

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:

examp	ole one
tree of old query	tree of new query
Hash Join (node o1) Seq Scan (node o2) Hash (node o3) Seq Scan (node o4)	Nested Loop (node n1) Seq Scan (node n2) Hash (node n3) Seq Scan (node n4)

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

exam	ole 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.

0.0	Main Frame
Please input old query:	Please input new query:
	view output clear input
Old query execution plan:	New query execution plan:
Old query tree structure:	New query tree structure:
Differer	nce between two query plans:
	clear output
	quit program

3. Sample Queries and Results

3.1 Query 1A and Query 1B

where customer.nationkey = nation.nationkey and customer.custkey >= 75000 and nation.nationkey >= 1 explain (analyze, costs, verbose, buffers, format json) select customer.custkey, customer.name, nation.n Please Input New Query: order by customer.custkey desc; from customer, nation where customer.nationkey = nation.nationkey and customer.acctbal >= 1000 and nation.nationkey >= 10 explain (analyze, costs, verbose, buffers, format json) select customer.custkey, customer.name, nation.n Please Input Old Query: order by customer.custkey desc;

Old Query Execution Plan:

Step 1, perform sequential scan on table customer and filtering on customer acctbal >= '1000' double pr ecision to get intermediate table T1.

Step 2, perform sequential scan on table nation and filtering on nation.nationkey >= 10 to get intermediat Step 3, hash table T2 and perform hash join on table T1 and table T2 under condition customer nationkey Step 4, perform sort on table T3 with attribute customer.custkey DESC to get intermediate table T4. Step 5, perform gather merge on table T4 to get the final result. = nation.nationkey to get intermediate table T3.

Old Query Tree Structure:

-- Sort -- Hash Join |-- Seq Scan -- Gather Merge

-- Sort -- Hash Join |-- Index Scan

-- Seq Scan

New Query Execution Plan:

e table T1.

Step 3, hash table T1 and perform hash join on table customer with index customer pkey and table T1 un der condition customer.nationkey = nation.nationkey to get intermediate table T2.

Step 4, perform sort on table T2 with artifibute customer.custkey DESC to get intermediate table T3.

Step 5, perform gather merge on table T3 to get the final result. The query is executed as follow. Step 1, perform index scan on table customer with index customer_pkey. Step 2, perform sequential scan on table nation and filtering on nation.nationkey >= 10 to get intermediat New Query Tree Structure:

Differences Between Two QEPs and Reasons:

Reason for Difference 1: Sequential Scan in P1 on relation customer has now evolved to Index Scan in P2 on relation customer. This is because P2 uses the index, i.e. customer_bkey, for selection while P1 doesn't. and the actual row number decreases from 40960 to 25000. This may be due to the selection predicate changes from (customer.acctbal >= '1000'::double precision) to (customer.custkey >= 75000). Difference 1 : sequential scan on table customer and filtering on customer.acctbal >= '1000' double precision to get intermediate table T1 has been changed to index scan on table customer with index customer_pkey

3.2 Query 2A and Query 2B

New Query Execution Plan:

Old Query Execution Plan:

from customer, orders where customer.custkey and orders orderdate >= '1996-01-01' and customer.nationkey y >= 10; y >= 10; explain (analyze, costs, verbose, buffers, format json) select orders.orderkey, orders.orderdate, customer Please Input New Query: .custkey, customer.name explain (analyze, costs, verbose, buffers, format json) select orders.orderkey, orders.orderdate, customer where customer custkey = orders custkey and orders orderkey >= 40000 and customer custkey >= 50000, Please Input Old Query: from customer, orders

Step 3, hash table T2 and perform hash join on table T1 and table T2 under condition orders.custkey = cu stomer.custkey to get the final result. The query is executed as follow. Step 1, perform sequential scan on table orders and filtering on orders.orderdate >= '1996-01-01' date t o get intermediate table T1. Step 2, perform sequential scan on table customer and filtering on customer.nationkey >= 10 to get inter mediate table T2. New Query Tree Structure: -- Seq Scan -- Hash Join |-- Seq Scan `-- Hash The query is executed as follow.

Step 1, perform sequential scan on table orders and filtering on orders.orderkey >= 40000 to get interme diate table 11, perform index scan on table customer with index customer_pkey.

Step 2, perform index scan on table customer with index customer_pkey.

Step 3, hash table customer with index customer_pkey and perform hash join on table T1 and table customer with index customer_pkey and perform hash join on table T1 and table customer with index customer_pkey under condition orders.custkey = customer.custkey to get the final result. Old Query Tree Structure:

`-- Hash Join |-- Seq Scan `-- Hash `-- Index Scan

Differences Between Two QEPs and Reasons:

Reason for Difference 1: Index Scan in P1 on relation customer has now evolved to Sequential Scan in P2 on relation customer. This is because P1 uses the index, i.e. customer_pkey, for selection while P2 doesn't. Difference 1: index scan on table customer with index customer_pkey has been changed to sequential scan on table customer and filtering on customer nationkey >= 10 to get intermediate table T2

3.3 Query 3A and Query 3B

Reason for Difference 1: Index Scan in P1 on relation part has now evolved to Sequential Scan in P2 on relation part. This is because P1 uses the index, i.e. part_pkey, for selection while P2 doesn't. and the actual row nu Step 1, perform sequential scan on table lineitem and filtering on lineitem.commitdate > '1997-01-01' da te to get intermediate table T1. Step 2, perform sequential scan on table part. Step 2, hash table part and perform hash join on table T1 and table part under condition lineitem.partkey Step 3, hash table part and perform hash join on table T1 and table T2 and table T2. Step 4, perform sort on table T2 with attribute lineitem.commitdate DESC to get intermediate table T3. Step 5, perform gather merge on table T3 to get the final result. explain (analyze, costs, verbose, buffers, format json) select part.brand, part.partkey, lineitem.commitdat where lineitem.partkey = part.partkey and lineitem.commitdate > '1997-01-01' order by lineitem.commitdate desc; New Query Tree Structure: New Query Execution Plan: Please Input New Query: Differences Between Two QEPs and Reasons: .-- Gather Merge .-- Sort .-- Hash Join I-- Seq Scan .-- Hash -- Seq Scan mber increases from 16667 to 66667, This may be due to the selection predicates change from (part.partkey >= 150000) to None . from part, lineitem Difference 1: index scan on table part with index part_pkey has been changed to sequential scan on table part Step 2, perform index scan on table part with index part_pkey. Step 3, hash table part with index part_pkey and perform hash join on table lineitem and table part with in dex part_pkey under condition lineitem.partkey = part_partkey to get intermediate table T1. Step 4, perform sort on table T1 with attribute part_partkey DESC to get intermediate table T2. Step 5, perform gather merge on table T2 to get the final result. explain (analyze, costs, verbose, buffers, format json) select part.brand, part.partkey, lineitem.commitdat where lineitem.partkey = part.partkey and part.partkey >= 150000 Old Query Execution Plan: Old Query Tree Structure: Please Input Old Query: Step 1, perform sequential scan on table lineitem. order by part.partkey desc; -- Sort -- Hash Join |-- Seq Scan -- Index Scan

3.4 Query 4A and Query 4B

Please Input Old Query:

explain (analyze, costs, verbose, buffers, format json) select customer.custkey, customer.name, nation_a. from customer, (select nationkey, name from nation where nationkey >= 10) as nation_a where customer.nationkey = nation_a.nationkey and customer.acctbal >= 1000 order by customer.custkey desc;

Please Input New Query:

explain (analyze, costs, verbose, buffers, format json) select customer_a.custkey, customer_a.name, nati from (select name, custkey, nationkey from customer where customer.custkey <= 750) as customer_a, n where customer_a.nationkey = nation.nationkey and nation.nationkey >= 10 order by customer_a.custkey desc; on.name ation

New Query Execution Plan:

Step 1, perform index scan on table customer with index customer_pkey. | Step 2, perform sequential scan on table nation and filtering on nation.nationkey >= 10 to get intermediat The query is executed as follow. e table T1.

Step 3, hash table T1 and perform hash join on table customer with index customer_pkey and table T1 un der condition customer nationkey = nation nationkey to get intermediate table T2. Step 4, perform sort on table T2 with attribute customer custkey DESC to get the final result.

Step 3, hash table T2 and perform hash join on table T1 and table T2 under condition customer nationkey

= nation.nationkey to get intermediate table T3. Step 4, perform sort on table T3 with attribute customer.custkey DESC to get intermediate table T4.

Old Query Tree Structure:

-- Sort -- Hash Join I-- Seq Scan -- Hash

--- Seq Scan

Step 1, perform sequential scan on table customer and filtering on customer.acctbal >= '1000' double pr Step 2, perform sequential scan on table nation and filtering on nation.nationkey >= 10 to get intermediat

ecision to get intermediate table T1.

The query is executed as follow.

Old Query Execution Plan:

New Query Tree Structure:

|-- Index Scan |-- Hash |-- Seq Scan - Sort `-- Hash Join

Differences Between Two QEPs and Reasons:

Difference 1 : gather merge on table T4 to get intermediate table T5 has been changed to hash table T1 and hash join on table customer with index customer_pkey and table T1 under condition customer.nationkey = nation.nationkey to get intermediate table T2

Reason for Difference 1:

Difference 2: hash table T2 and hash join on table T1 and table T2 under condition customer.nationkey = nation.nationkey to get intermediate table T3 has been changed to hash table T1 and hash join on table customer with index customer_pkey and table T1 under condition customer.nationkey = nation.nationkey to get intermediate table T2

Reason for Difference 2:

3.5 Query 5A and Query 5B

	Main Frame
Please Input Old Query:	Please Input New Query:
explain (analyze, costs, verbose, buffers, format ison) select * from nation, region where nation.regionkey = region.regionkey and nation.nationkey >= 15;	explain (analyze, costs, verbose, buffers, format json) select from nation, region where nation.regionkey = region.regionkey and region.regionkey = 1;
Old Query Execution Plan:	New Query Execution Plan:
The query is executed as follow. Step 1, perform sequential scan on table region. Step 2, perform sequential scan on table nation and filtering on nation.nationkey >= 15 to get intermediat e table T1. Step 3, hash table T1 and perform hash join on table region and table T1 under condition region.regionke y = nation.regionkey to get the final result.	The query is executed as follow. Step 1, perform sequential scan on table nation and filtering on nation.regionkey = 1 to get intermediate t able T1. Step 2, perform index scan on table region with index region_pkey. Step 3, perform nested loop on table T1, and table region with index region_pkey to get the final result.
Old Query Tree Structure:	New Query Tree Structure:
` Hash Join Seg Scan ` Hash ` Seg Scan	` Nested Loop Seg Scan Index Scan
Differences Bet	Differences Between Two QEPs and Reasons:
Difference 1 : hash table T1 and hash join on table region and table T1 under condition region.regionkey = nation.regionkey to get intermediate table T2 has been changed to nested loop on table T1, and table region with index region_pkey to get intermediate table T2	ion.regionkey to get intermediate table T2 has been changed to nested loop on table T1, and table region wit
Reason for Difference 1: hash Join in P.1 on has now evolved to Nested Loop in P.2 on relation. This is because the actual row number decreases from 10 to 5. Difference 2: sequential scan on table nation and filtering on nation.nationkey >= 15 to get intermediate table T1 has been changed to index scan on table region with index region_pkey	e tne actual row number decreases from 10 to 5. T1 has been changed to index scan on table region with index region_pkey
Reason for Difference 2: Sequential Scan in P1 on relation nation has now evolved to Index Scan in P2 on relation region. This is because P2 uses the index, i.e. region_pkey, for selection while P1 doesn't. and the actual row number decreases from 10 to 1. This may be due to the selection predicate changes from (nation.nationkey >= 15) to (region.regionkey = 1).	ion region. This is because P2 uses the index, i.e. region_bkey, for selection while P1 doesn't. and the actual ey >= 15) to (region.regionkey = 1).

	Main Frame
Please Input Old Query:	Please Input New Query:
explain (analyze, costs, verbose, buffers, format ison) select * from (SELECT supplier.nationkey, supplier.suppkey FROM supplier WHERE 20=suppkey) AS a join (SELECT nation.nationkey, nation.regionkey FROM nation) As b on a.nationkey = b.nationkey	explain (analyze, costs, verbose, buffers, format json) select * from (SELECT supplier nationkey, supplier suppkey FROM supplier WHERE 200 <suppkey) (select="" a="" a.nationkey="b.nationkey</td" as="" b="" from="" join="" nation)="" nation.nationkey,="" nation.regionkey="" on=""></suppkey)>
Old Query Execution Plan:	New Query Execution Plan:
The query is executed as follow. Step 1, perform index scan on table supplier with index supplier_pkey. Step 2, perform index scan on table nation with index nation_pkey. Step 3, perform nested loop on table supplier with index supplier_pkey, and table nation with index nation_pkey to get the final result.	The query is executed as follow. Step 1, perform sequential scan on table supplier and filtering on 200 < supplier.suppkey to get intermediate table T1. Step 2, perform sequential scan on table nation. Step 2, perform sequential scan on table nation. Step 3, hash table nation and perform hash join on table T1 and table nation under condition supplier.nationkey = nation.nationkey to get the final result.
Old Query Tree Structure:	New Query Tree Structure:
` Nested Loop Index Scan Index Scan	` Hash Join Seq Scan ` Hash ` Seq Scan
Differences Bet	Differences Between Two QEPs and Reasons:
Difference 1 : nested loop on table supplier with index supplier_pkey, and table nation with index nation_pke; n under condition supplier.nationkey = nation.nationkey to get intermediate table T2	Difference 1 : nested loop on table supplier with index supplier_pkey, and table nation with index nation_bkey to get intermediate table T1 has been changed to hash table nation and hash join on table T1 and table nation under condition supplier nationkey to get intermediate table T2
Reason for Difference 1: Nested Loop in P1 on has now evolved to Hash Join in P2 on relation . This is because the actual row number increases from 1 to 9800.	e the actual row number increases from 1 to 9800.
Difference 2: index scan on table supplier with index supplier_pkey has been changed to sequential scan on table supplier and filtering on 200 < supplier.supplier, supplier, get intermediate table T1 Reason for Difference 2: Index Scan in P1 on relation supplier has now evolved to Sequential Scan in P2 on relation supplier. This is because P1 uses the index, i.e. supplier_pkey, for selection chair reverses from 1 to 0800. This may be due to the selection requires change from 1200 < supplier pkey, for selection	_pkey has been changed to sequential scan on table supplier and filtering on 200 < supplier.suppkey to get intermediate table T1 has now evolved to Sequential Scan in P2 on relation supplier. This is because P1 uses the index, i.e. supplier_pkey, for selection while P2 doesn't, and the a feature from (20 - europies enoughed) to the selection prodicates change from (20 - europies enoughed) to the selection prodicates change from (20 - europies enoughed) to the selection prodicates change from (20 - europies enoughed) to the selection prodicates change from (20 - europies enoughed) to the selection prodicates change from (20 - europies enoughed) to the selection prodicates change from (20 - europies enoughed) to the selection product to
Difference 3 : index scan on table nation with index nation_pkey has been changed to sequential scan on table nation	nation
Reason for Difference 3: Index Scan in P1 on relation nation has now evolved to Sequential Scan in P2 on relation nation. This is because P1 us row number increases from 1 to 25, This may be due to the selection predicates change from (nation.nationkey = supplier.nationkey) to None.	Reason for Difference 3: Index Scan in P1 on relation nation has now evolved to Sequential Scan in P2 on relation nation. This is because P1 uses the index, i.e. nation_pkey, for selection while P2 doesn't. and the actual row number increases from 1 to 25, This may be due to the selection predicates change from (nation.nationkey = supplier.nationkey) to None.

3.7 Query 7A and Query 7B

New Query Execution Plan:

explain (analyze, costs, verbose, buffers, format ison) select * from (SELECT supplier.nationkey, supplier.suppkey FROM supplier WHERE 200>suppkey ORDER BY supp fler.nationkey) AS a join (SELECT nation.nationkey, nation.regionkey FROM nation ORDER BY nation.nationkey) As b on a.nationkey = b.nationkey) Please Input New Query: explain (analyze, costs, verbose, buffers, format ison) select* from (SELECT supplier, nationkey, supplier, suppkey RROM supplier WHERE 200<suppkey) AS a join (SELECT nation, nationkey, nation, regionkey FROM nation) As b on a nationkey = b.nationkey Please Input Old Query:

New Query Execution Plan:	The query is executed as follow. Step 1, perform sequential scan on table nation. Step 2, perform sequential scan on table nation. Step 3, perform sort on table supplier with index supplier_pkey. Step 4, perform sort on table supplier with index supplier_pkey with attribute supplier_nationkey to get intermediate table 72. Step 5, perform materialize on table 12 to get intermediate table 13. Step 6, perform materialize on table 12 to get intermediate table 13. Step 6, perform materialize on table 12 to get intermediate table 13. Step 6, perform materialize on table 12 to get intermediate table 13. Step 6, perform materialize on table parloam merge join on table nation and table 13 to get the final result.
Old Query Execution Plan:	The query is executed as follow. Step 1, perform sequential scan on table supplier and filtering on 200 < supplier.suppkey to get intermedi and table be to the condition of the condition supplier. Step 2, perform sequential scan on table nation. Step 3, hash table nation and perform hash join on table T1 and table nation under condition supplier.nati onkey = nation.nationkey to get the final result.

New Query Tree Structure:	Merge Join Sort ' Sort ' Materialize ' Sort ' Index Scan
Old Query Tree Structure:	Hash Join Seq Scan Seq Scan Seq Scan

Differences Between Two QEPs and Reasons:

Difference 1: hash table nation and hash join on table T1 and table nation under condition supplier nationkey = nation.nationkey to get intermediate table T2 has been changed to sort and table nation and merge join or table nation and table T3 to get intermediate table T4
Reason for Difference 1: Hash Join in P1 on has now evolved to Merge Join in P2 on relation. The actual row number decreases from 9800 to 199. The both side of the Join operator of P2 can be sorted on the join concition efficiently.
Ofference 2 : sequential scan on table supplier and filtering on 200 < supplier, suppkey to get intermediate table T1 has been changed to sequential scan on table nation
Difference 3 : sequential scan on table nation has been changed to materialize on table T2 to get intermediate table T3
Difference 4 : sequential scan on table nation has been changed to index scan on table supplier with index supplier_pkey

Reason for Difference 4: Sequential Scan in P1 on relation nation has now evolved to Index Scan in P2 on relation supplier. This is because P2 uses the index, i.e. supplier_pkey, for selection while P1 doesn't.

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]
```

```
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)
```

```
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:</pre>
                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:</pre>
                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:</pre>
                        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 "
            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 "
                    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)
```

```
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:</pre>
                    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))
```

```
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):</pre>
            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) <</pre>
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):</pre>
                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):</pre>
                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):</pre>
                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):</pre>
                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):</pre>
                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):</pre>
                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
```

```
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)
```

```
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
```