NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ4031 DATABASE SYSTEM PRINCIPLES

GROUP PROJECT 2 REPORT

Group Members	Matriculation Number
Ma Xiao	U1722964K
Tang Jiayun	U1621004H
Shi Yuxin	U1621078D

1. Project Methodology

In the course project 2, we are tasked to develop an application to generate a natural language description of the difference between the query execution plans of two related queries. The inputs and outputs of the application are shown in the table below:

input	output
two related queries in SQL language	➔ descriptions of steps taken to execute the two queries in natural language ➔ visualisations of the execution plans of the two queries in a tree structure ➔ description of the difference in the execution plans of the two queries and the reason that accounts for the difference in natural language

With that, the implementation of our application consists of five sequential steps:

Step 1: Generate execution plans and tree visualisation
Step 2: Compare the difference between the nodes in the two trees
Step 3: Generate natural language description on the difference
Step 4: Match the difference with its possible reasons
Step 5: Present the results in the graphical user interface

Each part would be explained in detail in chapter 2: Project Implementation.

2.Project Implementation

2.1 Generate execution plans and tree visualisation

First of all we generated execution plans in binary tree data structure and then visualized the plans. To achieve this, we input queries into Postgresql and used the built-in **explain clause** generate a json file of the query execution plan. We then passed the json file as a parameter to the **pare_json(json_obj)** method adopted from **vocalizer.py** (the assistant file on NTULearn) to generate natural language description on the steps taken to execute a query.

To further visualise tree structure, we defined a method which takes in a tree’s root node and prints out a physical tree using the node.node_type (eg. seq scan, sort).

During implementation, we also applied graph simplification technique by subtracting the children’s **actual_time** attribute from the root’s **actual_time** attribute. After this, we can get the execution time consumed by the node itself. Generating natural language description is achieved by utilizing the function get_text().

One important thing to note is that, on top of the previous attribute declaration of the node, we declared another attribute named 'description' to node class. This attribute records the natural language description of the steps that are undergoing when this node is under execution. This attribute will be reused inside section 2.3, where we are generating the natural language description of the difference between the two QEPs.

2.2 Compare the difference of nodes in two trees

The rule for node comparison is that, if two nodes have exactly the same node_type and number of children, the two nodes are considered 'same'. Otherwise, the two nodes are considered different and the description in the natural language would be generated.

To compare the nodes of two trees, we have applied preorder traversal. The root node will be checked first, followed by its left children and right children. The special rule to mention is that, when we meet node of node_type **'Hash'**, **'Gather'** or **'Sort'** in one tree, we skip this node as it is trivial and use its leftmost child to compare with the node at the same level of this node. Following examples would be illustrating the search logic:

<table>
<tr><th colspan="2">example one</th></tr>
<tr><th>tree of old query</th><th>tree of new query</th></tr>
<tr><td>--Hash Join (node o1)
  |--Seq Scan (node o2)
  |--Hash (node o3)
    |--Seq Scan (node o4)</td><td>--Nested Loop (node n1)
  |--Seq Scan (node n2)
  |--Hash (node n3)
    |--Seq Scan (node n4)</td></tr>
</table>

In example one, the traversal would be carried out in the following sequence:

comparison 1: node o1 VS node n1 -> difference detected

comparison 2: node o2 VS node n2 -> same node

(node o3 and node n3 are skipped as they are trivial)

comparison 3: node o4 VS node n4 -> difference detected

example two	
tree of old query	tree of new query

|--Gather Merge (node o1)
  |--Sort (node o2)
    |--Hash Join (node o3)
      |--Seq Scan (node o4)
      |--Hash (node o5)
        |--Seq Scan (node o6)
	|--Sort (node n1)
	  |--Hash Join (node n2)
	    |--Index Scan (node n3)
	    |--Hash (node n4)
	      |--Seq Scan (node n5)

In example two, the traversal would be carried out in the following sequence:

#node o1 is skipped as it is trivial

comparison 1: node o2 VS node n1 -> same node

comparison 2: node o3 VS node n2 -> same node

comparison 3: node o4 VS node n3 -> difference detected

comparison 4: node o5 VS node n4 -> same node

comparison 5: node o6 VS node n5 -> same node

2.3 Generate natural language description on the difference

During generation execution plans explained in chapter 2.1, each node is assigned with an attribute '*node.description*' which is to store the description of operation taken at the node. Hence whenever difference is detected during preorder tree traversal, the *node.description* of the pair of different nodes would be retrieved and combined to form a complete sentence. The sentence is further processed by removing some action verbs (e.g. 'perform') to make the sentences concise and grammatically correct.

2.4 Match the difference of query plans with its possible reasons

In this project, the reasons causing the changes between P1 and P2 after modifying Q to Q’ are required to be further elaborated. In our project, we focus more on the changes due to modifications in the WHERE clause. We keep the input and output attributes of each query pair the same, i.e. not changing the FROM clause and SELECT clause. The difference of query plans consists of two parts. One is the difference in the SCAN operators, and the other one is the difference in the JOIN operators.

For SCAN operations, we mainly focus on “Index Scan” and “Seq Scan”(Sequential Scan). After running through various experiments, we realise that index scan usually happened when we extract the index keys from tables, and it usually happened when the table size is now that big. Seq Scan are applicable on the tables with large size. Hence, when Scan operation in query plans changes from Index Scan to Seq Scan, usually either the table size is too big or the key of the table is not extracted. Hence, when switches between “Index Scan” and “Seq Scan” happen, we look into a few node information (shown in the figure below). For example, “Index Name” indicates whether the table key is extracted and “Actual Rows” indicates the number of rows remaining after selection. We utilise this information to explain the difference between query plans.

```
reason_scan_dict = {
    "Index Scan": ["Index Name", "Relation Name", "Actual Rows", "Output", "Index Cond"],
    "Seq Scan": ["Relation Name", "Actual Rows", "Output", "Filter"],
    "Merge Join":["Actual Rows", "Output", "Merge Cond"],
    "Hash Join":["Actual Rows", "Output", "Hash Cond"],
    "Nested Loop": ["Actual Rows"]
}
```

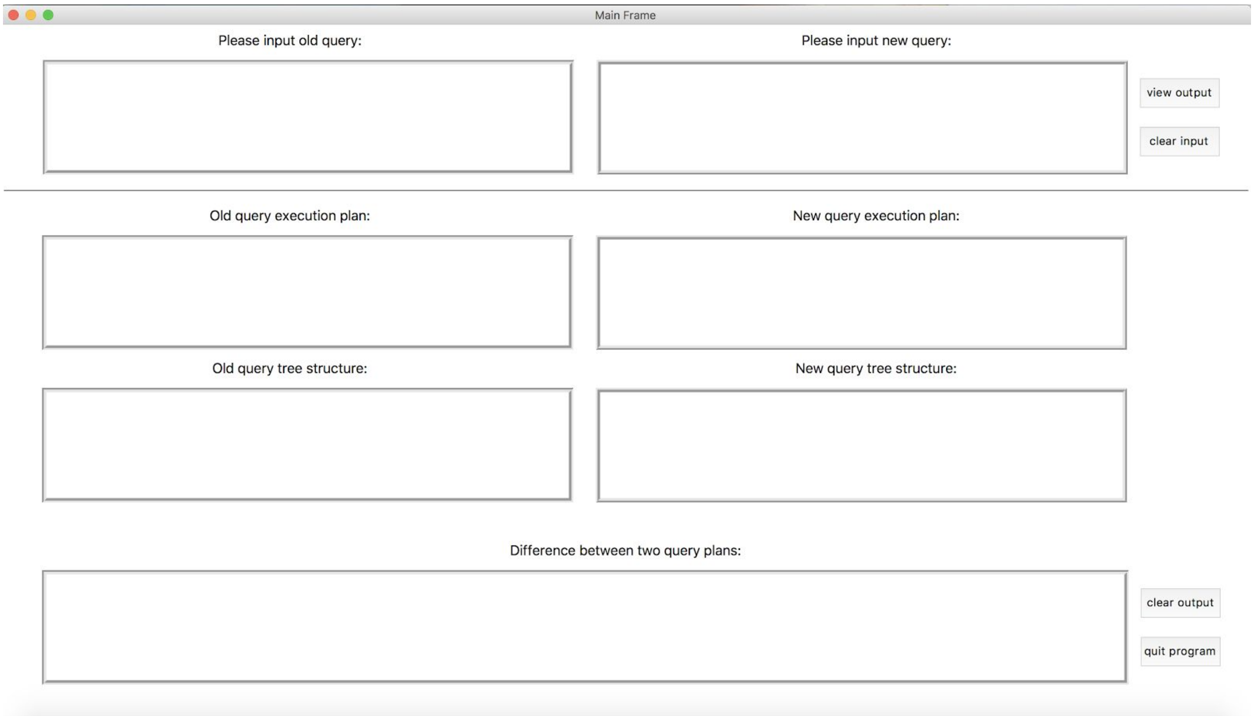
For JOIN operations, we mainly focus on “Nested Loop” (Nested loop join), “Merge Join” and “Hash Join”. 1. Nested loop joins are performed when one or more of the sides of the join has few rows. It is used as the only option if the join condition does not use the equality operator. 2. Merge join is better when we use the equality operator. Both sides of the join can be sorted on joining conditions efficiently. 3. Hash join is used when both sides of the join operator are large and the length of each hash bucket should be able to fit into the memory. For the changes in JOIN operations, we focus on looking at “Actual Rows” to compare the data size remaining after modifying Q to Q’. “Merge Cond” and “Hash Cond” which both records the join condition of the join operation is also put into consideration to explain the differences.

2.5 Present the results in the graphical user interface

The input queries will be fetched from the user interface and executed on Postgresql. To connect with the application user interface to the Postgresql server, **psycopg2** module is used. The function **psycopg2.connect()** is used for the connection.

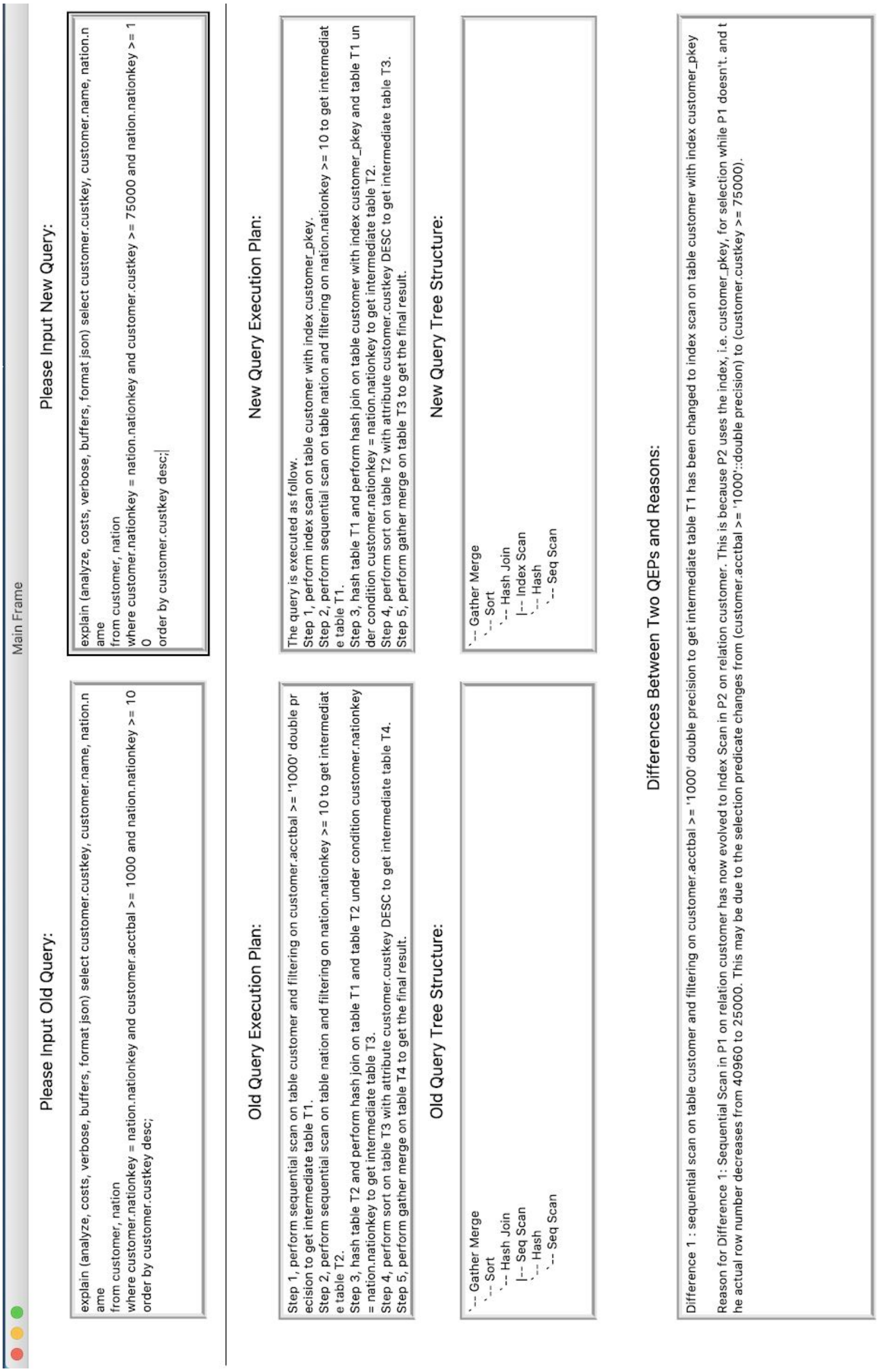
After generating the query result, we can get the corresponding analysis using steps mentioned in chapter 2.1-2.4 and the analysis results are presented in a graphical user interface.

The graphical user interface is implemented with tkinter package in Python library. The input window at the top and output window at the bottom of the window is segregated by a line. Once a user has input queries and pressed 'view result' button, the results would be displayed in the output segment.

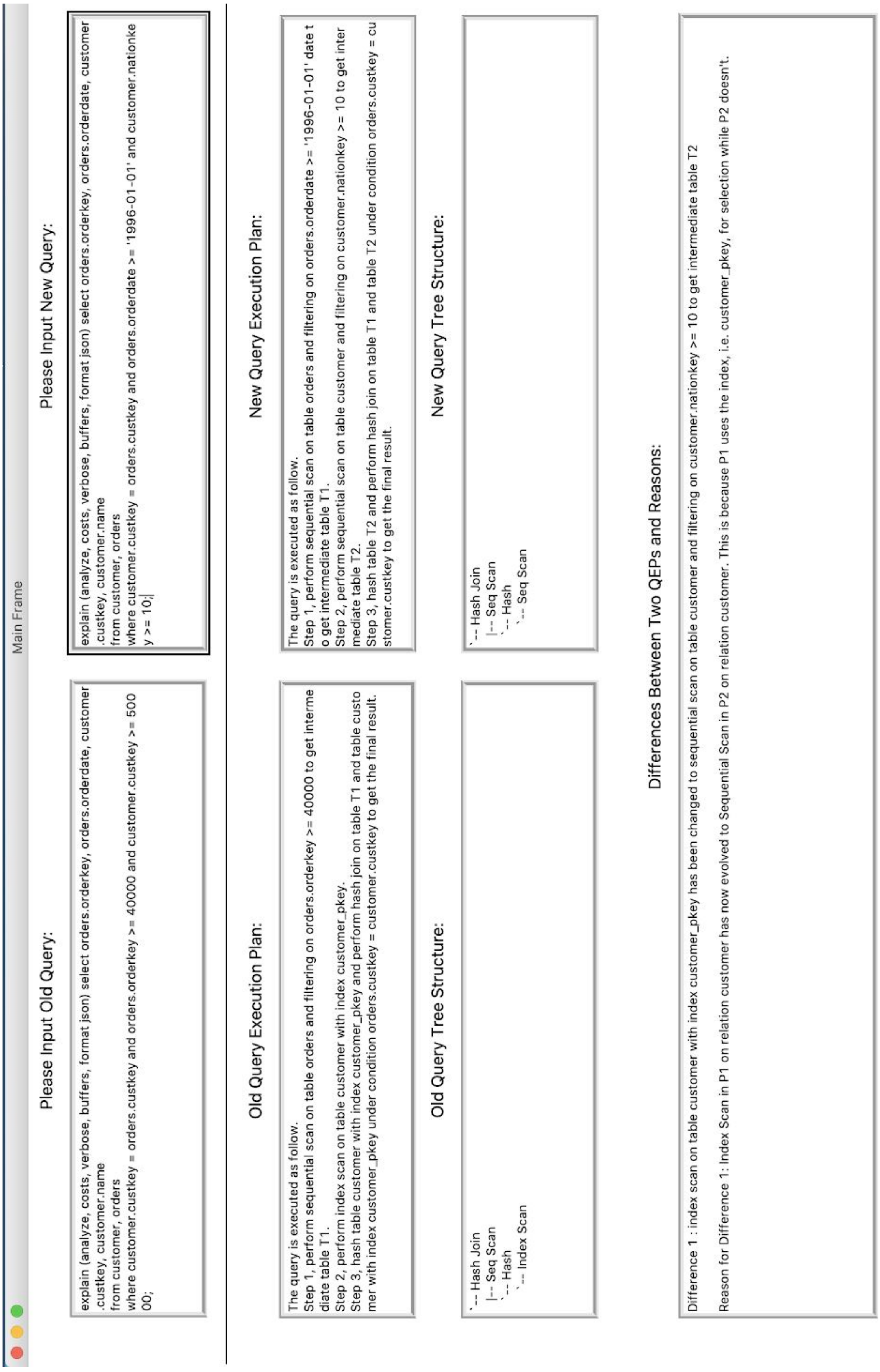


3. Sample Queries and Results

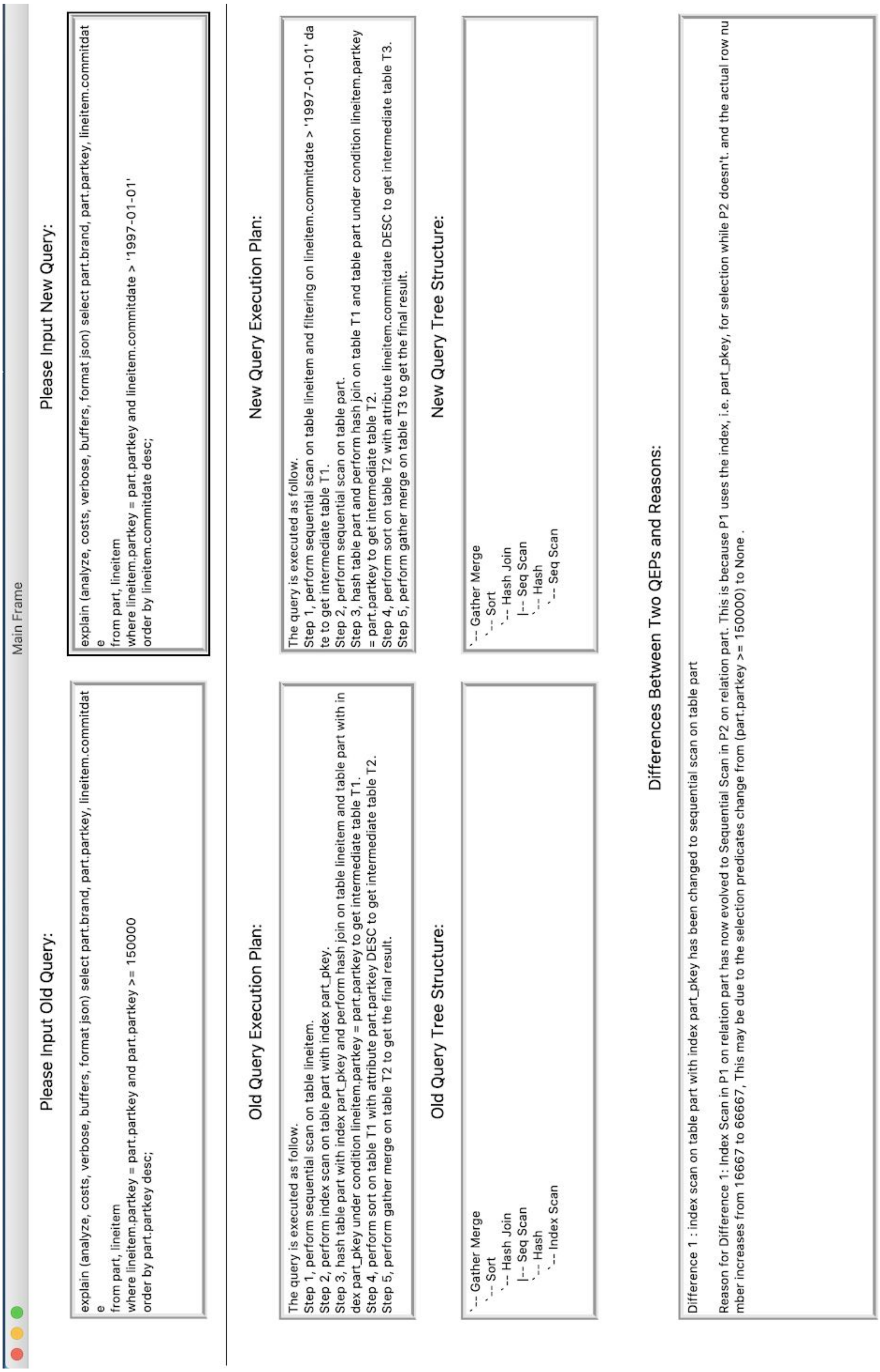
3.1 Query 1A and Query 1B



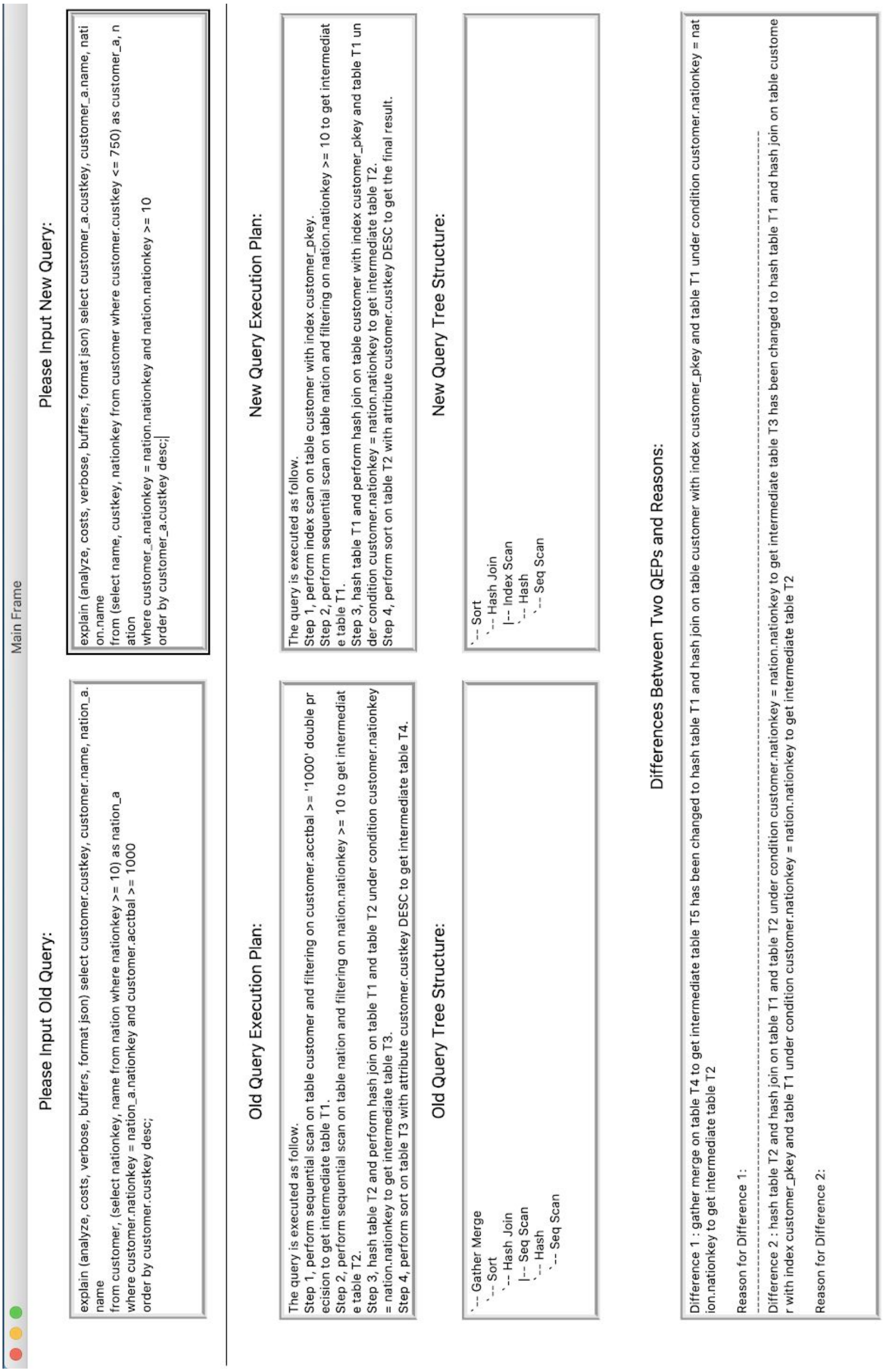
3.2 Query 2A and Query 2B



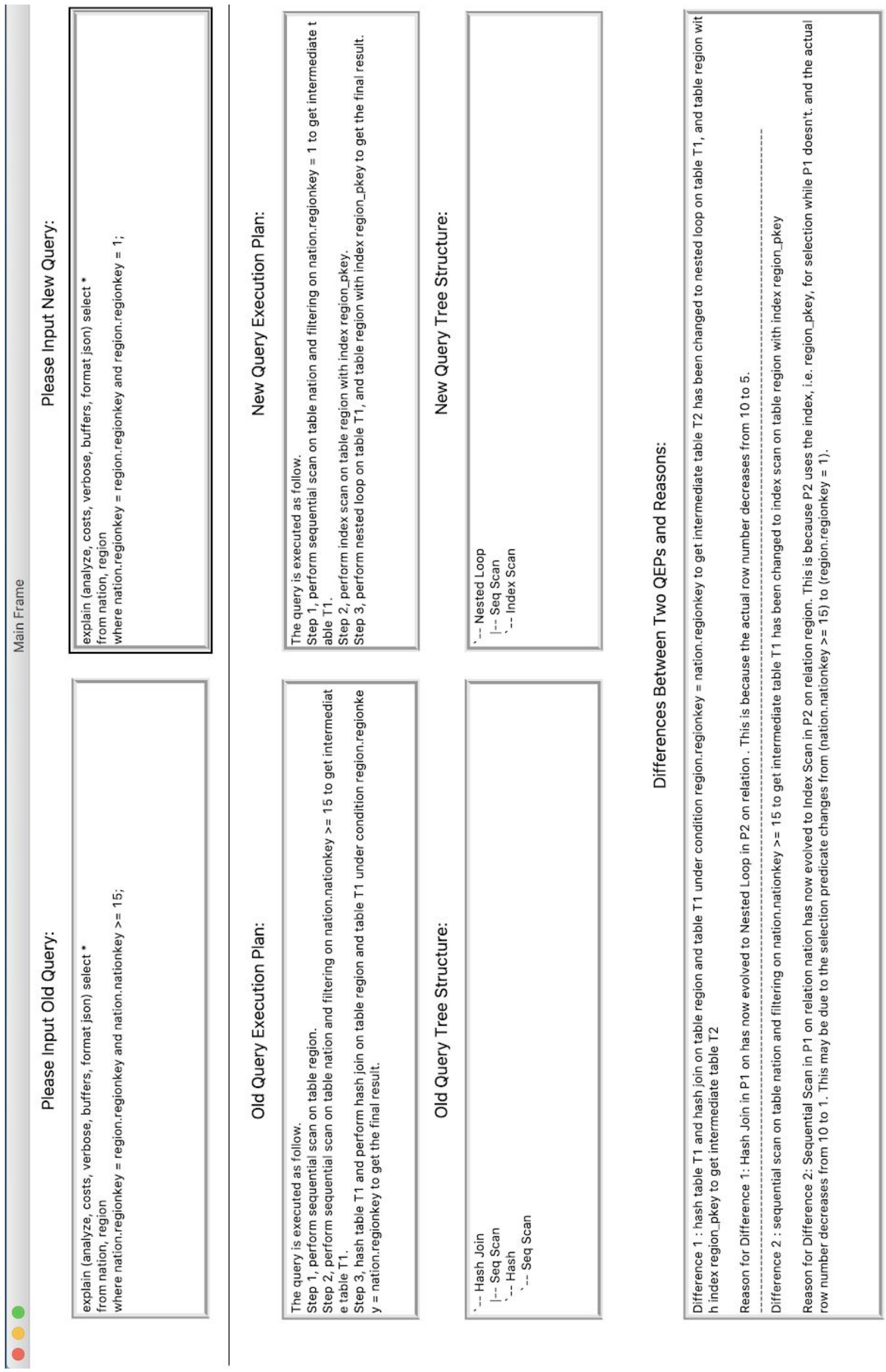
3.3 Query 3A and Query 3B



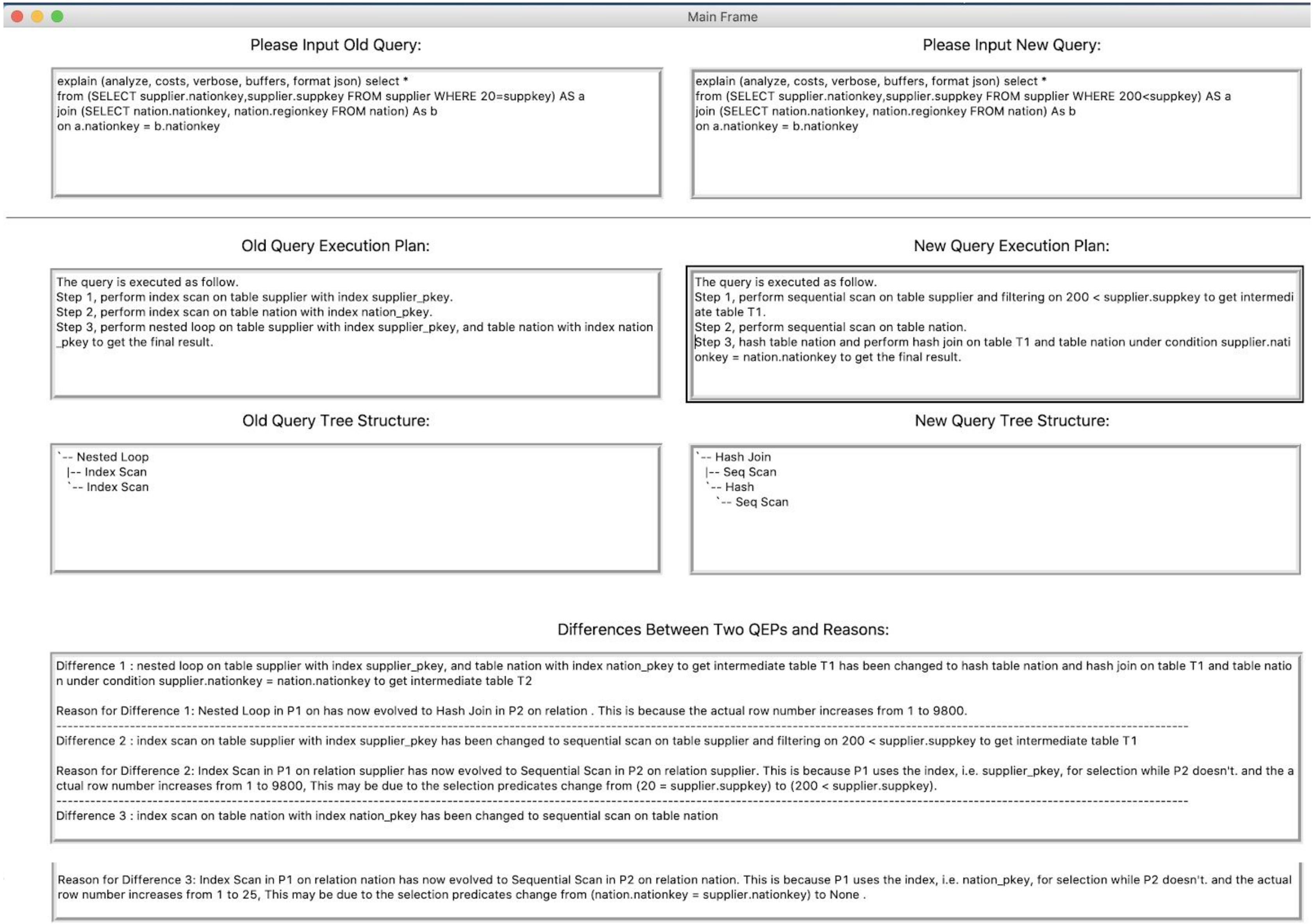
3.4 Query 4A and Query 4B



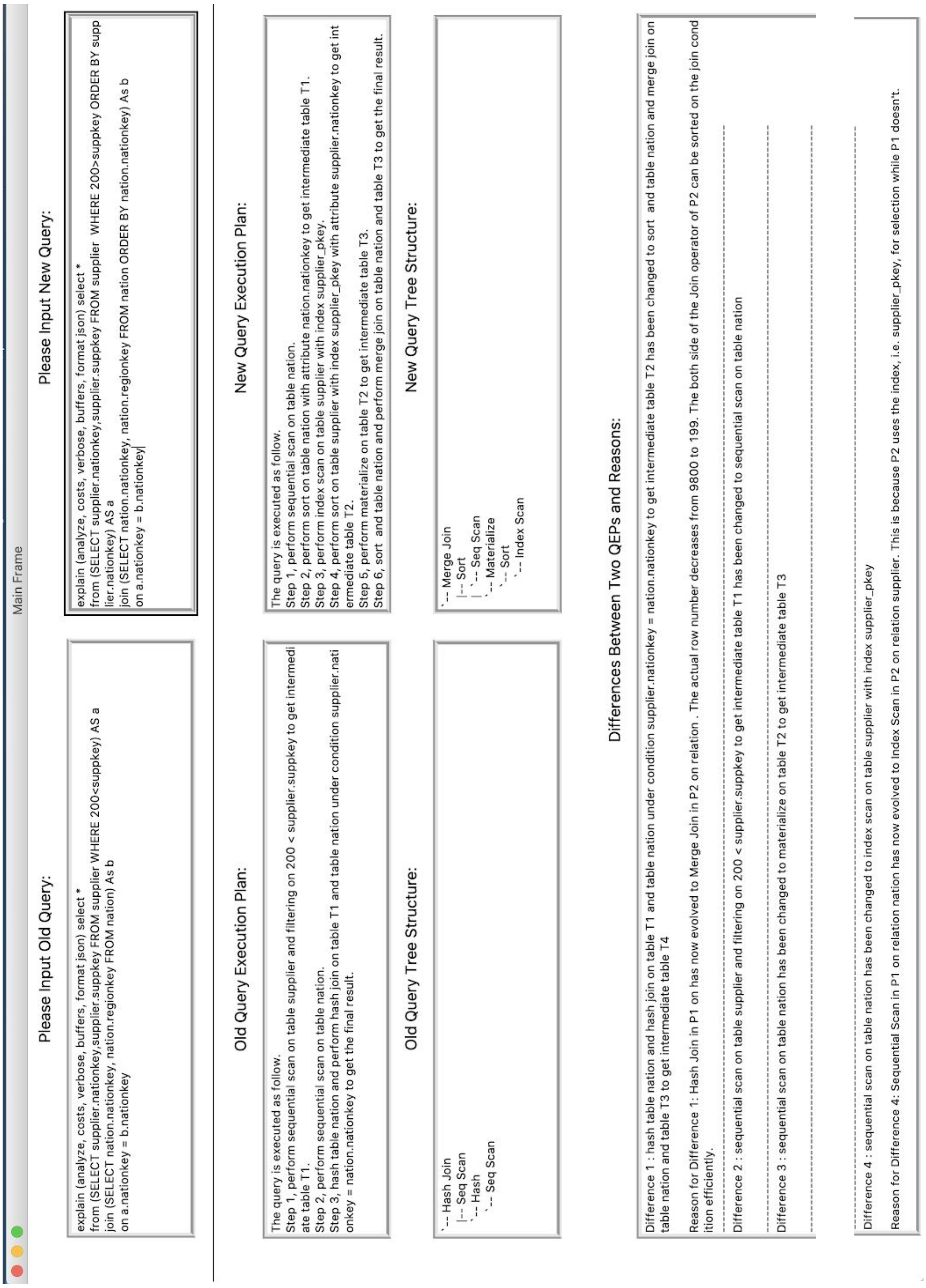
3.5 Query 5A and Query 5B



3.6 Query 6A and Query 6B



3.7 Query 7A and Query 7B



4. Appendix: Code

app.py (main application)

```
import sys
import json
import tkinter as tk
from tkinter import *
from tkinter.font import Font
from tkinter import messagebox
from query_description import *
from pyconnect import DBConnection
import argparse


class App(object):

    def __init__(self, parent, host, port, database, user, password):
        self.root = parent
        self.root.title("Main Frame")
        self.frm_input_text = tk.Frame(self.root)
        self.frm_input_text.pack()
        self.frm_input = tk.Frame(self.root)
        self.frm_input.pack()
        self.frm_line = tk.Frame(self.root)
        self.frm_line.pack()
        canvas = Canvas(self.frm_line, width=2000, height=20)
        canvas.create_line(0, 15, 2000, 15)
        canvas.pack()
        output_font = Font(family=None, size=15)
        self.frm_nlp_text = tk.Frame(self.root)
        self.frm_nlp_text.pack()
        self.frm_nlp = tk.Frame(self.root)
        self.frm_nlp.pack()
        self.frm_tree_text = tk.Frame(self.root)
        self.frm_tree_text.pack()
        self.frm_tree = tk.Frame(self.root)
        self.frm_tree.pack()
        self.frm_diff_text = tk.Frame(self.root)
        self.frm_diff_text.pack()
        self.frm_diff = tk.Frame(self.root)
        self.frm_diff.pack()

        self.frm_input_t = tk.Frame(self.frm_input)
        self.frm_input_t.pack(side=LEFT)
        self.frm_input_btt = tk.Frame(self.frm_input)
        self.frm_input_btt.pack(side=RIGHT)

        self.frm_nlp_t = tk.Frame(self.frm_nlp)
        self.frm_nlp_t.pack(side=LEFT)
```

```
        self.frm_nlp_btt = tk.Frame(self.frm_nlp)
        self.frm_nlp_btt.pack(side=RIGHT)

        self.frm_tree_t = tk.Frame(self.frm_tree)
        self.frm_tree_t.pack(side=LEFT)
        self.frm_tree_btt = tk.Frame(self.frm_tree)
        self.frm_tree_btt.pack(side=RIGHT)

        self.frm_tree_t = tk.Frame(self.frm_tree)
        self.frm_tree_t.pack(side=LEFT)
        self.frm_tree_btt = tk.Frame(self.frm_tree)
        self.frm_tree_btt.pack(side=RIGHT)

        self.frm_diff_t = tk.Frame(self.frm_diff)
        self.frm_diff_t.pack(side=LEFT)
        self.frm_diff_btt = tk.Frame(self.frm_diff)
        self.frm_diff_btt.pack(side=RIGHT)

        self.input_text1 = tk.Label(
            self.frm_input_text, text='Please Input Old Query:', font=(None, 16),
width=60)
        self.input_text1.pack(side=LEFT, pady=5)
        self.input_text2 = tk.Label(
            self.frm_input_text, text='Please Input New Query:                     ',
font=(None, 16), width=75)
        self.input_text2.pack(side=RIGHT, pady=5)

        self.input1 = tk.Text(self.frm_input_t, relief=GROOVE,
                              width=75, height=8, borderwidth=5, font=(None, 12))
        self.input1.pack(side=LEFT, padx=10)
        self.input2 = tk.Text(self.frm_input_t, relief=RIDGE,
                              width=75, height=8, borderwidth=5, font=(None, 12))
        self.input2.pack(side=RIGHT, padx=10)

        self.view = tk.Button(self.frm_input_btt, text="view output",
                              width=10, height=2, command=self.retrieve_input)
        self.view.pack(pady=10)

        self.clear = tk.Button(self.frm_input_btt, text="clear input",
                               width=10, height=2, command=self.clear_input)
        self.clear.pack(pady=10)

        self.nlp_text1 = tk.Label(
            self.frm_nlp_text, text='Old Query Execution Plan:', font=(None, 16),
width=60)
        self.nlp_text1.pack(side=LEFT, pady=5)
        self.nlp_text2 = tk.Label(
            self.frm_nlp_text, text='New Query Execution Plan:                     ',
font=(None, 16), width=75)
        self.nlp_text2.pack(side=RIGHT, pady=5)

        self.nlp1 = tk.Text(self.frm_nlp_t, relief=GROOVE, width=75,
                            height=8, borderwidth=5, font=(None, 12), state='disabled')
```

```
        self.nlp1.pack(side=LEFT, padx=10)
        self.nlp2 = tk.Text(self.frm_nlp_t, relief=RIDGE, width=75,
                            height=8, borderwidth=5, font=(None, 12), state='disabled')
        self.nlp2.pack(side=RIGHT, padx=10)
        self.placeholder1 = tk.Label(self.frm_nlp_btt, width=10)
        self.placeholder1.pack()

        self.tree_text1 = tk.Label(
            self.frm_tree_text, text='Old Query Tree Structure:', font=(None, 16),
width=60)
        self.tree_text1.pack(side=LEFT, pady=5)
        self.tree_text2 = tk.Label(
            self.frm_tree_text, text='New Query Tree Structure:                     ',
font=(None, 16), width=75)
        self.tree_text2.pack(side=RIGHT, pady=5)

        self.tree1 = tk.Text(self.frm_tree_t, relief=GROOVE, width=75,
                             height=8, borderwidth=5, font=(None, 12), state='disabled')
        self.tree1.pack(side=LEFT, padx=10)
        self.tree2 = tk.Text(self.frm_tree_t, relief=RIDGE, width=75,
                             height=8, borderwidth=5, font=(None, 12), state='disabled')
        self.tree2.pack(side=RIGHT, padx=10)
        self.placeholder2 = tk.Label(self.frm_tree_btt, width=10)
        self.placeholder2.pack()

        self.placeholder3 = tk.Label(self.frm_diff_text, width=60)
        self.placeholder3.pack(pady=5)

        self.diff_text = tk.Label(
            self.frm_diff_text, text='Differences Between Two QEPs and Reasons:',
font=(None, 16), width=60)
        self.diff_text.pack(side=LEFT, pady=5)

        self.diff = tk.Text(self.frm_diff_t, relief=GROOVE, width=155,
                            height=15, borderwidth=5, font=(None, 12), state='disabled')
        self.diff.pack(side=LEFT, padx=10)

        self.clear_out = tk.Button(
            self.frm_diff_btt, text="clear output", width=10, height=2,
command=self.clear_output)
        self.clear_out.pack(pady=10)

        self.quit_ = tk.Button(self.frm_diff_btt, text="quit program",
                               width=10, height=2, command=self.quitprogram)
        self.quit_.pack(pady=10)

        self.host = host
        self.port = port
        self.database = database
        self.user = user
        self.password = password

    def retrieve_input(self):
```

```
        global query_old
        global query_new
        global desc
        global result
        query_old = self.input1.get("1.0", END)
        query_new = self.input2.get("1.0", END)
        result_old = self.get_query_result(query_old)
        result_new = self.get_query_result(query_new)
        result_old_obj = json.loads(json.dumps(result_old))
        result_new_obj = json.loads(json.dumps(result_new))
        result_old_nlp = self.get_description(result_old_obj)
        result_new_nlp = self.get_description(result_new_obj)
        result_old_tree = self.get_tree(result_old_obj)
        result_new_tree = self.get_tree(result_new_obj)
        result_diff = self.get_difference(result_old_obj, result_new_obj)
        self.nlp1.configure(state='normal')
        self.nlp2.configure(state='normal')
        self.tree1.configure(state='normal')
        self.tree2.configure(state='normal')
        self.diff.configure(state='normal')

        self.nlp1.delete("1.0", END)
        self.nlp1.insert(END, result_old_nlp)
        self.nlp2.delete("1.0", END)
        self.nlp2.insert(END, result_new_nlp)
        self.tree1.delete("1.0", END)
        self.tree1.insert(END, result_old_tree)
        self.tree2.delete("1.0", END)
        self.tree2.insert(END, result_new_tree)
        self.diff.delete("1.0", END)
        self.diff.insert(END, result_diff)

    def clear_input(self):
        self.input1.delete("1.0", END)
        self.input2.delete("1.0", END)

    def clear_output(self):
        self.nlp1.delete("1.0", END)
        self.nlp2.delete("1.0", END)
        self.tree1.delete("1.0", END)
        self.tree2.delete("1.0", END)
        self.diff.delete("1.0", END)
        self.nlp1.configure(state='disabled')
        self.nlp2.configure(state='disabled')
        self.tree1.configure(state='disabled')
        self.tree2.configure(state='disabled')
        self.diff.configure(state='disabled')

    def get_query_result(self, query):
        # DBConnection takes 5 arguments
        connection = DBConnection(self.host, self.port, self.database, self.user,
self.password)
        result = connection.execute(query)[0][0]
```

```python
        connection.close()
        return result

    def get_description(self, json_obj):
        descriptions = get_text(json_obj)
        result = ""
        for description in descriptions:
            result = result + description + "\n"
        return result

    def get_tree(self, json_obj):
        head = parse_json(json_obj)
        return generate_tree("", head)

    def get_difference(self, json_object_A, json_object_B):
        diff = get_diff(json_object_A, json_object_B)
        return diff

    def quitprogram(self):
        result = messagebox.askokcancel(
            "Quit the game.", "Are you sure?", icon='warning')
        if result == True:
            self.root.destroy()


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--host', help='postgresql connection host')
    parser.add_argument('--port', help='postgresql connection port')
    parser.add_argument('--database', help='the tpch database to connect')
    parser.add_argument('--user', help='db user')
    parser.add_argument('--password', help='db password')
    args = parser.parse_args()
    host = args.host
    port = args.port
    database = args.database
    user = args.user
    password = args.password
    root = tk.Tk()
    app = App(root, host, port, database, user, password)
    root.geometry('1500x1000+0+0')
    root.mainloop()
```

pyconnect.py (Utility module for connecting to Postgresql server)

```
import psycopg2
import json

class DBConnection:
    def __init__(self, host="localhost", port = 5432, database="tpch_db", user="eric",
password=""):
        self.conn = psycopg2.connect(host=host, port=port, database=database, user=user,
password=password)
        self.cur = self.conn.cursor()

    def execute(self,query):
        self.cur.execute(query)
        query_results = self.cur.fetchall()
        return query_results

    def close(self):
        self.cur.close()
        self.conn.close()
```

query_description.py (all utility functions related to query analysis)

```
from __future__ import print_function
import logging
import json
import argparse
import copy
import random
import string
import os
import queue


class Node(object):
    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, actual_rows,
            actual_time,description):
        self.node_type = node_type
        self.children = []
        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.actual_rows = actual_rows
        self.actual_time = actual_time
        self.description = description

    def add_children(self, child):
        self.children.append(child)

    def set_output_name(self, output_name):
        if "T" == output_name[0] and output_name[1:].isdigit():
            self.output_name = int(output_name[1:])
        else:
            self.output_name = output_name

    def get_output_name(self):
        if str(self.output_name).isdigit():
            return "T" + str(self.output_name)
```

```python
        else:
            return self.output_name

    def set_step(self, step):
        self.step = step

    def update_desc(self,desc):
        self.description = desc


def parse_json(json_obj):
    q = queue.Queue()
    q_node = queue.Queue()
    plan = json_obj[0]['Plan']
    q.put(plan)
    q_node.put(None)

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

        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 =
actual_rows = actual_time = description = None
        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 'Actual Rows' in current_plan:
            actual_rows = current_plan['Actual Rows']
```

```
        if 'Actual Total Time' in current_plan:
            actual_time = current_plan['Actual Total Time']
        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']

        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, actual_rows, actual_time, description)

        if "Limit" == current_node.node_type:
            current_node.plan_rows = current_plan['Plan Rows']

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

        if parent_node is not None:
            parent_node.add_children(current_node)
        else:
            head_node = current_node

        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_node)

    return head_node


def simplify_graph(node):
    new_node = copy.deepcopy(node)
    new_node.children = []

    for child in node.children:
        new_child = simplify_graph(child)
        new_node.add_children(new_child)
        new_node.actual_time -= child.actual_time

    if node.node_type in ["Result"]:
```

```
        return node.children[0]

    return new_node


def parse_cond(op_name, conditions, table_subquery_name_pair):
    if isinstance(conditions,str):
        if "::" in conditions:
            return conditions.replace("::", " ")[1:-1]
        return conditions[1:-1]
    cond = ""
    for i in range (len(conditions)):
        cond = cond + conditions[i]
        if (not (i == len(conditions)-1)):
            cond = cond + "and"
    return cond


def to_text(node, skip=False):
    global steps, cur_step, cur_table_name
    increment = True
    # skip the child if merge it with current node
    if node.node_type in ["Unique", "Aggregate"] and len(node.children) == 1 \
            and ("Scan" in node.children[0].node_type or node.children[0].node_type ==
"Sort"):
        children_skip = True
    elif node.node_type == "Bitmap Heap Scan" and node.children[0].node_type == "Bitmap
Index Scan":
        children_skip = True
    else:
        children_skip = False

    # recursive
    for child in node.children:
        if node.node_type == "Aggregate" and len(node.children) > 1 and child.node_type
== "Sort":
            to_text(child, True)
        else:
            to_text(child, children_skip)

    if node.node_type in ["Hash"] or skip:
        return

    step = ""

    # generate natural language for various QEP operators
    if "Join" in node.node_type:
        # special preprocessing for joins
        if node.join_type == "Semi":
            # add the word "Semi" before "Join" into node.node_type
            node_type_list = node.node_type.split()
            node_type_list.insert(-1, node.join_type)
            node.node_type = " ".join(node_type_list)
```

```
        else:
            pass

        if "Hash" in node.node_type:
            step += " and perform " + node.node_type.lower() + " on "
            for i, child in enumerate(node.children):
                if child.node_type == "Hash":
                    child.set_output_name(child.children[0].get_output_name())
                    hashed_table = child.get_output_name()
                if i < len(node.children) - 1:
                    step += ("table " + child.get_output_name())
                else:
                    step += (" and table " + child.get_output_name())
            # combine hash with hash join
            step = "hash table " + hashed_table + step + " under condition " + \
                parse_cond("Hash Cond", node.hash_cond,
                           table_subquery_name_pair)

        elif "Merge" in node.node_type:
            step += "perform " + node.node_type.lower() + " on "
            any_sort = False  # whether sort is performed on any table
            for i, child in enumerate(node.children):
                if child.node_type == "Sort":
                    child.set_output_name(child.children[0].get_output_name())
                    any_sort = True
                if i < len(node.children) - 1:
                    step += ("table " + child.get_output_name())
                else:
                    step += (" and table " + child.get_output_name())
            # combine sort with merge join
            if any_sort:
                sort_step = "sort "
                for child in node.children:
                    if child.node_type == "Sort":
                        if i < len(node.children) - 1:
                            sort_step += ("table " + child.get_output_name())
                        else:
                            sort_step += (" and table " +
                                          child.get_output_name())
                step = sort_step + " and " + step

    elif node.node_type == "Bitmap Heap Scan":
        # combine bitmap heap scan and bitmap index scan
        if "Bitmap Index Scan" in node.children[0].node_type:
            node.children[0].set_output_name(node.relation_name)
            step = " with index condition " + \
                parse_cond("Recheck Cond", node.recheck_cond,
                           table_subquery_name_pair)

        step = "perform bitmap heap scan on table " + \
            node.children[0].get_output_name() + step

    elif "Scan" in node.node_type:
```

```
        if node.node_type == "Seq Scan":
            step += "perform sequential scan on table "
        else:
            step += "perform " + node.node_type.lower() + " on table "

        step += node.get_output_name()

        # if no table filter, remain original table name
        if not node.table_filter:
            increment = False

    elif node.node_type == "Unique":
        # combine unique and sort
        if "Sort" in node.children[0].node_type:
            node.children[0].set_output_name(
                node.children[0].children[0].get_output_name())
            step = "sort " + node.children[0].get_output_name()
            if node.children[0].sort_key:
                step += " with attribute " + \
                    parse_cond(
                        "Sort Key", node.children[0].sort_key, table_subquery_name_pair)
+ " and "

            else:
                step += " and "

        step += "perform unique on table " + node.children[0].get_output_name()

    elif node.node_type == "Aggregate":
        for child in node.children:
            # combine aggregate and sort
            if "Sort" in child.node_type:
                child.set_output_name(child.children[0].get_output_name())
                step = "sort " + child.get_output_name() + " and "
            # combine aggregate with scan
            if "Scan" in child.node_type:
                if child.node_type == "Seq Scan":
                    step = "perform sequential scan on " + child.get_output_name() + "
and "
                else:
                    step = "perform " + child.node_type.lower() + " on " + \
                        child.get_output_name() + " and "

        step += "perform aggregate on table " + \
            node.children[0].get_output_name()
        if len(node.children) == 2:
            step += " and table " + node.children[1].get_output_name()

    elif node.node_type == "Sort":
        step += "perform sort on table " + node.children[0].get_output_name(
        ) + " with attribute " + parse_cond("Sort Key", node.sort_key,
table_subquery_name_pair)
```

```python
    elif node.node_type == "Limit":
        step += "limit the result from table " + \
            node.children[0].get_output_name() + " to " + \
            str(node.plan_rows) + " record(s)"

    else:
        step += "perform " + node.node_type.lower() + " on"
        # binary operator
        if len(node.children) > 1:
            for i, child in enumerate(node.children):
                if i < len(node.children) - 1:
                    step += (" table " + child.get_output_name() + ",")
                else:
                    step += (" and table " + child.get_output_name())
        # unary operator
        else:
            step += " table " + node.children[0].get_output_name()

    # add conditions
    if node.group_key:
        step += " with grouping on attribute " + \
            parse_cond("Group Key", node.group_key, table_subquery_name_pair)

    if node.table_filter:
        step += " and filtering on " + \
            parse_cond("Table Filter", node.table_filter,
                       table_subquery_name_pair)

    if node.join_filter:
        step += " while filtering on " + \
            parse_cond("Join Filter", node.join_filter,
                       table_subquery_name_pair)

    # set intermediate table name
    if increment:
        node.set_output_name("T" + str(cur_table_name))
        step += " to get intermediate table " + node.get_output_name()
        cur_table_name += 1
    if node.subplan_name:
        table_subquery_name_pair[node.subplan_name] = node.get_output_name()

    node.update_desc(step)
    step = "Step " + str(cur_step) + ", " + step + "."
    node.set_step(cur_step)
    cur_step += 1

    steps.append(step)


def random_word(length):
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters) for _ in range(length))
```

```python
def get_text(json_obj):
    global steps, cur_step, cur_table_name, table_subquery_name_pair
    global current_plan_tree
    steps = ["The query is executed as follow."]
    cur_step = 1
    cur_table_name = 1
    table_subquery_name_pair = {}

    head_node = parse_json(json_obj)
    simplified_graph = simplify_graph(head_node)

    to_text(simplified_graph)
    if " to get intermediate table" in steps[-1]:
        steps[-1] = steps[-1][:steps[-1].index(
            " to get intermediate table")] + ' to get the final result.'

    return steps

def clear_cache():
    global steps, cur_step, cur_table_name, table_subquery_name_pair
    steps = []
    cur_step = 1
    cur_table_name = 1
    table_subquery_name_pair = {}

def generate_tree(tree, node, _prefix="", _last=True):
    if _last:
        tree = "{}`-- {}\n".format(_prefix, node.node_type)
    else:
        tree = "{}|-- {}\n".format(_prefix, node.node_type)

    _prefix += "    " if _last else "|   "
    child_count = len(node.children)
    for i, child in enumerate(node.children):
        _last = i == (child_count - 1)
        tree = tree + generate_tree(tree, child, _prefix, _last)
    return tree


def generate_why(node_a, node_b, num):

    text = ""
    if node_a.node_type =="Index Scan" and node_b.node_type == "Seq Scan":
        text = "Reason for Difference " + str(num) + ": "
        text += node_a.node_type + " in P1 on relation " + node_a.relation_name + " has 
now evolved to Sequential Scan in P2 on relation " + node_b.relation_name + ". This is 
because "
        if node_b.index_name is None:
            text += "P1 uses the index, i.e. " + node_a.index_name + ", for selection 
while P2 doesn't. "
        if int(node_a.actual_rows) < int(node_b.actual_rows):
            text += "and the actual row number increases from " +
```

```
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ", "

        if node_a.index_cond != node_b.table_filter and int(node_a.actual_rows) <
int(node_b.actual_rows):
            text += "This may be due to the selection predicates change from " +
(node_a.index_cond if node_a.index_cond is not None else "None ") + " to " +
(node_b.table_filter if node_b.table_filter is not None else "None ") + ". "

    elif node_b.node_type =="Index Scan" and node_a.node_type == "Seq Scan":
        text = "Reason for Difference " + str(num) + ": "
        text += "Sequential Scan in P1 on relation " + node_a.relation_name + " has now
evolved to " + node_b.node_type +" in P2 on relation " + node_b.relation_name + ". This
is because "
        if  node_a.index_name is None:
            text += "P2 uses the index, i.e. " + node_b.index_name + ", for selection
while P1 doesn't. "
        elif node_a.index_name is not None:
            text += "Both P1 and P2 uses the index, which are respectively " +
node_a.index_name + " and " + node_b.index_name + ". "
        if int(node_a.actual_rows) > int(node_b.actual_rows):
            text += "and the actual row number decreases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
        if node_a.table_filter != node_b.index_cond and int(node_a.actual_rows) >
int(node_b.actual_rows):
            text += "This may be due to the selection predicate changes from " +
(node_a.table_filter if node_a.table_filter is not None else "None") + " to " +
(node_b.index_cond if node_b.index_cond is not None else "None") + ". "

    elif node_a.node_type and node_b.node_type in ['Merge Join', "Hash Join", "Nested
Loop"]:
        text = "Reason for Difference " + str(num) + ": "
        if node_a.node_type == "Nested Loop" and node_b.node_type == "Merge Join":
            text += node_a.node_type + " in P1 on has now evolved to " +
node_b.node_type +" in P2 on relation " + ". This is because "
            if int(node_a.actual_rows) < int(node_b.actual_rows):
                text += "the actual row number increases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ", "
            if "=" in node_b.node_type:
                text += "The join condition uses an equality operator. "
            text += "The both side of the Join operator of P2 can be sorted on the join
condition efficiently . "

        if node_a.node_type == "Nested Loop" and node_b.node_type == "Hash Join":
            text += node_a.node_type + " in P1 on has now evolved to " +
node_b.node_type +" in P2 on relation " + ". This is because "
            if int(node_a.actual_rows) < int(node_b.actual_rows):
                text += "the actual row number increases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
            if "=" in node_b.node_type:
                text += "The join condition uses an equality operator. "

        if node_a.node_type == "Merge Join" and node_b.node_type == "Nested Loop":
            text += node_a.node_type + " in P1 on has now evolved to " +
```

```
node_b.node_type +" in P2 on relation " + ". This is because "
            if int(node_a.actual_rows) > int(node_b.actual_rows):
                text += "the actual row number decreases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
            elif int(node_a.actual_rows) < int(node_b.actual_rows):
                text += "the actual row number increases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
                text += node_b.node_type + " joins are used  if the join condition does
not use the equality operator"

        if node_a.node_type == "Merge Join" and node_b.node_type == "Hash Join":
            text += node_a.node_type + " in P1 on has now evolved to " +
node_b.node_type +" in P2 on relation " + ". "
            if int(node_a.actual_rows) < int(node_b.actual_rows):
                text += "This is because the actual row number increases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
            if int(node_a.actual_rows) > int(node_b.actual_rows):
                text += "The actual row number decreases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
            text += "The both side of the Join operator of P2 can be sorted on the join
condition efficiently . "

        if node_a.node_type == "Hash Join" and node_b.node_type == "Nested Loop":
            text += node_a.node_type + " in P1 on has now evolved to " +
node_b.node_type +" in P2 on relation " + ". This is because "
            if int(node_a.actual_rows) > int(node_b.actual_rows):
                text += "the actual row number decreases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
            elif int(node_a.actual_rows) < int(node_b.actual_rows):
                text += "the actual row number increases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
                text += node_b.node_type + " joins are used  if the join condition does
not use the equality operator"

        if node_a.node_type == "Hash Join" and node_b.node_type == "Merge Join":
            text += node_a.node_type + " in P1 on has now evolved to " +
node_b.node_type +" in P2 on relation " + ". "
            if int(node_a.actual_rows) < int(node_b.actual_rows):
                text += "The actual row number increases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
            if int(node_a.actual_rows) > int(node_b.actual_rows):
                text += "The actual row number decreases from " +
str(node_a.actual_rows) + " to " + str(node_b.actual_rows) + ". "
            text += "The both side of the Join operator of P2 can be sorted on the join
condition efficiently. "

    return text


def modify_text(str):
    str = str.replace('perform ', '')
    return str
```

```python
def check_children(nodeA, nodeB, difference, reasons):
    global num
    childrenA = nodeA.children
    childrenB = nodeB.children
    children_no_A = len(childrenA)
    children_no_B = len(childrenB)

    if nodeA.node_type == nodeB.node_type and children_no_A == children_no_B:
        if children_no_A != 0:
            for i in range(len(childrenA)):
                check_children(childrenA[i], childrenB[i],  difference, reasons)

    else:
        if nodeA.node_type == 'Hash' or nodeA.node_type == 'Sort':
            text = 'Difference ' + \
                str(num) + ' : ' + nodeA.children[0].description + \
                ' has been changed to ' + nodeB.description
            text = modify_text(text)
            difference.append(text)
            reason = generate_why(nodeA.children[0], nodeB, num)
            reasons.append(reason)
            num += 1

        elif nodeB.node_type == 'Hash' or nodeB.node_type == 'Sort':
            text = 'Difference ' + str(num) + ' : ' + nodeA.description + \
                ' has been changed to ' + nodeB.children[0].description
            text = modify_text(text)
            difference.append(text)
            reason = generate_why(nodeA, nodeB.children[0], num)
            reasons.append(reason)
            num += 1

        elif 'Gather' in nodeA.node_type:
            check_children(childrenA[0], nodeB, difference, reasons)

        elif 'Gather' in nodeB.node_type:
            check_children(nodeA, childrenB[0],  difference, reasons)
        else:
            text = 'Difference ' + \
                str(num) + ' : ' + nodeA.description + \
                ' has been changed to ' + nodeB.description
            text = modify_text(text)
            difference.append(text)
            reason = generate_why(nodeA, nodeB, num)
            reasons.append(reason)
            num += 1

        if children_no_A == children_no_B:
            if children_no_A == 1:
                check_children(childrenA[0], childrenB[0], difference, reasons)
            if children_no_A == 2:
                check_children(childrenA[0], childrenB[0], difference, reasons)
                check_children(childrenA[1], childrenB[1],  difference, reasons)
```

```python
def get_diff(json_obj_A, json_obj_B):
    global num
    head_node_a = parse_json(json_obj_A)
    clear_cache()
    to_text(head_node_a)

    head_node_b = parse_json(json_obj_B)
    clear_cache()
    to_text(head_node_b)

    num=1
    difference = []
    reasons = []
    check_children(head_node_a, head_node_b, difference, reasons)
    diff_str = ""
    for i in range (len(reasons)):
        diff_str = diff_str + difference[i] + "\n\n"
        if reasons[i] != "":
            diff_str = diff_str + reasons[i] + "\n"
        if i != len(reasons)-1:
            diff_str = diff_str + "-"*200 + "\n"
    return diff_str
```