from (l_shipdate <= '1998-09-
16'::date) into None.Aggregate (3,Old), Sort (2,Old), Gather Merge
(1,Old) and Aggregate (0,Old) that is after Seq Scan (4,Old) gets
deleted."
```

Test Case 7

The following test case explores the changes when new nodes containing merge are added to the graph. This test case explores whether the algorithm is able to handle cases when query plan contains a join. The assertion ensures that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def test_subqueries_are_reflected_correctly():
    old_query = "select * from orders;"
    new_query = "select o_orderpriority, count(*) as order_count from
orders as o where o_orderdate >= '1996-05-01' and o_orderdate < '1996-
08-01' and exists (select * from lineitem where l_orderkey =
o.o_orderkey and l_commitdate < l_receiptdate) group by
o_orderpriority order by o_orderpriority;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "Query
projections has changed from ['*'] in the old query to ['count(*) as
order_count', 'o_orderpriority'] in the new query.The node with node
label 0 of type Seq Scan gets mapped to new node label 5 of type Seq
```

```
Scan, alias has changed from orders into o and table filter has
changed from None into ((o_orderdate >= '1996-05-01'::date) AND
(o_orderdate < '1996-08-01'::date)).Index Scan (6,New) gets inserted
before Nested Loop (4,New).Nested Loop (4,New) joins Seq Scan (5,New)
and Index Scan (6,New) and gets inserted.Sort (3,New), Aggregate
(2,New), Gather Merge (1,New) and Aggregate (0,New) gets inserted
after Nested Loop (4,New)."
######################################################################
    old_query = "select o_orderpriority, count(*) as order_count from
orders as o where o_orderdate >= '1996-05-01' and o_orderdate < '1996-
08-01' and exists (select * from lineitem where l_orderkey =
o.o_orderkey and l_commitdate < l_receiptdate) group by
o_orderpriority order by o_orderpriority;"
    new_query = "select * from orders;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "Query
projections has changed from ['count(*) as order_count',
'o_orderpriority'] in the old query to ['*'] in the new query.The node
with node label 5 of type Seq Scan gets mapped to new node label 0 of
type Seq Scan, alias has changed from o into orders and table filter
has changed from ((o_orderdate >= '1996-05-01'::date) AND (o_orderdate
< '1996-08-01'::date)) into None.Index Scan (6,Old) that is before
Nested Loop (4,Old) gets deleted.Nested Loop (4,Old), Sort (3,Old),
Aggregate (2,Old), Gather Merge (1,Old) and Aggregate (0,Old) that is
after Seq Scan (5,Old) gets deleted."

######################################################################
######################################################################
##########################################
    old_query = "select l_orderkey, sum(l_extendedprice * (1 -
l_discount)) as revenue, o_orderdate, o_shippriority from orders,
customer, lineitem where c_mktsegment = 'BUILDING' and c_custkey =
o_custkey and l_orderkey = o_orderkey and o_orderdate < '1995-03-22'
and l_shipdate > '1995-03-22' group by l_orderkey, o_orderdate,
o_shippriority order by revenue desc, o_orderdate limit 10;"
    new_query = "select * from orders;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "Query
projections has changed from ['l_orderkey', 'o_orderdate',
'o_shippriority', 'sum(l_extendedprice * (1 - l_discount)) as
revenue'] in the old query to ['*'] in the new query.The node with
node label 9 of type Seq Scan gets mapped to new node label 0 of type
Seq Scan and table filter has changed from (o_orderdate < '1995-03-
22'::date) into None.Seq Scan (11,Old) and Hash (10,Old) that is
before Hash Join (7,Old) gets deleted.Hash Join (7,Old) that is in
between Seq Scan (9,Old) and Nested Loop (6,Old) gets deleted.Index
Scan (8,Old) that is before Nested Loop (6,Old) gets deleted.Nested
Loop (6,Old), Sort (5,Old), Aggregate (4,Old), Gather Merge (3,Old),
Aggregate (2,Old), Sort (1,Old) and Limit (0,Old) that is after Hash
Join (7,Old) gets deleted."
```

Test Case 8

The following test case explores the changes when query contains subplan nodes. The assertion ensures that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def test_subplans_are_reflected_correctly():
    old_query = "select p_brand, p_type, p_size, count(distinct
ps_suppkey) as supplier_cnt from partsupp, part where p_partkey =
ps_partkey and p_brand <> 'Brand#34' and p_type not like 'ECONOMY
BRUSHED%' and p_size in (22, 14, 27, 49, 21, 33, 35, 28) and
partsupp.ps_suppkey not in ( select s_suppkey from supplier where
s_comment like '%Customer%Complaints%') group by p_brand, p_type,
p_size order by supplier_cnt desc, p_brand, p_type, p_size;"
    new_query = "select * from orders;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "Query
projections has changed from ['count(distinct ps_suppkey) as
supplier_cnt', 'p_brand', 'p_size', 'p_type'] in the old query to
['*'] in the new query.The node with node label 7 of type Seq Scan
gets mapped to new node label 0 of type Seq Scan, relation name
supplier has changed into relation name orders, alias has changed from
supplier into orders, table filter has changed from ((s_comment)::text
~~ '%Customer%Complaints%'::text) into None, subplan name has changed
```

```
from SubPlan 1 into None and output name has changed from supplier
into orders.Seq Scan (5,Old) that is in between Seq Scan (7,Old) and
Hash Join (4,Old) gets deleted.Seq Scan (8,Old) and Hash (6,Old) that
is before Hash Join (4,Old) gets deleted.Hash Join (4,Old), Gather
(3,Old), Sort (2,Old), Aggregate (1,Old) and Sort (0,Old) that is
after Seq Scan (5,Old) gets deleted."
```

Test Case 9

The following test case explores the changes when query contains a branch of a join node gets deleted/inserted. The assertion ensures that the algorithm works as intended by checking on whether the output is identical as our intended output.

```
def test_some_sample_queries_from_neuron():
    old_query = "select l_orderkey, sum(l_extendedprice * (1 -
l_discount)) as revenue, o_orderdate, o_shippriority from orders,
lineitem where l_orderkey = o_orderkey and o_orderdate < date '1995-
03-21' and l_shipdate > date '1995-03-21' group by l_orderkey,
o_orderdate, o_shippriority order by revenue desc, o_orderdate limit
10;"
    new_query = "select l_orderkey, sum(l_extendedprice * (1 -
l_discount)) as revenue, o_orderdate, o_shippriority from customer,
orders, lineitem where c_mktsegment = 'HOUSEHOLD' and c_custkey =
o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-
21' and l_shipdate > date '1995-03-21' group by l_orderkey,
o_orderdate, o_shippriority order by revenue desc, o_orderdate limit
10;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "The node with
node label 6 of type Hash Join gets mapped to new node label 7 of type
Hash Join and hash condition has changed from (lineitem.l_orderkey =
orders.o_orderkey) into (orders.o_custkey = customer.c_custkey).The
node with node label 7 of type Seq Scan gets mapped to new node label
11 of type Seq Scan, relation name lineitem has changed into relation
name customer, alias has changed from lineitem into customer, table
filter has changed from (l_shipdate > '1995-03-21'::date) into
(c_mktsegment = 'HOUSEHOLD'::bpchar) and output name has changed from
lineitem into customer.The node with node label 8 of type Hash gets
mapped to new node label 10 of type Hash.Index Scan (8,New) gets
inserted before Nested Loop (6,New).Nested Loop (6,New) joins Hash
Join (7,New) and Index Scan (8,New) and gets inserted."

######################################################################
    old_query = "select l_orderkey, sum(l_extendedprice * (1 -
l_discount)) as revenue, o_orderdate, o_shippriority from customer,
orders, lineitem where c_mktsegment = 'HOUSEHOLD' and c_custkey =
o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-
21' and l_shipdate > date '1995-03-21' group by l_orderkey,
o_orderdate, o_shippriority order by revenue desc, o_orderdate limit
10;"
```

```
    new_query = "select l_orderkey, sum(l_extendedprice * (1 -
l_discount)) as revenue, o_orderdate, o_shippriority from orders,
lineitem where l_orderkey = o_orderkey and o_orderdate < date '1995-
03-21' and l_shipdate > date '1995-03-21' group by l_orderkey,
o_orderdate, o_shippriority order by revenue desc, o_orderdate limit
10;"
    old_query_plan, success_1 =
postgres_wrapper.get_query_plan_of_query(old_query)
    new_query_plan, success_2 =
postgres_wrapper.get_query_plan_of_query(new_query)
    assert success_1 == True
    assert success_2 == True
    natural_language_string =
find_difference_between_two_query_plans(old_query, old_query_plan,
new_query, new_query_plan)
    assert natural_language_string.replace("\n", "") == "The node with
node label 7 of type Hash Join gets mapped to new node label 6 of type
Hash Join and hash condition has changed from (orders.o_custkey =
customer.c_custkey) into (lineitem.l_orderkey = orders.o_orderkey).The
node with node label 10 of type Hash gets mapped to new node label 8
of type Hash.The node with node label 11 of type Seq Scan gets mapped
to new node label 7 of type Seq Scan, relation name customer has
changed into relation name lineitem, alias has changed from customer
into lineitem, table filter has changed from (c_mktsegment =
'HOUSEHOLD'::bpchar) into (l_shipdate > '1995-03-21'::date) and output
name has changed from customer into lineitem.Index Scan (8,Old) that
is before Nested Loop (6,Old) gets deleted.Nested Loop (6,Old) that is
in between Hash Join (7,Old) and Sort (5,Old) gets deleted."
```