NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Nanyang Technological University

CZ4031 Database System Principles

Assignment 2

Prepared by:

Name	Matriculation Number
Au Yew Rong Roydon	U2021424J
Ahmad Syafiq Bin Ahmad Ghozali	U2022711K
Damien Lee Wen Hao	U2022328A
Chen Wei Yi	U2021480L
Ong Xin Rui Celestine	U2022737G

School of Computer Science and Engineering
Nanyang Technology University

Date: 12/11/2022

1. Introduction

In this project, we are tasked with the goal to translate and annotate a given SQL Query by obtaining their Optimal QEP and respective AQPs. Using these AQPs, we are able to explain how different components of the query are executed by the underlying query processor and why the operators are chosen among other alternatives.

1.1 File Structure

Our project is split into four main python files:

1) **Project.py**
   This is the main file that our application uses that invokes all the necessary procedures from the other three files.
2) **Interface.py**
   This file is incharge of the display of the optimal query plan together with annotations on the plan. This file also contains the logic for how the user interacts with the program. (e.g. Handling errors from user inputs)
3) **Preprocessing.py**
   The main responsibilities of these file are:
   1. To establish connection with the database tables
   2. Obtain Optimal QEP
   3. Obtain AQPs
   4. Make preparations before passing over to annotations to annotate the Optimal QEP.
4) **Annotation.py**
   This file is in charge of annotating respective tree nodes in the optimal QEP.

1.2 Libraries Used

The libraries that are required to run this project are:

- Psycopg2, which is a library that is used as a PostgreSQL database adapter for Python.
- Tkinter, which is a built-in python library that is made for building simple GUI for application in python

1.3 Instructions

a. Ensure python version is 3.8 (For 3.10 version and above psycopg2 wont run and an error will show: ModuleNotFoundError: No module named 'psycopg2')
b. Pip install psycopg2 (If that doesn't work, use pip install psycopg2-binary)
c. Change to the correct directory and Run the project.py file

1.4 Contributions

Everyone contributed equally in this project.

2. GUI

Fig 1 below shows the overall GUI layout that we have designed.

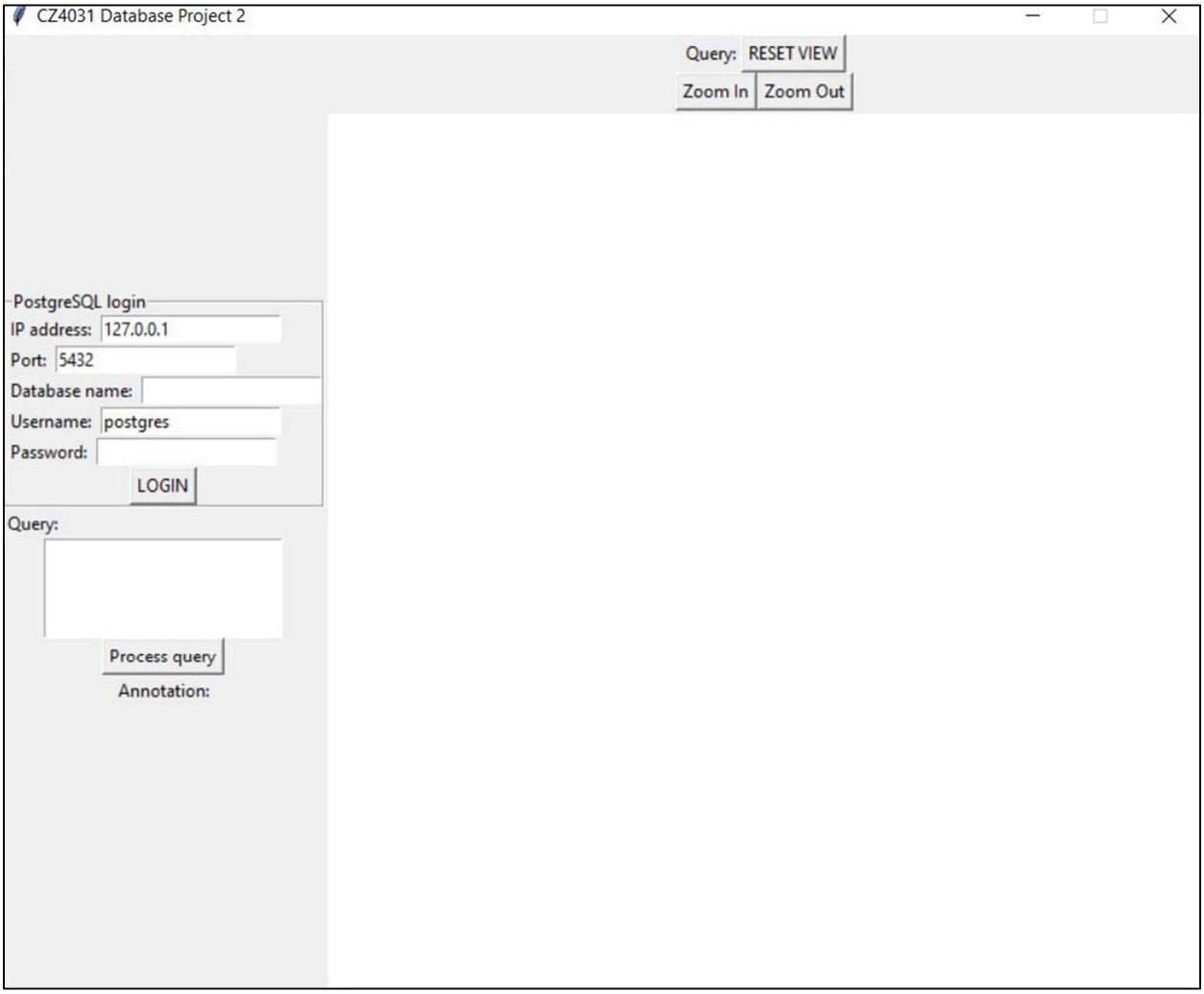


Fig 1 Overall GUI Of Project

2.1 Running the program

To use our application, the user will first have to connect to the PostgreSQL server. If the user is not logged in, they are not able to use any of the features of the application and would be shown an error message that states that the user needs to be logged in. Thus to login the user has to enter this five fields below:

1. Host name
2. Port Number
3. Database name
4. Username
5. Password

Once the user enters the required information he is then able to login and a success message would pop-up, as shown in Fig 2.

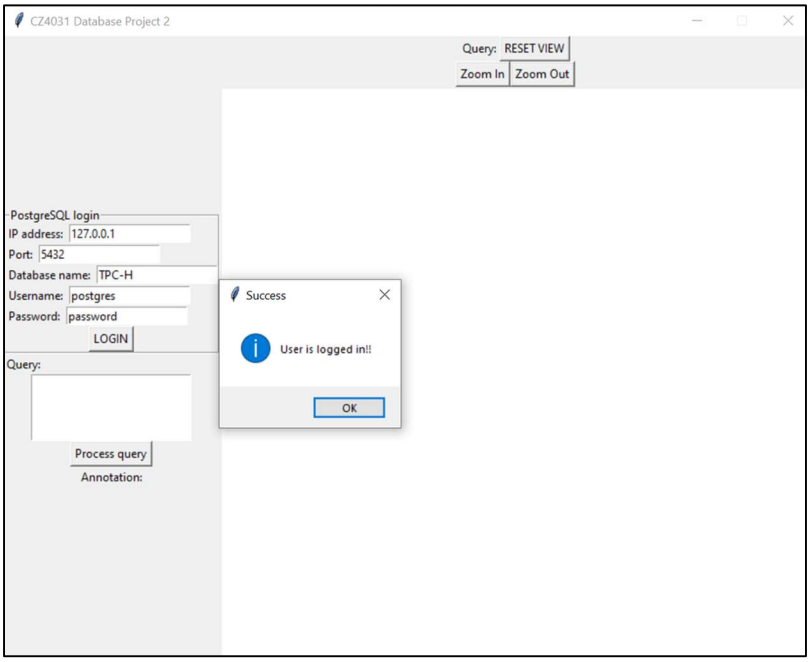


Fig 2 Successful Login

Next, the user can enter a query. By clicking on the Process Query button, the Optimal Query Plan tree would be generated, which is the Optimal Query Plan (Fig 3).

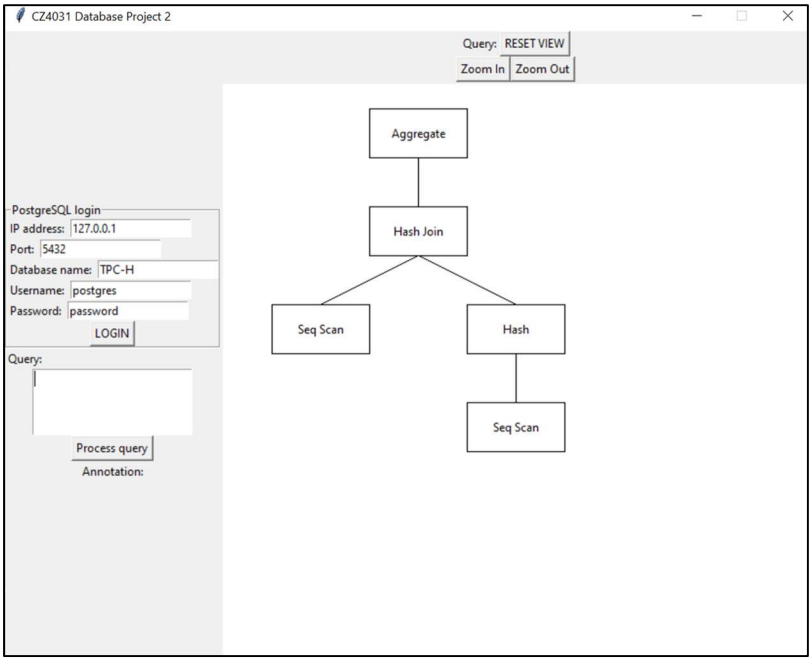


Fig 3 Displaying Tree

By clicking on the leaf nodes in the tree, users would be able to view the respective annotation for that particular node. They would be greeted with a pop up that has the respective text associated with that particular tree node (Fig 4).

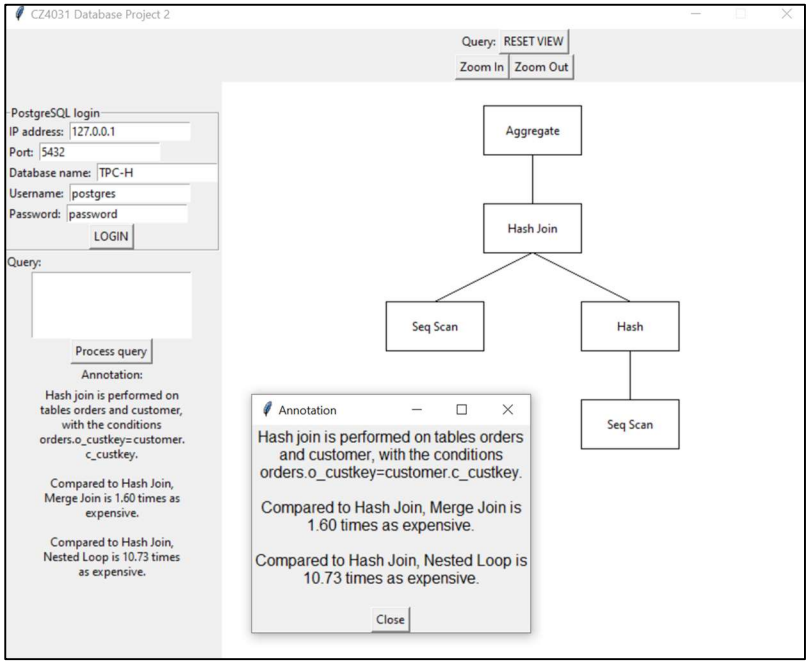
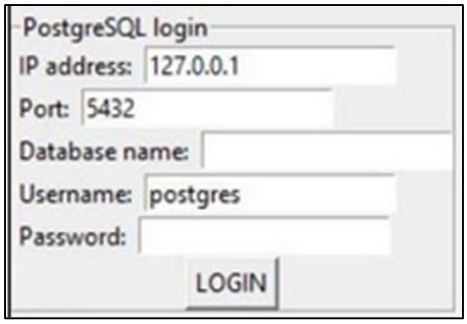


Fig 4 Annotation of hash join

2.2 Features

To improve the user-friendliness of our application, we have incorporated some extra features. These features makes the application more intuitive and simple to use. The first feature would be the filling in of default values in the login section of the application. This would cut down the number of inputs that the user is required to enter, thus streamlining the login process. (Fig 5)



The second feature would be the validation messages that would pop up to notify the user of any mistakes that were made during the usage of the application (example can be seen in Fig 6).

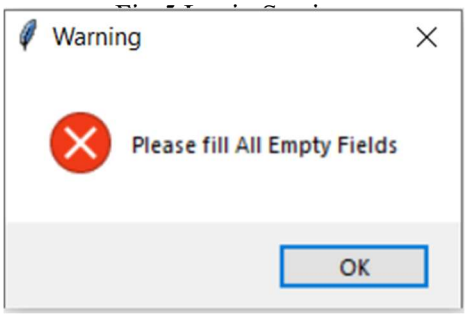


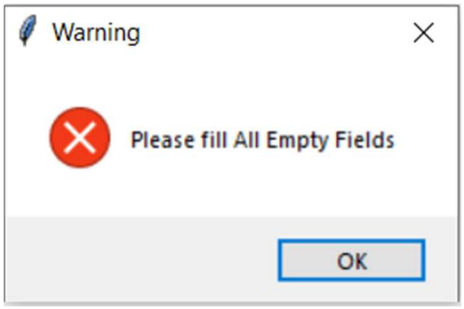
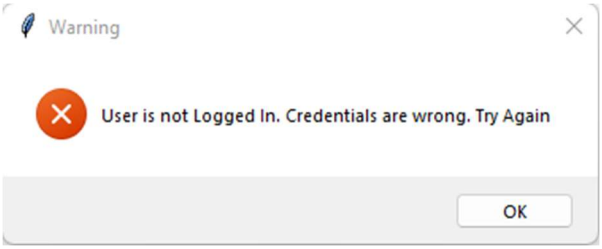
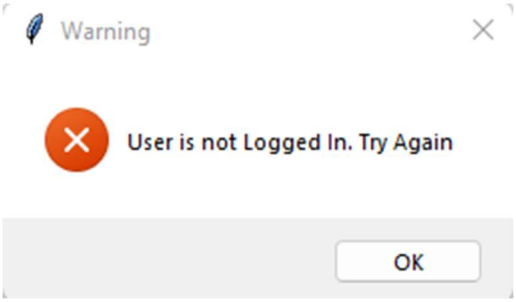
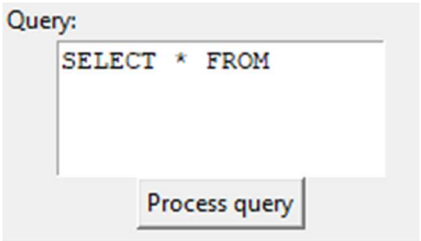
Fig 6 Error for empty field

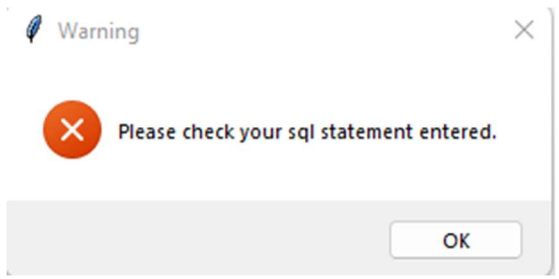
The third feature that was included would be the zooming in and out of the tree. For large trees, the user would hence be able to zoom out for a complete view. By clicking on the reset view, the user would easily go back to the original view.

Lastly, after successfully generating the optimal query tree, the query box would be automatically cleared for the user to easily enter the next query.

2.3 Error Handling

Our application has implemented the feature of error handling as explained above, however this section would be dedicated to explain each of the possible error messages you may encounter during the execution of the application and how to resolve them.

Errors	Description/Solution
	**Reason:** This error is triggered when the user did not enter the login details as explained in the section "Running the program" **Solution:** This can be resolved by entering all the fields in the PostgreSQL Login section of the application
	**Reason:** This error is triggered when the user enters a wrong login credential. **Solution:** This can be resolved by entering the right credentials
	**Reason:** This error is triggered when the user clicks the "Process Query" button without login. **Solution:** User has to first login with the right credentials
	**Reason:** This error is triggered when the user enters a query that is not understandable by Postgres such as misspelling certain parts of the query. **Solution:** This can be resolved by checking the

	query entered and editing any mistakes

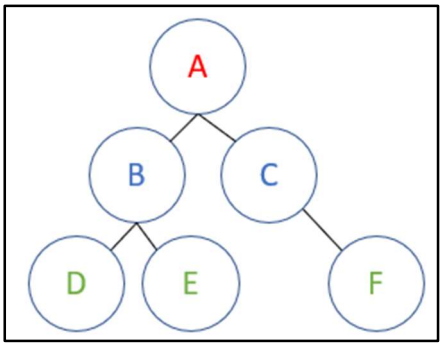
2.4 Displaying a Tree

To generate the tree, we need to know where to place each node. We can make use of the fact that each node has up to 2 children to help us. For nodes with 1 child, we can simply put it just above the child node. The issue comes with nodes with multiple children, we firstly need to know how to space the children out such that each child has enough space for their respective descendents.

To do that, we simply segment the graph into cells. Each leaf node should occupy 1 cell. To locate where other nodes should occupy, we sum up the number of leaf nodes that are its descendents. We therefore know that eventually each leaf node must occupy 1 cell, so the node should be put in the middle of the number of cells. For example imagine such a graph:

We know that nodes D, E and F each occupy a cell. We will allocate cell index 0, 1 and 2 to D, E and F respectively. We know that C can be placed directly above F in cell index 2, while B has to be in the middle of cell index 0 and 1. Lastly, A will be placed in the middle of 0, 1 and 2.

We then calculate the x and y coordinates of the node using the cell index and the depth of the node.



3. Preprocessing

3.1 Establishing a connection

To connect to the database, we utilized psycopg2 which is the most popular PostgreSQL database adapter for the Python programming language, we used the connect function to establish a stable connection to the PostgreSQL database.

The user will first be prompted to key in: Host (Ip Address), Port, Database Name, Username, Password (Fig 7).

```
# , host, port_num, database_name, username, password
def __init__(self, host, port_num, database_name, username, password):
    # Setting up connection
    self.connection = None
    try:
        self.connection = psycopg2.connect(host = host, port = port_num,database= database_name, user= username, password= password)
        #self.connection = psycopg2.connect(host = config("HOST"), port = config("PORT") ,database= config("DATABASE"), user= config("
        self.verify = True
        if(self.verify):
            print("Connected")
            self.cursor = self.connection.cursor()
    except (Exception, psycopg2.DatabaseError):
        self.verify = False
```

Fig 7 Connection to Database

We used a try-except block to catch connection failures. In the case that the user keys in incomplete information or wrong credentials, the connection will fail and will be caught by the except block. The interface would then inform the user of the failed connection.

In the event of success, the class is constructed with a connection attribute and a cursor that will be used to execute queries.

3.2 Obtaining Optimal QEP

Given an input query from the UI, the executeQuery() function will be called with a click of the Process Query button. Using the connection established with the database we can obtain the best query plan that PostgreSQL has decided on. This is through the use of the EXPLAIN statement together with the query to be executed (Fig 8).

```
def executeQuery(self, query, off=[]):
    optimalQEP = "EXPLAIN (VERBOSE, COSTS, FORMAT JSON)" + query.strip()

    try:
        # set cursor variable
        cursor = self.cursor
        #Setting off for alternate query plans
        # Default turn off variables not used
        cursor.execute(self.off_config["Tid Scan"])
        cursor.execute(self.off_config["Index Only Scan"])
        for condition in off:
            cursor.execute(self.off_config[condition])

        cursor.execute(optimalQEP)

        explain_query = cursor.fetchall()

        # Setting config back on to set up for next alternate query plan
        for condition in off:
            #print(self.on_config[condition])
            cursor.execute(self.on_config[condition])
        # write explain details into json file
        # with open('chosenQueryPlan.json', 'w') as output_file:
        #     chosenQueryPlan = (json.dump(explain_query, output_file, ensure_ascii = False, indent = 4))
        self.queryError = False

        return explain_query[0][0][0]['Plan']
    except(Exception, psycopg2.DatabaseError) as error:

            # Check how to seperate errors
            print("Your error is: ", error)
            # print("Your query is: ", query)
            message = "Please check your sql statement: \n" + query
            print(message)
            self.queryError = True
            return "error"
```

Fig 8 Execute Query Function

Since we will be comparing the costs with alternate query plans, the EXPLAIN statement would provide us with the details needed for comparison. We decided to use the EXPLAIN statement without the ANALYZE parameter so as to estimate the costs of the operators in the Optimal QEP without actually executing the query. This reduces the runtime especially if we were to estimate Alternate QEPs which might have very high costs.

Before running the statement, we also turned off Tid Scan and Index Only Scan for comparison consistency across all plans. This would mean that queries will not be able to utilize Tid Scan and Index Only Scan. We turned it off since we only compared scans that were taught in lecture.

In the case of an error due to mistyping of SQL statements, the user will be informed.

After successful execution, the data returned would be in JSON format and to extract the relevant details of the plan we created a PlanNode Class to store relevant information for each operator, as shown in Fig 9.

```
class PlanNode():

    # Construct a node
    def __init__(self) -> None:
        self.parent = None
        self.children = []
        self.attributes = {}
        self.annotations = ""
        self.alternate_plans = {} # Key: alternative plan used, Value: ratio, how much more costly (alternate plan cost / optimal cost)

    def print_tree(self):
        queue = []
        queue.append(self)
        while len(queue) != 0:
            childlength = len(queue)
            # Add appropriate children ndoes
            for i in range(childlength):
                node = queue.pop(0)
                if node != None:
                    print(node.attributes['Node Type'] + "->cost: ", node.attributes['Total Cost'], "->Alternate Plans: ", node.alternate_plans , end='\n')
                for child in node.children:
                    queue.append(child)
```

Fig 9 Plan node Class

The attributes are as follows:

Attributes	Description
parent	Parent PlanNode which is the next operator receiving the results
children	List of Children PlanNodes
attributes	Dictionary containing different attributes such as Node Type and Cost
annotations	Annotation on a node (Operator)
alternate_plans	Dictionary of alternate query plans for a particular operator and ratio of the total alternative query plan cost to the total optimal query plan cost

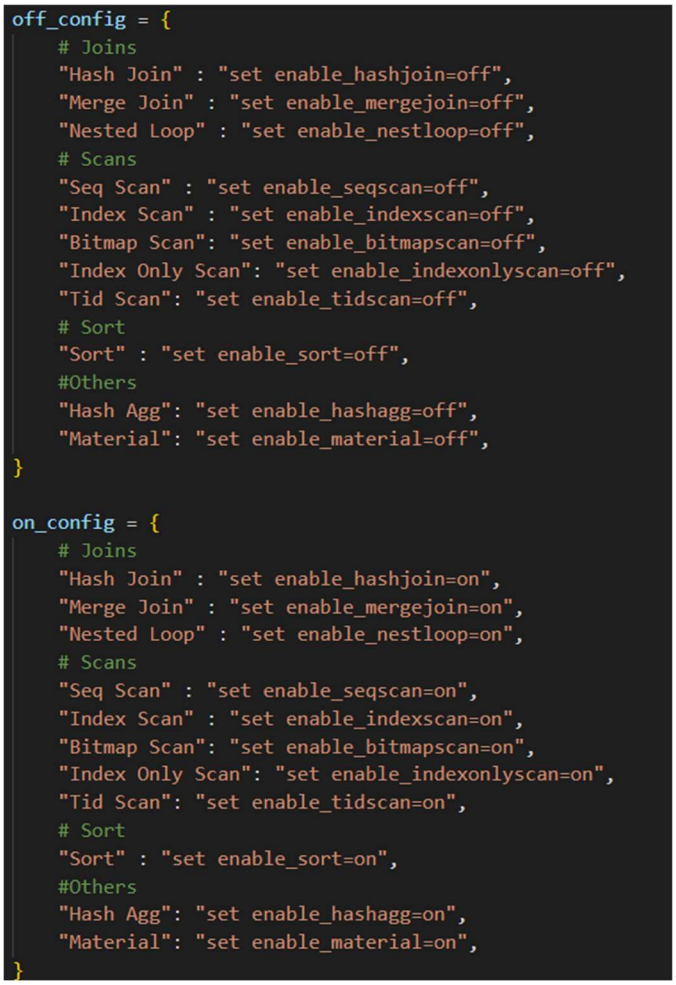
From Fig 10, to build the query plan tree, we used a recursive function to create a root node, and create its children recursively. This will return us the root node which represents a tree structure of the query plan.

```
def add_attributes(self, plan, node):
    """
    Get current plan and insert the corresponding attributes into the current node
    """
    for key,val in plan.items():
        if key != "Plans":
            node.attributes[key] = val

def add_node(self, plan,node):
    """
    Recursive create nodes based on the number of "Plans" of current node (Each "Plans" corresponds to an additional child node)
    """
    # Break condition when no further nodes need to be created
    if "Plans" not in plan:
        return
    for plan in plan["Plans"]:
        # Create PlanNode
        child = PlanNode()
        child.parent = node
        self.add_attributes(plan,child)
        self.add_node(plan,child)
        node.children.append(child)

def build_tree(self, plan):
    """
    Build the query plan tree
    """
    if (plan == "error"):
        print("Please check your sql statements")
        return
    root = PlanNode()
    self.add_attributes(plan, root)
    self.add_node(plan,root)
    return root
```

Fig 10 Functions used for building tree

3.3 Obtaining AQPs (Algorithm)

To annotate the optimal query plan, we would have to first extract out multiple AQPs to compare with. However, to generate different AQPs, we would have to force the query processor to execute without a certain condition by turning the configuration off. This is done using the SET word and the code shown in Fig 11, which turns off conditions forcing the query processor to not be able to utilise a certain kind of join or scan etc. (EG: turning off index scan forces the processor to not use index scan). To ensure a fair comparison, after turning them off, we would turn them on again before turning off other conditions.



```
off_config = {
    # Joins
    "Hash Join" : "set enable_hashjoin=off",
    "Merge Join" : "set enable_mergejoin=off",
    "Nested Loop" : "set enable_nestloop=off",
    # Scans
    "Seq Scan" : "set enable_seqscan=off",
    "Index Scan" : "set enable_indexscan=off",
    "Bitmap Scan": "set enable_bitmapscan=off",
    "Index Only Scan": "set enable_indexonlyscan=off",
    "Tid Scan": "set enable_tidscan=off",
    # Sort
    "Sort" : "set enable_sort=off",
    #Others
    "Hash Agg": "set enable_hashagg=off",
    "Material": "set enable_material=off",
}

on_config = {
    # Joins
    "Hash Join" : "set enable_hashjoin=on",
    "Merge Join" : "set enable_mergejoin=on",
    "Nested Loop" : "set enable_nestloop=on",
    # Scans
    "Seq Scan" : "set enable_seqscan=on",
    "Index Scan" : "set enable_indexscan=on",
    "Bitmap Scan": "set enable_bitmapscan=on",
    "Index Only Scan": "set enable_indexonlyscan=on",
    "Tid Scan": "set enable_tidscan=on",
    # Sort
    "Sort" : "set enable_sort=on",
    #Others
    "Hash Agg": "set enable_hashagg=on",
    "Material": "set enable_material=on",
}
```

Fig 11 Image of Configuration

```
for condition in off:
    cursor.execute(self.off_config[condition])

cursor.execute(optimalQEP)

explain_query = cursor.fetchall()

# Setting config back on to set up for next alternate query plan
for condition in off:
    #print(self.on_config[condition])
    cursor.execute(self.on_config[condition])
```

Fig 12 Conditional Query

Due to the great number of possible combinations of turning off the configurations, we restricted the number of Alternate Query Plan Generated to 6 different algorithms. For example, if we want to only use sort-merge join algorithms for all joins we will input [‘Nested Loop’, ‘Hash Join’] as our off_conditions so that PostgreSQL cannot use these algorithms to join and can only use sort-merge join.

Our 6 algorithms are show below (Fig 13):

Joins: Only Sort-Merge Join, Only Hash Join, Only Nested Loop Join

Scans: Only Sequential Scan, Only Index Scan, Only Bitmap Scan

```
#Alternate plans (Max: 6)
#Checking for AEP for Joins
#Full Merge Join
plan = self.executeQuery(query, ["Nested Loop", "Hash Join"])
plan_type = "Merge Join"
self.addToAlternatePlans(plan, plan_type)

#Full hash join
plan = self.executeQuery(query, ['Nested Loop', "Merge Join"])
plan_type = "Hash Join"
self.addToAlternatePlans(plan, plan_type)

#Full nested loop join
plan = self.executeQuery(query, ['Merge Join', "Hash Join"])
plan_type = "Nested Loop"
self.addToAlternatePlans(plan, plan_type)

#Checking for AEP for Scans
#Seq scan
plan = self.executeQuery(query, ["Index Scan", "Bitmap Scan"])
plan_type = "Seq Scan"
self.addToAlternatePlans(plan, plan_type)

# Index Scan
plan = self.executeQuery(query, ["Seq Scan", "Bitmap Scan"])
plan_type = "Index Scan"
self.addToAlternatePlans(plan, plan_type)

# Bitmap Scan
plan = self.executeQuery(query, ["Seq Scan", "Index Scan"])
plan_type = "Bitmap Scan"
self.addToAlternatePlans(plan, plan_type)
```

Fig 13 The 6 algorithms

To prevent generating plans that are exactly the same as the optimal plan, we used our compareTree function (Fig 14) to check node by node. If all nodes are exactly the same as the optimal plan, the alternate plan would not be added to our alternative_plans list (Fig 15).

```
def compareTree(self, chosen_plan_root, alternate_plan_root):
    queue1 = []
    queue2 = []
    queue1.append(chosen_plan_root)
    queue2.append(alternate_plan_root)
    while len(queue1) != 0:
        childlength = len(queue1)
        # Add appropriate children ndoes
        for i in range(childlength):
            node1 = queue1.pop(0)
            node2 = queue2.pop(0)
            if node1.attributes['Node Type'] != node2.attributes['Node Type']:
                return 0
            for child in node1.children:
                queue1.append(child)
            for child in node2.children:
                queue2.append(child)
    return 1
```

Fig 14 Function for comparing alternate and chosen plan

```
def addToAlternatePlans(self, plan, plan_type):
    if(plan == "error"):
        print(plan_type + " not added")
        return
    alternate_root = self.build_tree(plan)
    # If both trees are the same dont add to alternate plan
    if(self.compareTree(self.query_plans['chosen_plan'][1], alternate_root) == 1):
        print(plan_type + " not added")
        return
    cost = self.computeTotalCost(alternate_root)
    self.query_plans["alternative_plans"].append((plan_type,alternate_root, cost))
```

Fig 15 Total cost of a query plan

We also utilized the computeTotalCost function to calculate the total cost of the alternative query plan and placed them all together in a tuple (Fig 16). Hence each tuple consists of plan_type (EG: Index Nested Loop Join), the root PlanNode and lastly the cost of the plan.

```
query_plans = {
    "chosen_plan": "", # Will be replaced by a tuple of (plan_type, root_node, cost)
    "alternative_plans": [] # List of tuples (plan_type, root_node, cost)
}
```

Fig 16 For the chosen and alternative query plan and placed them all together in a tuple

Assumption: We only chose 6 different permutations for the AQP . However, this might not be the case for the comparisons done by Postgres since the scans, and joins can be randomly combined and permutated. However, it would be practically impossible for us to compute all the combinations manually. Hence, our team decided to choose these 6 different AQPs based on what was taught in lecture which would allow us to explain our comparisons.

3.4 Preparation before passing to Annotations (Algorithm)

Lastly, we would pass the optimal query plans tree node into the preAnnotation function (Fig 17). This would be used to compare **each operator** of the **optimal query plan** to the **respective alternative query plan**. For example, a join operator would be compared to the join alternative plans (Namely: Merge Join, Hash Join, Index Nested Loop Join). The ratio will be calculated (Total cost of Respective Alternate Query Plan / Total cost of optimal query plan) to find out how much costlier the alternative plans are compared to the original query plan for that particular operator. This would then be stored in the dictionary called alternate plans.

```
def preAnnotation(self, optimalTreeRoot):
    queue = []
    queue.append(optimalTreeRoot)
    while len(queue) != 0:
        childlength = len(queue)
        # Add appropriate children ndoes
        for i in range(childlength):
            node = queue.pop(0)
            if "Join" in node.attributes['Node Type'] or "Loop" in node.attributes['Node Type']:
                for alternate_nodes in self.query_plans["alternative_plans"]:
                    if ("Join" in alternate_nodes[0] or "Loop" in alternate_nodes[0]) and (alternate_nodes[0] != node.attributes['Node Type']):
                        if (alternate_nodes[2] > self.query_plans["chosen_plan"][2]):
                            # total alternate plan cost / total optimal plan cost
                            node.alternate_plans[alternate_nodes[0]] = alternate_nodes[2]/self.query_plans["chosen_plan"][2]
            elif "Scan" in node.attributes['Node Type']:
                for alternate_nodes in self.query_plans["alternative_plans"]:
                    if "Scan" in alternate_nodes[0] and (alternate_nodes[0] != node.attributes['Node Type']):
                        if (alternate_nodes[2] > self.query_plans["chosen_plan"][2]):
                            # total alternate plan cost / total optimal plan cost
                            node.alternate_plans[alternate_nodes[0]] = alternate_nodes[2]/self.query_plans["chosen_plan"][2]
            for child in node.children:
                queue.append(child)
```

Fig 17 Function preAnnotation

Assumption: We took the total cost of an alternate query plan instead of comparing the cost of each operator of an alternate query plan to the respective operator in the optimal query plan. (EG: Compare the cost of a join operator to the cost of another join operator in the other plan). This is because each alternate query plan structure may change from the optimal query plan depending on

the joins or scans used, hence making it very difficult to generalize a solution that is able to match both the optimal query plan and the alternative query plan operator by operator. Hence we used the next best solution which is to compare it based on the total cost of the plan.
For example for merge join, we will be able to **estimate** how much costlier it would be if we used index nested loop join or hash join by comparing the optimal query plans cost with the total cost of using only index nested loop join for the entire plan or using only hash join for the entire plan.

4. Annotations

To add annotations to the QEP, the traverseTree() method in Fig 18 takes in the root node of the tree plan first and traverse through the tree from root to leaf. At each node, the generateAnnotation() method is called to append annotations according to the operator at each node. Each annotation contains a brief description of the operation at each node.

```
def traverseTree(self, root):
    if not root:
        return

    self.generateAnnotation(root)

    if root.children:
        for child in root.children:
            self.traverseTree(child)
```

Fig 18 Function traverseTree()

The generateAnnotation() method in Fig 19 consists of multiple case statements depending on the node types of the operations. For example, the node type for Sequential Scan would be “Seq Scan”. After finding the correct case statement for the current operation, an annotation specific to that particular operation would be generated. Relevant information from each node, if available, is also extracted and displayed in the annotation. For the scans, we have included the name of the table that we are scanning, using the attribute ‘Relation Name’. For Index Scan, we have extracted the attribute ‘Index Cond’, and added it into our annotation to display the condition used to find the locations of rows from the index.

```
def generateAnnotation(self, node):
    nodeType = node.attributes['Node Type']
    annotation = ""

    # For scans
    if nodeType == "Seq Scan":
        table = node.attributes['Relation Name']
        node.annotations += "Sequential scan is used to read the {} table, because there is either no index available,
            "or the result is about the same size as the tree, which would cause an Index Scan to take longer".format(
        node.annotations += ".\n"
        self.comparison(node)

    if nodeType == "Index Scan":
        table = node.attributes['Relation Name']
        node.annotations += "Index scan is used to read the {} table, because there is an index available " \
            "and it would be faster compared to Sequential Scan".format(table)
        if "Index Cond" in node.attributes:
            cond = node.attributes['Index Cond']
            cond = self.formatCondition(cond)
            node.annotations += " with conditions {}".format(cond)
        node.annotations += ".\n"
        self.comparison(node)
```

Fig 19 Function generateAnnotation

As can be seen in Fig 20, for the joins, we have added the names of the 2 tables being joined, as well as the join conditions after some formatting, taken from the attributes ‘Hash Cond’ and ‘Merge Cond’. Afterwhich, we append the comparison with other joins to the back of the annotation.

```
# For joins
if nodeType == "Hash Join":
    cond = node.attributes['Hash Cond']
    try:
        cond = self.formatCondition(cond)

        condsplit = cond.split('.')
        table1 = condsplit[0]
        condsplit2 = condsplit[1].split('=')
        table2 = condsplit2[1]
        annotation = "Hash join is performed on tables {} and {}, with the conditions {}.\n".format(table1, table2
    except:
        annotation = "Hash join is performed with the conditions {}.\n".format(cond)
    node.annotations += annotation
    self.comparison(node)

if nodeType == "Merge Join":
    cond = node.attributes['Merge Cond']
    try:
        cond = self.formatCondition(cond)

        condsplit = cond.split('.')
        table1 = condsplit[0]
        condsplit2 = condsplit[1].split('=')
        table2 = condsplit2[1]
        annotation = "Merge join is performed on tables {} and {}, with the conditions {}.\n".format(table1, table
    except:
        annotation = "Merge join is performed with the conditions {}.\n".format(cond)
    node.annotations += annotation
    self.comparison(node)
```

Fig 20 Joining of two Tables

Apart from scans and joins, for other operations, we have briefly explained what the operation does and included relevant information, if available. This can be seen in Fig 21 below, where the attribute ‘Sort Key’ is added for the sort.

```
if nodeType == "Sort":
    annotation = "Sort the table on keys "
    desc = False
    for key in node.attributes['Sort Key']:
        if "DESC" in key:
            desc = True
        else:
            annotation += f"{key}, "
    annotation = annotation[:-2]
    if desc:
        annotation += ' in a decremental manner.'
    else:
        annotation += ' in an incremental manner.'
    annotation += '\n\n'
    node.annotations += annotation

if nodeType == "Unique":
    annotation = "Duplicates are removed from the table.\n"
    node.annotations += annotation

if nodeType == "Gather":
    annotation = "Gather operation is performed on the table.\n"
    node.annotations += annotation

if nodeType == "Gather Merge":
    annotation = "Gather Merge operation is performed, combining the output of child nodes.\n"
    node.annotations += annotation
```

Fig 21 Attribute ‘Sort Key’ added

If the operation is a scan or a join, the comparison() method is called to append to the annotation about the performance comparison of each available alternative scan or join (Fig 22). The cost of each alternative plan is displayed in terms of its ratio to that of the chosen algorithm. comparison() first checks if an alternative plan exists, and if there is at least one alternative plan, it loops through alternate_plans and retrieves the plan name and cost. Otherwise, it is assumed that the operation is done throughout all query plans.

```
def comparison(self, node):
    if len(node.alternate_plans) != 0:
        for altScan in node.alternate_plans:
            annotation = f"Compared to {node.attributes['Node Type']}, {altScan} is "\
                         f"{node.alternate_plans.get(altScan):.2f} times as expensive.\n"
            node.annotations += annotation
    else:
        annotation = f"{node.attributes['Node Type']} is used across all AQPs."
        node.annotations += annotation
```

Fig 22 Function comparison()

Three utility methods (Fig 23 to 25) were created to assist us in formatting the annotations. Firstly, getKeys() check for the keys used by the operator, and if any key is used, it returns the keys in the form of a single string. Secondly, getTables() receives a string containing the names of the tables to be output and returns the table names separately as two strings. Lastly, formatCondition() takes in a string containing several conditions and reformat it into a more presentable string.

```
def getKeys(self, node):
    group_key = []
    if "Group Key" in node.attributes:
        group_key = node.attributes['Group Key']

    list_of_keys = []
    for key in group_key:
        list_of_keys.append(key)
    stringOfKeys = ', '.join(list_of_keys)
    if len(stringOfKeys) > 1:
        stringOfKeys = ', '.join(list_of_keys)
    else:
        stringOfKeys = list_of_keys[0]
    return stringOfKeys
```

Fig 23 Function getKeys()

```
def getTables(self, cond):
    cond = cond.replace(" ", "")
    cond = cond.lstrip("(").rstrip(")")
    condsplit = cond.split('.')
    table1 = condsplit[0]
    condsplit2 = condsplit[1].split('=')
    table2 = condsplit2[1]
    return table1, table2
```

Fig 24 Function getTables()

```
def formatCondition(self, cond):
    cond = cond.replace(" ", "")
    cond = cond.replace(")AND(", ", ")
    cond = cond.replace(")OR(", ", ")
    cond = cond.replace("(", "")
    cond = cond.replace(")", "")
    return cond
```

Fig 25 Function formatCondition()

Assumptions: Due to the limited information that we could retrieve from the json using the EXPLAIN statement, we were not able to pinpoint reasons as to why an index or join is chosen over the others besides the use of cost. However, we did try to list possibilities for some of the operators. For example, for Sequential Scans, we wrote that it was chosen because of the possibility

that there is either no index available or the result is about the same size as the tree, which would cause an Index Scan to take longer.

5. Test/Examples

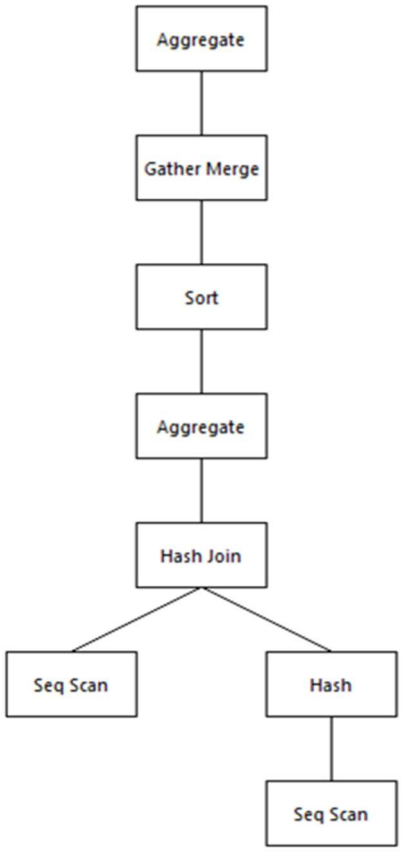
To ensure that our application works and that it is displaying the correct outputs on the GUI we used a range of different queries. We have chosen to show 5 queries in this report to showcase our working application. We traversed from the bottom-up as well as from the left to right for the output below starting from the scans.

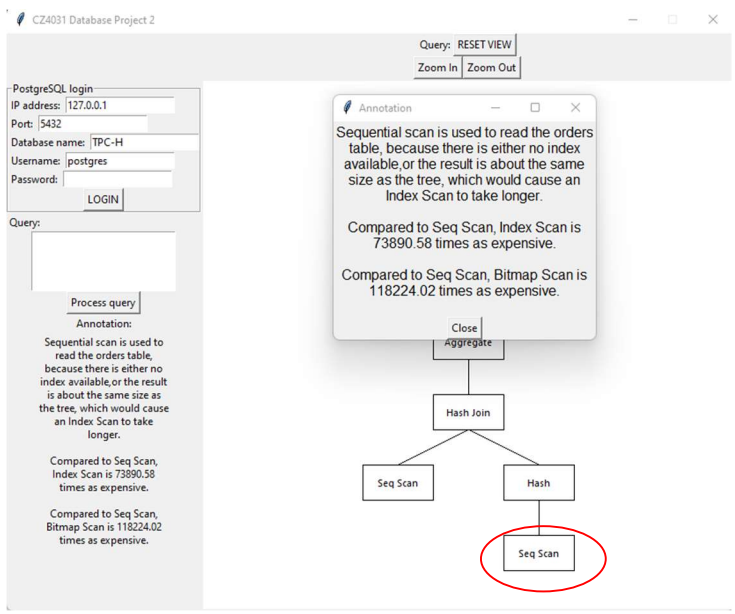
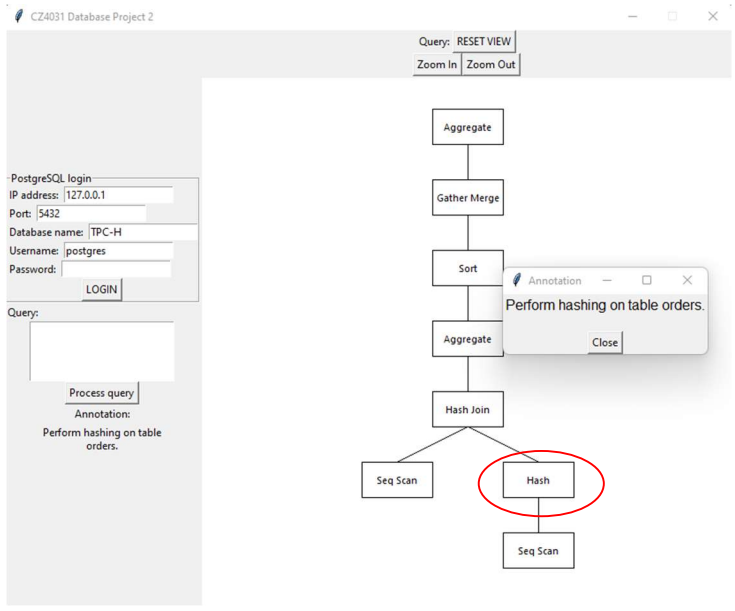
5.1 Query 1

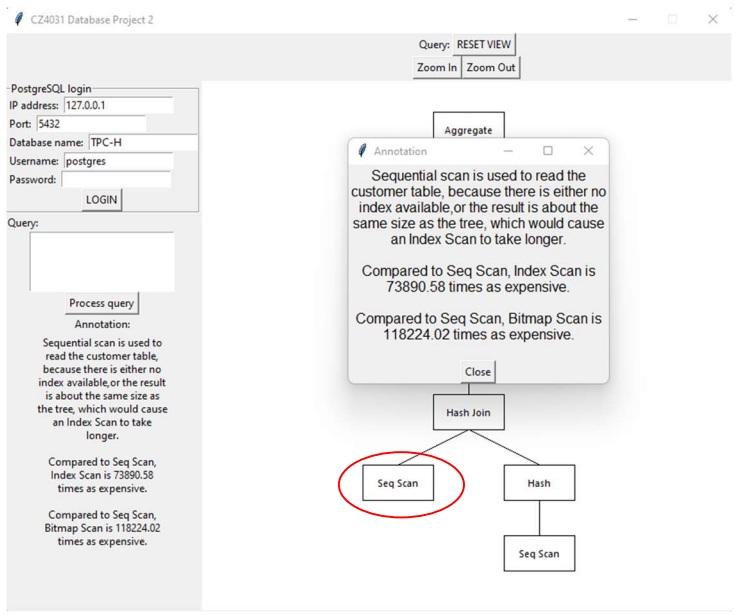
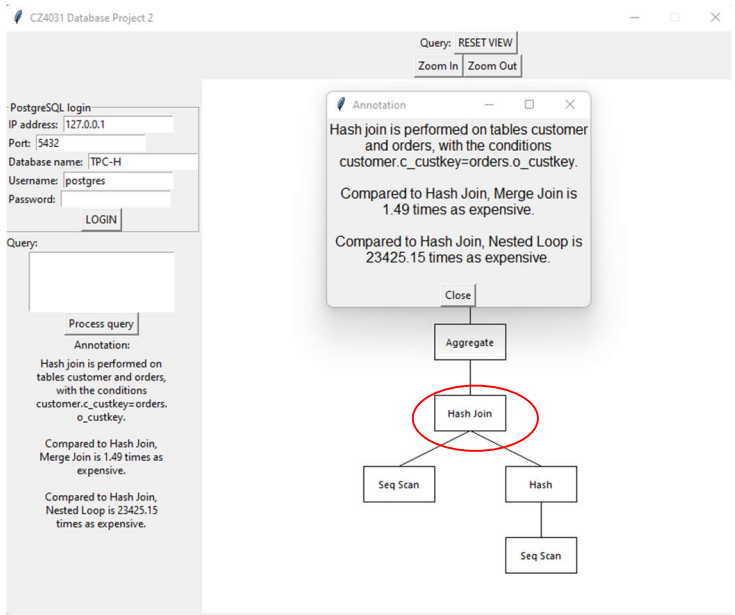
Query:

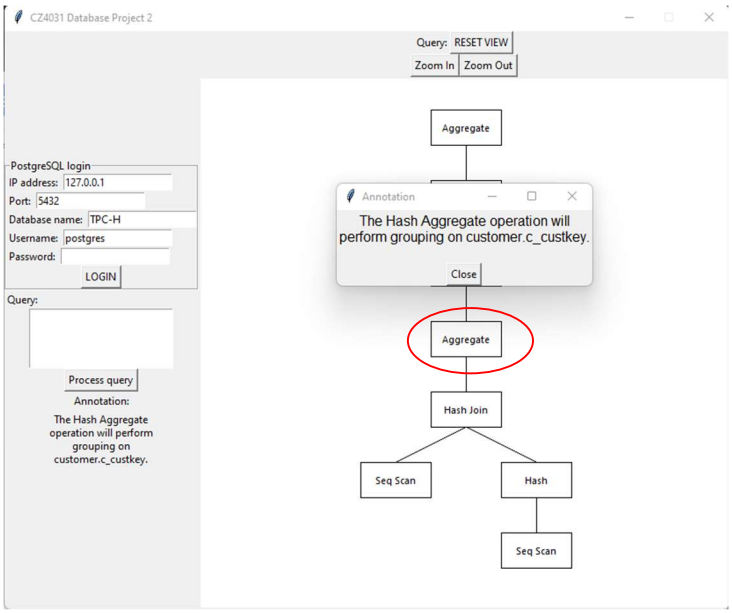
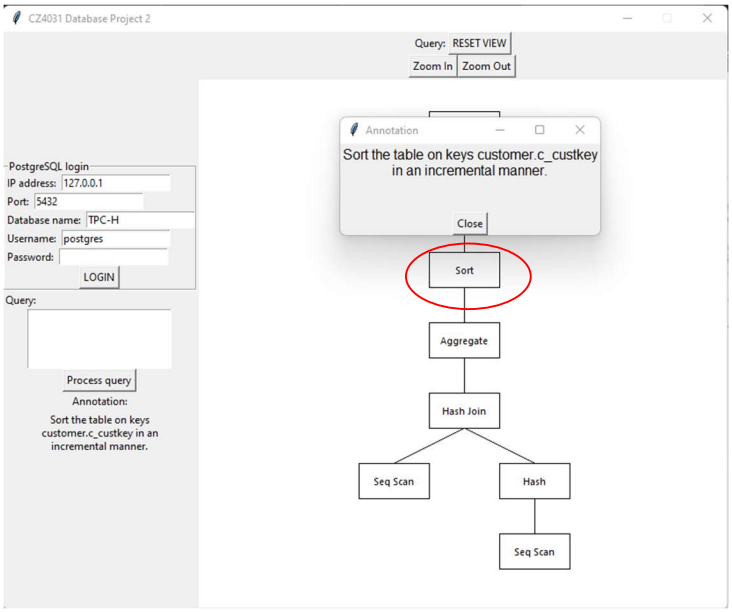
```
select
        c_custkey,
        count(o_orderkey)
from
        customer left outer join orders on
                c_custkey = o_custkey
                and o_comment not like '%pending%packages%'
group by
        c_custkey
```

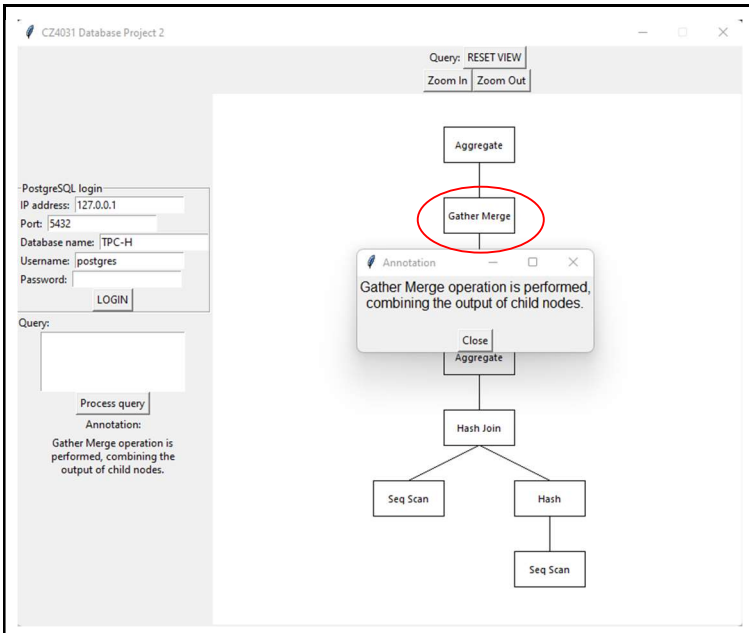
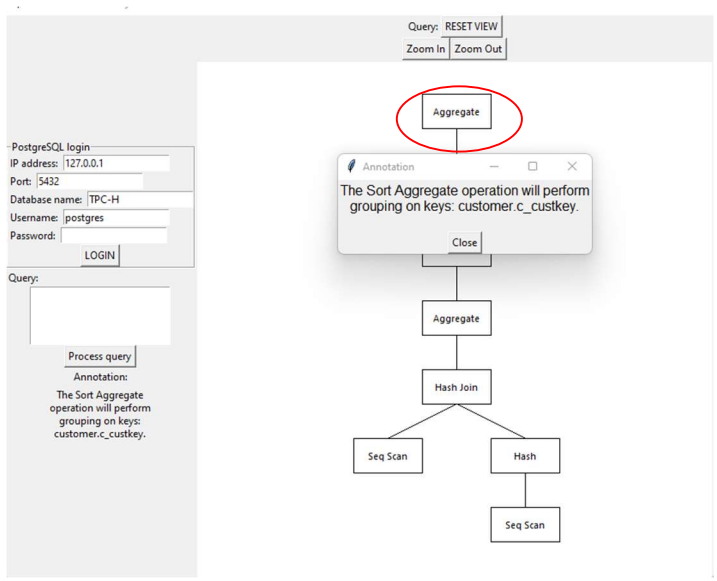
Output:

Overall Query Plan


	Sequential Scan is used to read the orders table, because there is either no index available, or the result is about the same size as the tree, which would cause an index scan to take longer Compared to Seq Scan, Index Scan is 73890.58 times as expensive. Compared to Seq Scan, Bitmap Scan is 118224.02 times as expensive
	Performing hashing on table orders

	Sequential scan is used to read the customer table, because there is either no index available, or the result is about the same size as the tree, which would cause an index Scan to take longer Compared to Seq Scan,Index Scan is 73890.58 times as expensive Compared to Seq Scan, Bitmap Scan is 118224.02 times as expensive
	Hash join is performed on tables customer and orders, with the conditions customer.c_custkey=orders.o_custkey Compared to Hash Join, Merge Join is 1.49 times as expensive Compared to Hash Join, Nested Loop is 23425.15 times as expensive

	The Hash Aggregate operation will perform grouping on customer.c_custkey
	Sort the table on keys customer.c_custkey in an incremental manner

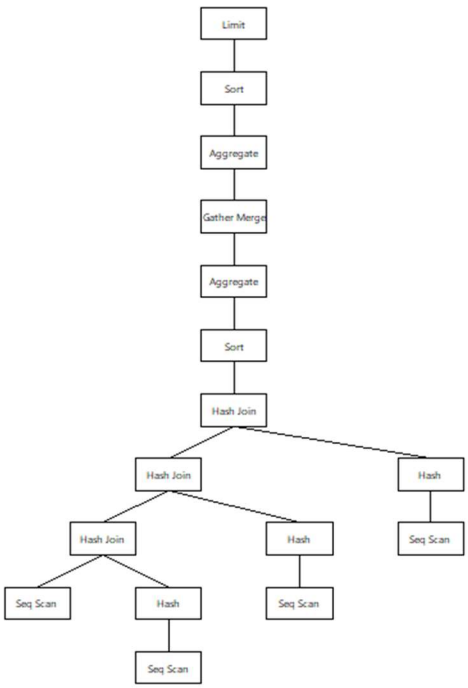
	Gather Merge operation is performed, combining the output of child nodes
	The sort Aggregate operation will perform grouping on keys customer.c_custkey

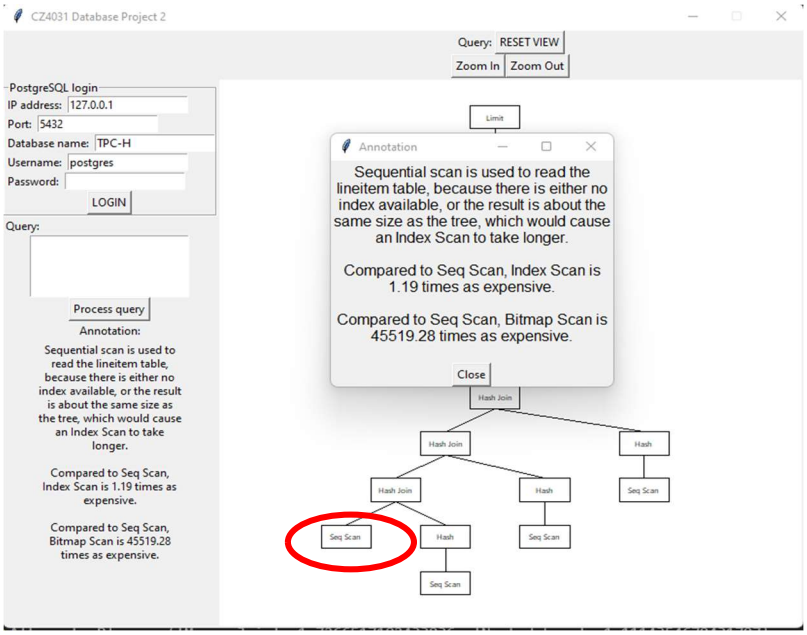
5.2 Query 2

Query:

```
select
        c_custkey,
        c_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        c_acctbal,
        n_name,
        c_address,
        c_phone,
        c_comment
from
        customer,
        orders,
        lineitem,
        nation
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate >= date '1995-01-01'
        and o_orderdate < date '1995-01-01' + interval '3' month
        and l_returnflag = 'R'
        and c_nationkey = n_nationkey
group by
        c_custkey,
        c_name,
        c_acctbal,
        c_phone,
        n_name,
        c_address,
        c_comment
order by
        revenue desc
limit 1;
```

Output:

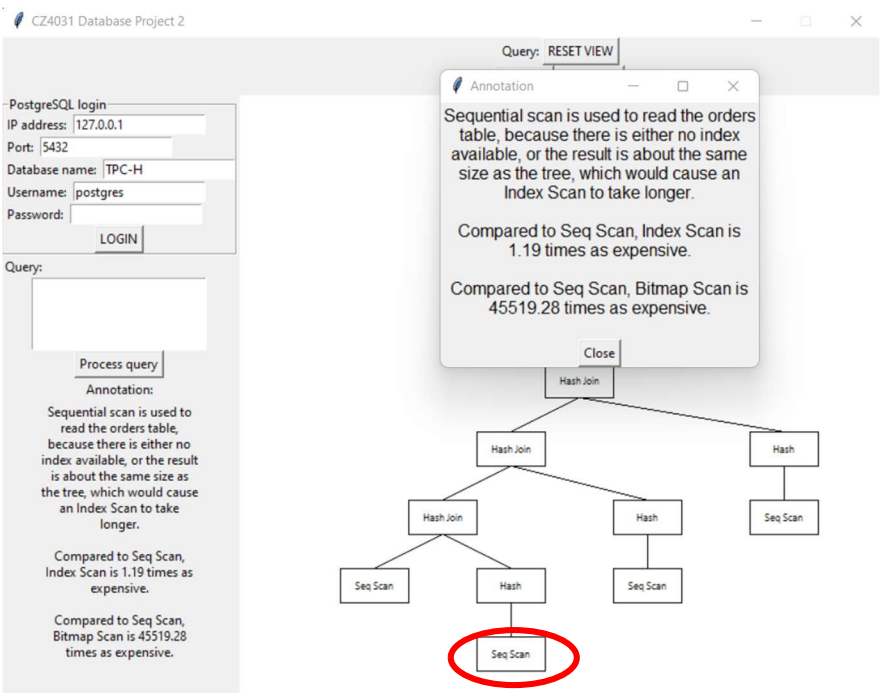
Overall Query Plan




Sequential scan is used to read the lineitem table, because there is either no index available, or the result is about the same size as the tree, which would cause an Index Scan to take longer

Compared to Seq Scan, Index Scan is 1.19 times as expensive

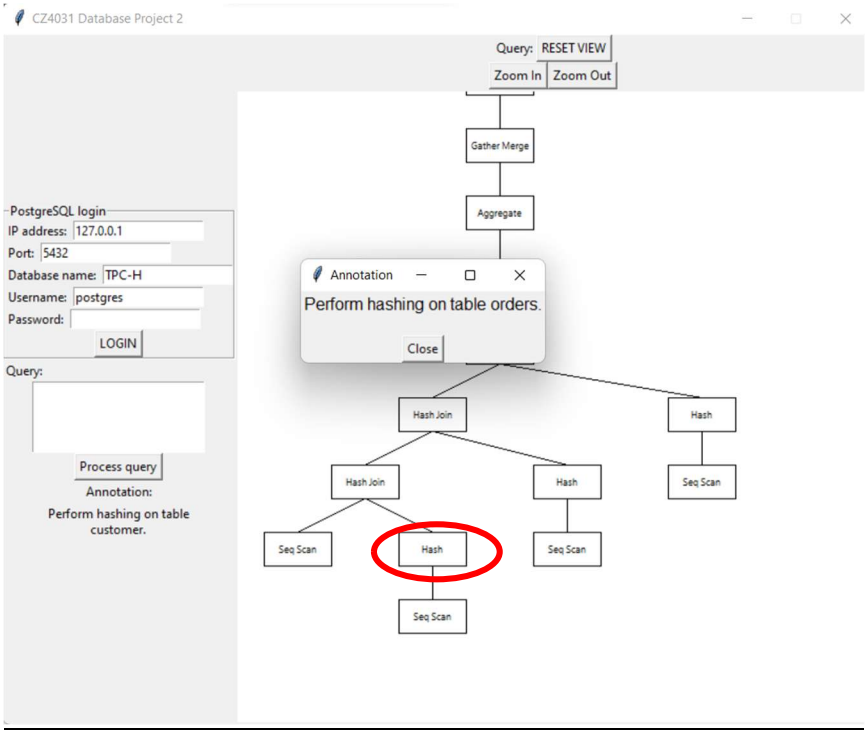
Compared to Seq Scan,Bitmap Scan is 45519.28 times as expensive



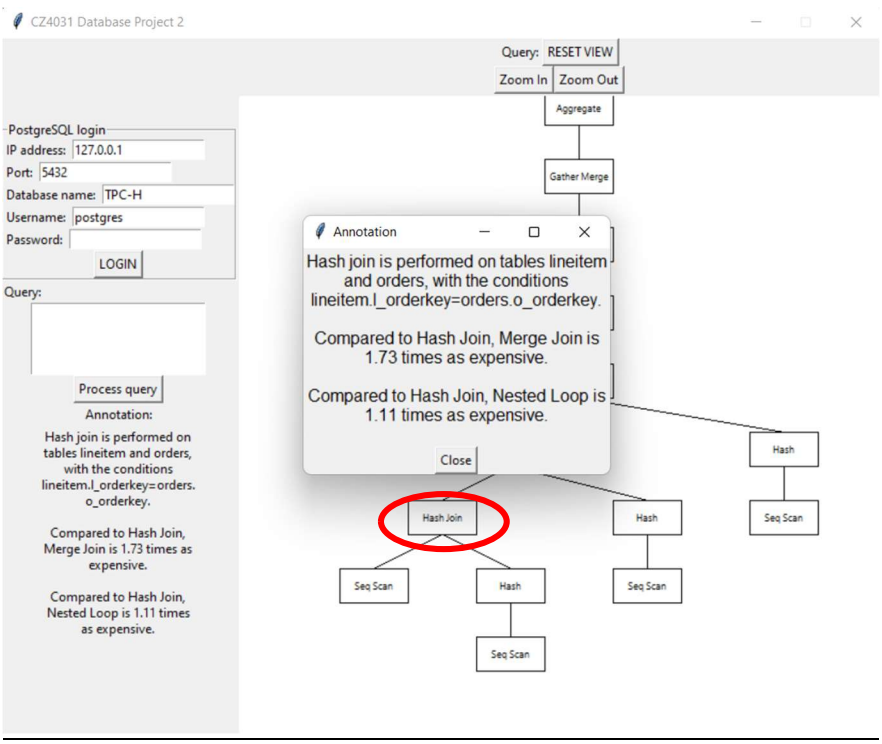
Sequential scan is used to read the nation table,because there is either no index available, or the result is about the same size as the tree, which would cause an Index Scan to take longer

Compared to Seq Scan, Index Scan is 1.19 times as expensive

Compared to Seq Scan,Bitmap Scan is 45519.28 times as expensive



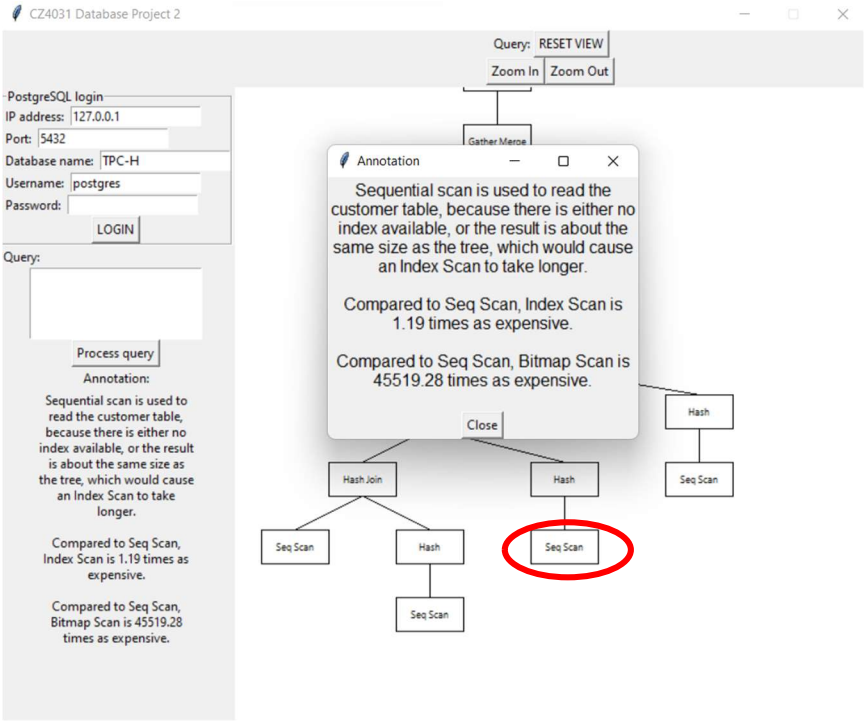
Perform Hashing on table orders



Hash join is performed on tables customer and nation, with the conditions customer.c_nationkey=nation.n_nationkey

Compared to Hash Join, Merge join is 1.73 times as expensive

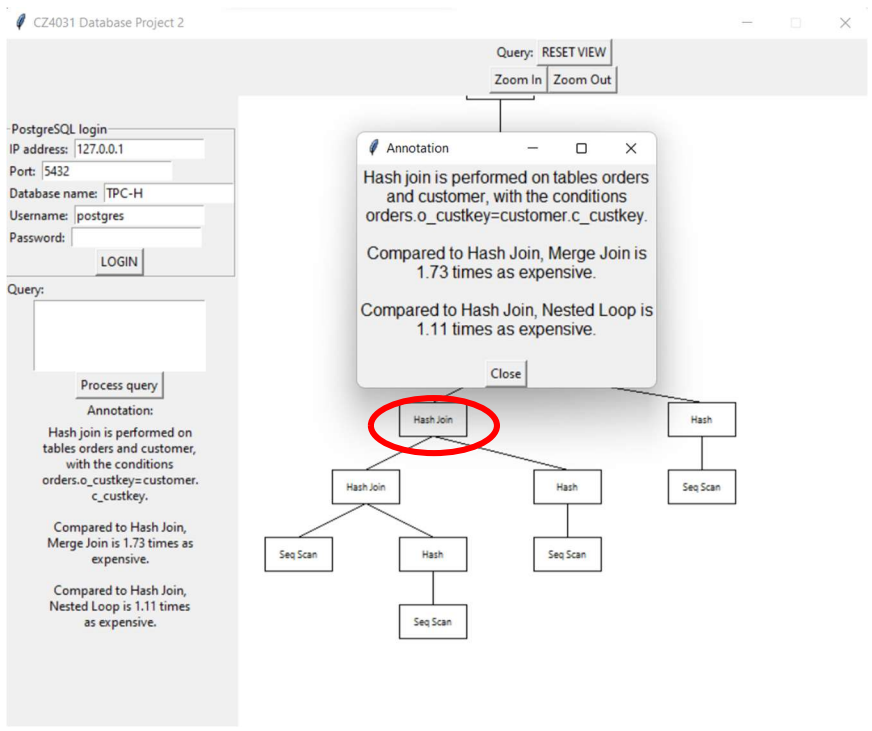
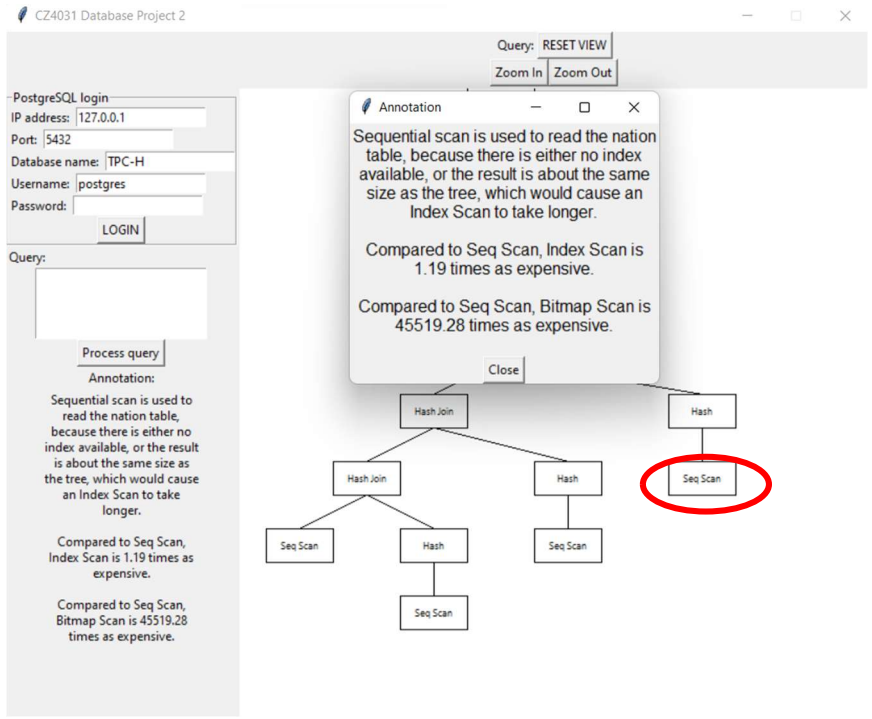
Compared to Hash Join, Nested Loop is 1.11 times as expensive

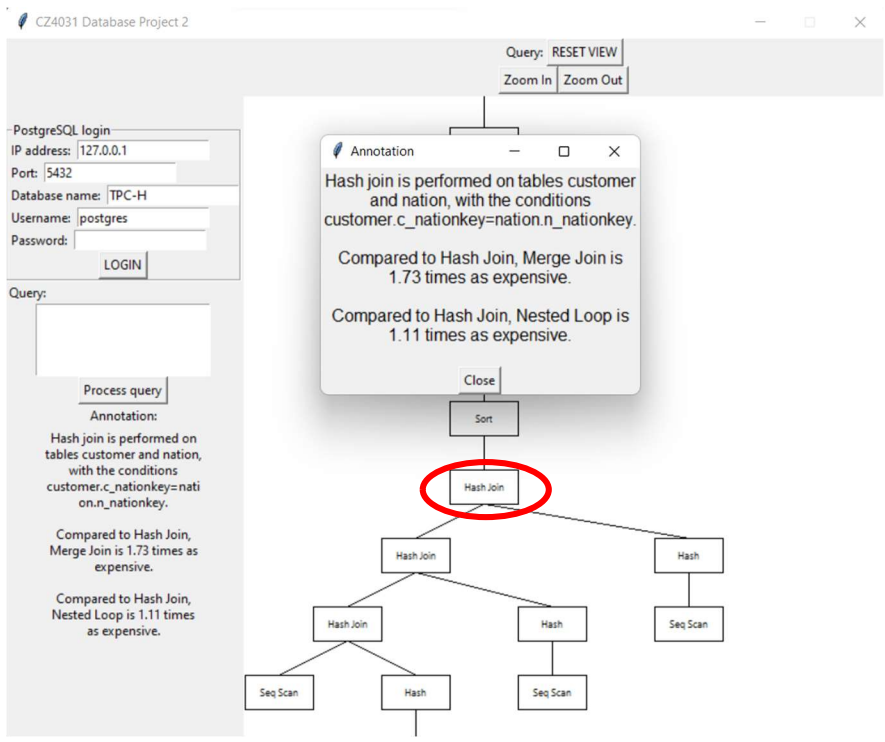
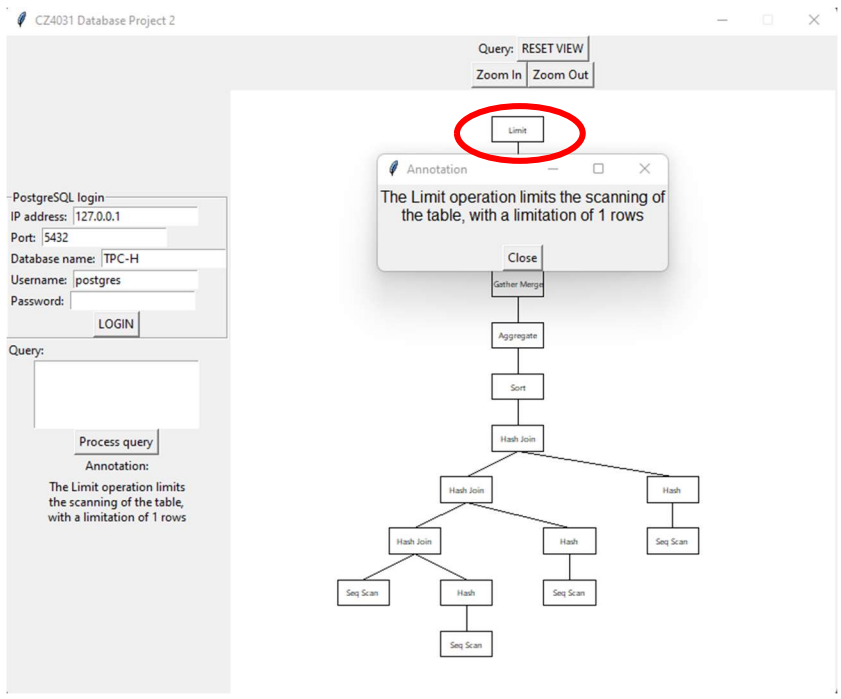


Sequential Scan is used to read the customer table, because there is either no index available, or the results is about the same size as the tree, which would cause an index Scan to take longer.

Compared to Seq Scan, index Scan is 1.19 times as expensive.

Compared to Seq Scan, Bitmap Scan is 45519.28 times as expensive

	Hash join is performed on tables orders and customer, with the conditions order.o_custkey=customer.c_custkey Compared to Hash Join, Merge Join is 1.73 times as expensive. Compared to Hash Join, Nested Loop is 1.11 times as expensive
	Sequential scan is used to read the nation table, because there is either no index available, or the results is about the same size as the tree, which could cause an Index Scan to take longer Compared to Seq Scan, Index Scan is 1.19 times as expensive Compared to Seq Scan, Bitmap Scan is 45519.28 times as expensive

	Hash join is performed on tables customer and nation, with the conditions customer.c_nationkey=nation.n_nationkey Compared to Hash Join, Merge Join is 1.73 times as expensive Compared to Hash Join, nested loop is 1.11 times as expensive
	The limit operation limits the scanning of the table, with a limitation of 1 rows

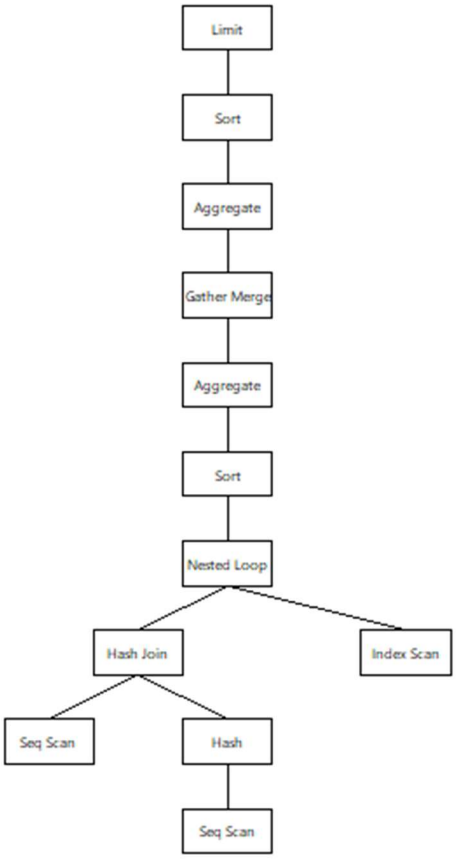
5.3 Query 3

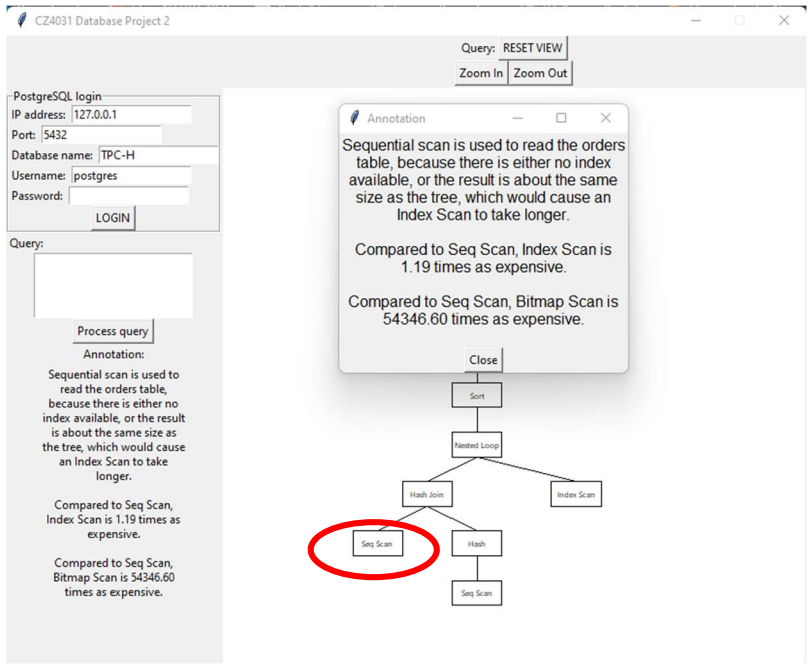
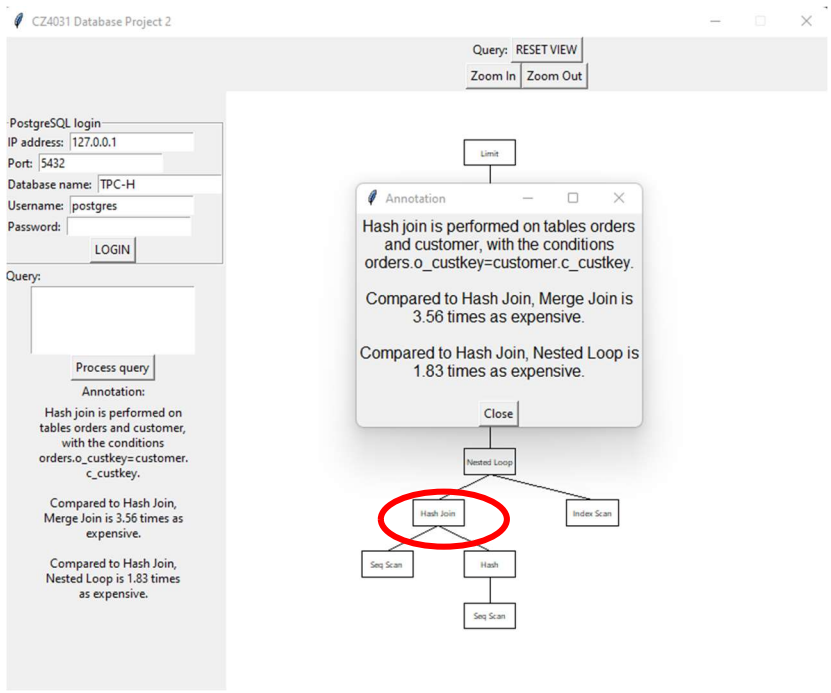
Query:

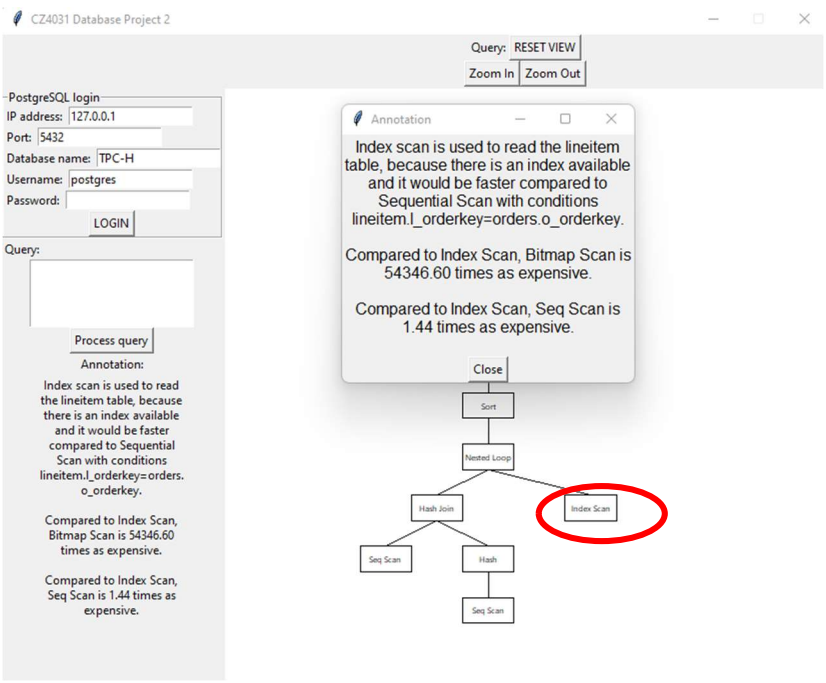
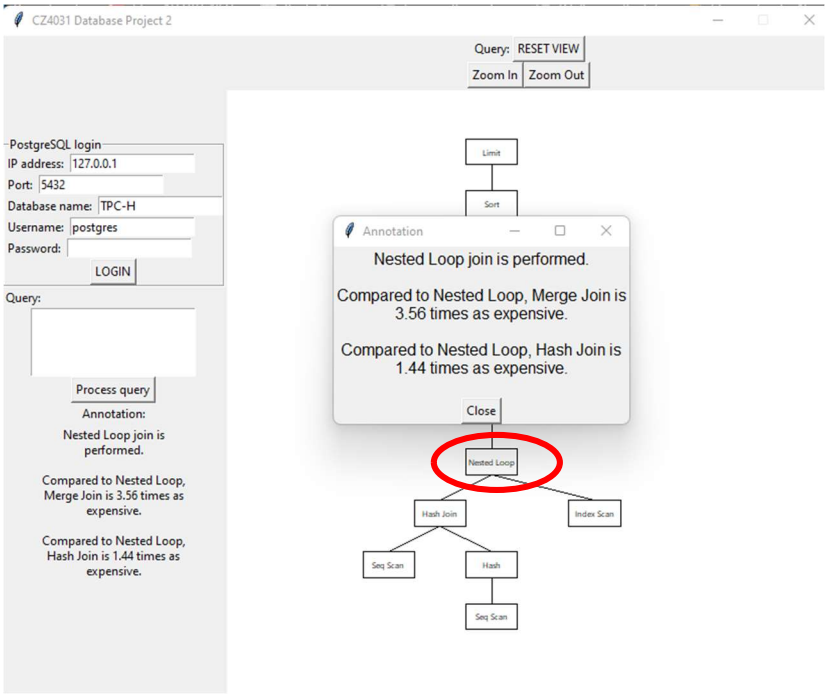
```
select
        l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = 'HOUSEHOLD'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey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

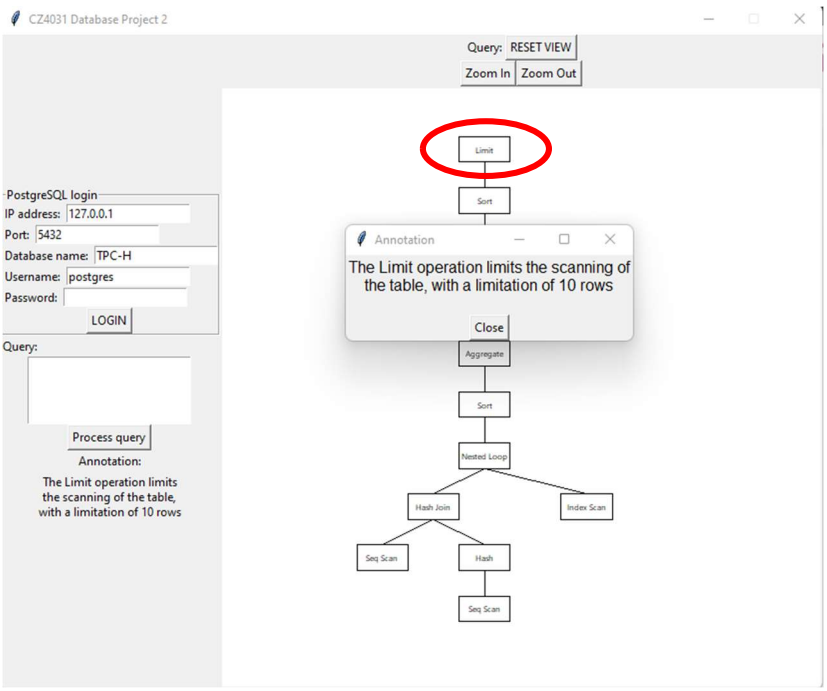
Output:

Overall Query Plan



	Sequential scan is used to read the orders table,because there is either no index available, or the result is about the same size as the tree, which would cause an Index Scan to take longer Compared to Seq Scan, Index Scan is 1.19 times as expensive Compared to Seq Scan,Bitmap Scan is 54346.60 times as expensive
	Hash join is performed on tables orders and customer,with the conditions orders.o_custkey=customer.c_custkey Compared to Hash Join, Merge Join is 3.56 times as expensive Compared to Hash Join,Nested Loop is 1.83 times as expensive

	Index Scan is used to read the lineitem table, because there is an index available and it would be faster compared to Sequential Scan with conditions lineitem.l_orderkey=orders.o_orderkey Compared to index Scan, Bitmap Scan is 54346.60 times as expensive Compared to Index Scan, Seq Scan is 1.44 times as expensive
	Nested Loop join is performed Compared to Nested Loop, Merge Join is 3.56 times as expensive. Compared Nested Loop, Hash Join is 1.44 times as expensive



The limit operation limits the scanning of the table, with a limitation of 10 rows

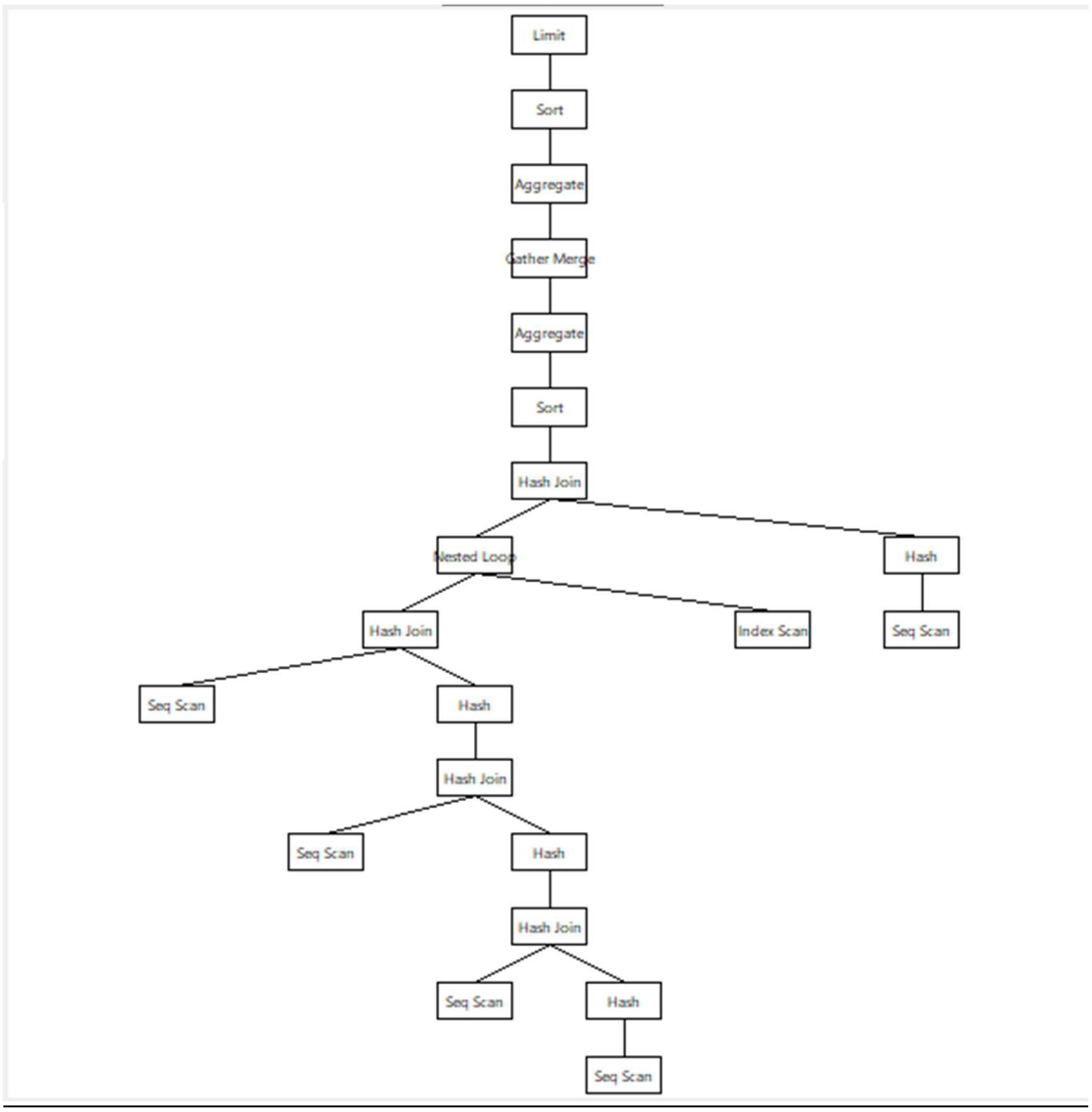
5.4 Query 4

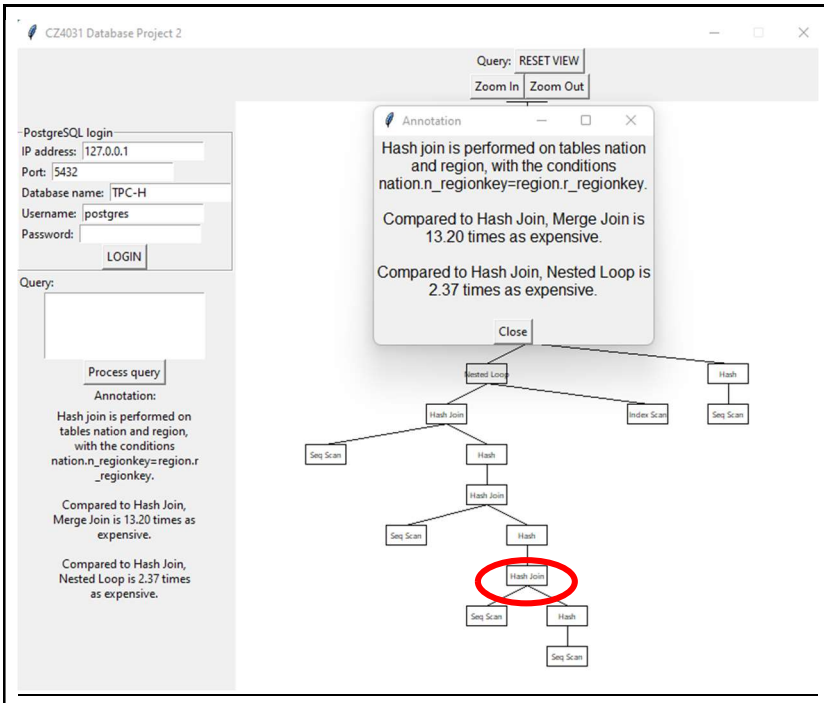
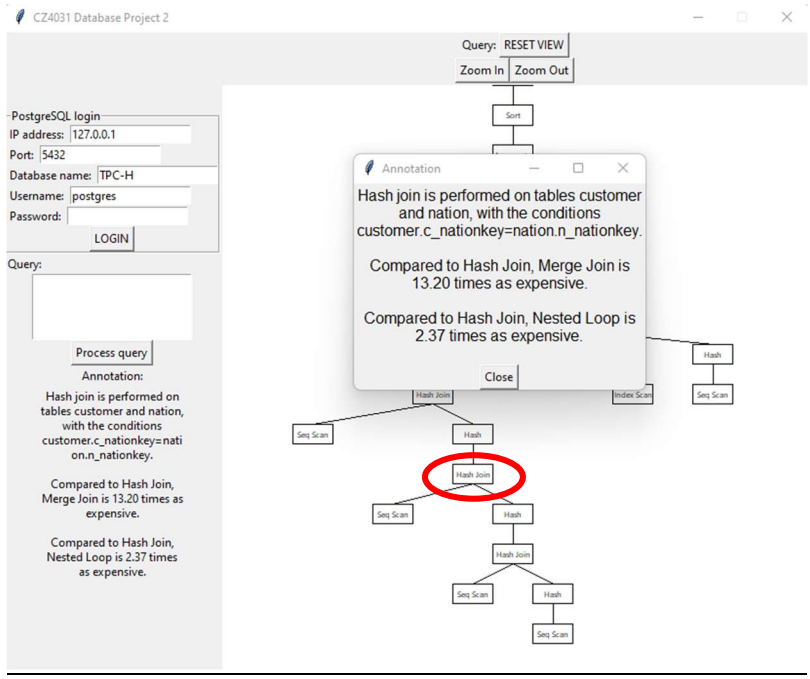
Query:

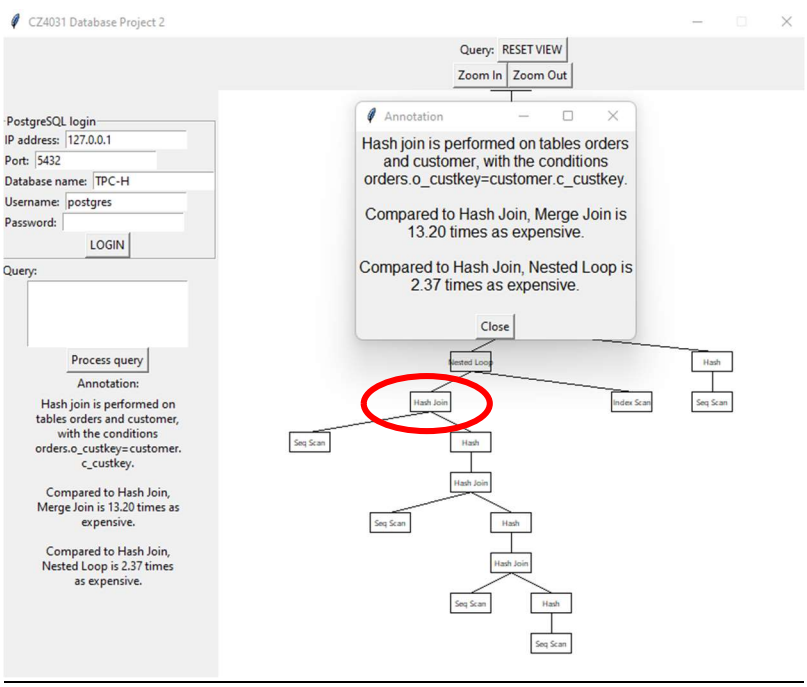
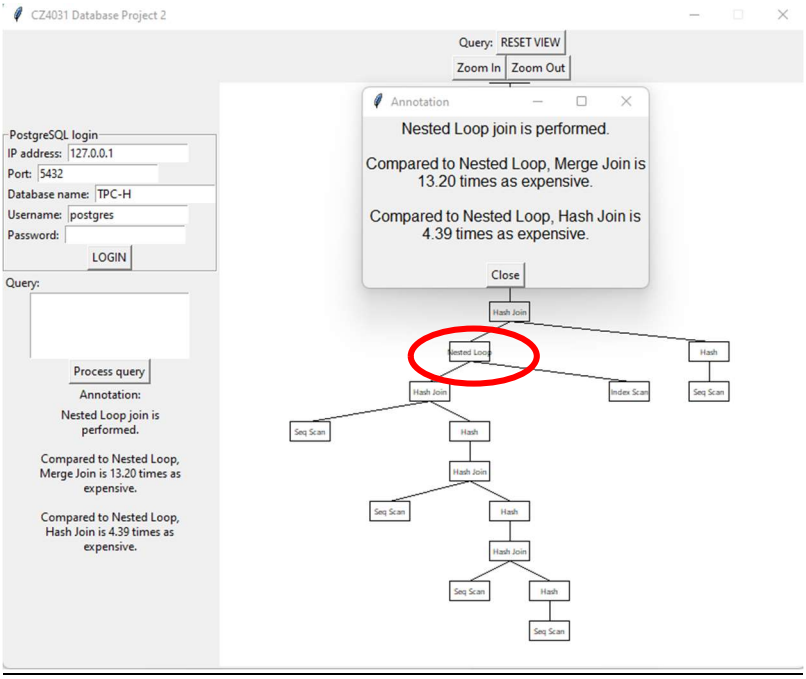
```
select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer,
        orders,
        lineitem,
        supplier,
        nation,
        region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'AMERICA'
        and o_orderdate >= date '1995-01-01'
        and o_orderdate < date '1995-01-01' + interval '1' year
group by
        n_name
order by
        revenue desc
limit 1;
```

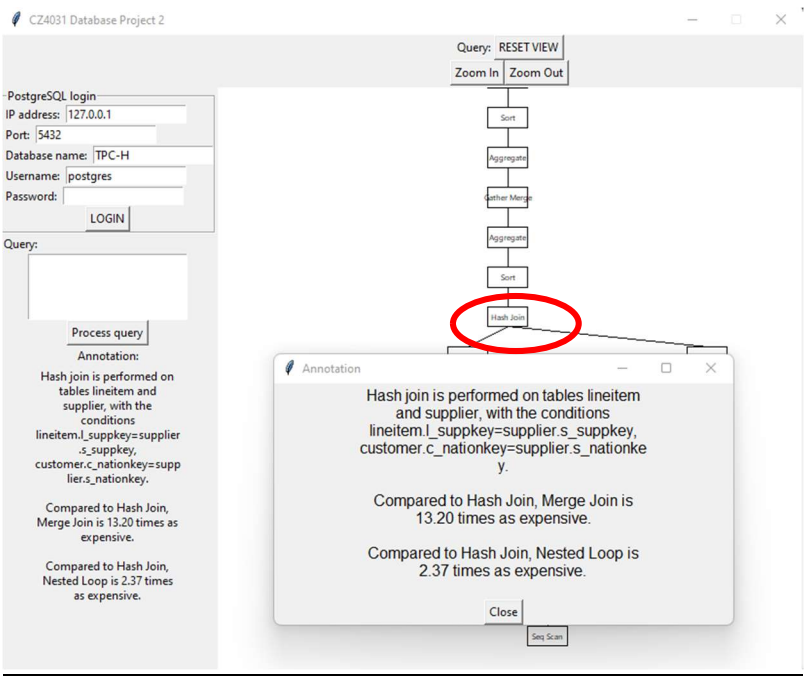
Output:

Overall Query Plan



	Hash join is performed on tables nation and region with the conditions nation.n_regionkey=region.r_regionkey Compared to Hash Join, Merge Join is 13.20 times as expensive Compared to Hash Join, Nested Loop is 2.37 times as expensive
	Hash join is performed on tables customer and nation,with the conditions customer.c_nationkey=nation.n_nationkey. Compared to Hash Join,Merge Join is 13.20 times as expensive Compared to Hash Join, Nested Loop is 2.37 as expensive

	Hash join is performed on tables customer and nation,with the conditions customer.c_nationkey=nation.n_nationkey. Compared to Hash Join,Merge Join is 13.20 times as expensive Compared to Hash Join, Nested Loop is 2.37 as expensive
	Nested Loop join is performed. Compared to Nested Loop, Merge Join is 13.20 times as expensive Compared to Nested Loop, Hash Join is 4.39 times as expensive



Hash Join is performed on tables lineitem and supplier, with the conditions lineitem.i_suppkey=supplier.s_suppkey,customer.c_nationkey=supplier.s_nationkey.

Compared to Hash join, Merge Join is 13.20 times as expensive

Compared to Hash Join, Nested Loop is 2.37 times as expensive

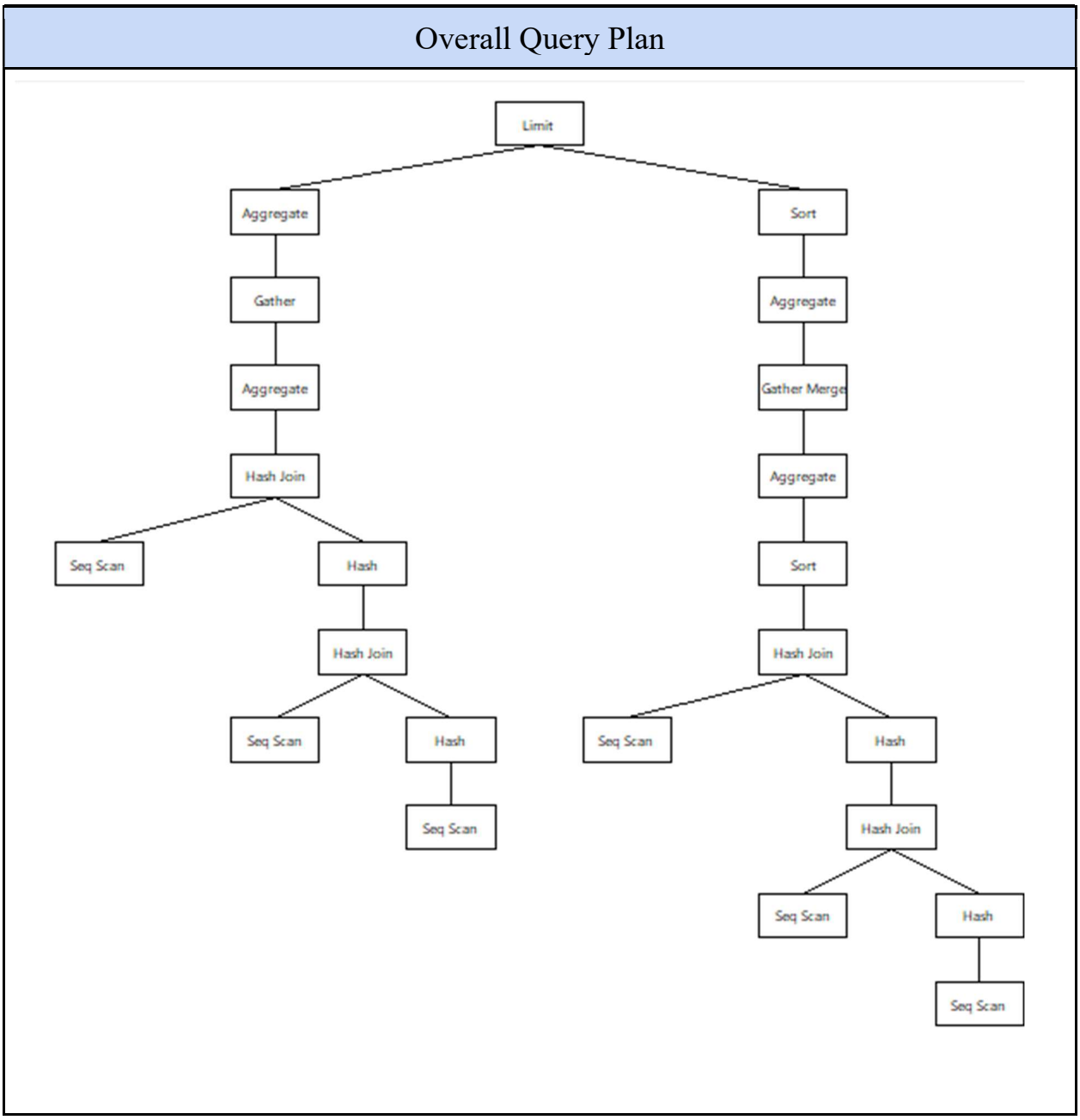
5.5 Query 5

Query:

```
select
        ps_partkey,
        sum(ps_supplycost * ps_availqty) as value
from
        partsupp,
        supplier,
        nation
where
        ps_suppkey = s_suppkey
        and s_nationkey = n_nationkey
        and n_name = 'SAUDI ARABIA'
group by
        ps_partkey having
                sum(ps_supplycost * ps_availqty) > (
                        select
                                sum(ps_supplycost * ps_availqty) * 0.0001000000
```

```
                from
                        partsupp,
                        supplier,
                        nation
                where
                        ps_suppkey = s_suppkey
                        and s_nationkey = n_nationkey
                        and n_name = 'SAUDI ARABIA'
        )
order by
        value desc
limit 1;
```

Output:


Overall Query Plan
Limit
Aggregate
Gather
Aggregate
Hash Join
Seq Scan
Hash
Hash Join
Seq Scan
Hash
Seq Scan
Sort
Aggregate
Gather Merge
Aggregate
Sort
Hash Join
Seq Scan
Hash
Hash Join
Seq Scan
Hash
Seq Scan

6. Limitations

There are a few limitations which make it difficult to perform an analysis on the Optimal QEP.

Firstly, it was difficult to generate AQPs as PostgreSQL allowed us to only disable a type of operation to generate them. As stated above, due to many permutations and combinations, our analysis was limited. This is because of the following:

- Different joins have differing costs. Influencing the type of joins used can possibly change the logical query plan, and this has been observed. Hence, comparing between joins is tricky. This limits the accuracy of our analysis since the changes in cost is a combined effect of multiple joins changing cost.

- Scan methods have been observed to also influence the logical query plan. Changing of the scans used also changes the joins used. In the case that the structure is not changed, the comparison of the cost of using two different methods of scan would be more accurately estimated. However, due to these changes in structure, if we were to compare it node by node, this will understate the actual effect on the cost. At the same time, due to the fact that we have changed the scan methods of every other scans and not just localised it to one part of the query, trying to compare the total cost of the two query plans is also inaccurate as this factors in the differences in cost of another part of the query that has changed, but not because of the particular operation we are looking at.

In addition, we found it difficult to properly compare the costs of joins. This is because the true cost of joins is not just limited to the cost stated in the node of the QEP, but also other operators. For example, hash join requires one of the tables to be hashed first, this means there is an operator called “Hash” as one of the child nodes. While for some joins this is easy, others are not so straightforward. As such, the values provided may be a good estimate of the relative costs of using another method, but are not accurate in factoring out the differences in the actual cost of execution and the cost PostgreSQL estimated.

7. Improvement

An improvement to the algorithm used for comparison could be to decrease the tree size accordingly to match the larger tree plan to the smaller tree plan. This can be done by combining the costs of certain operators together that are caused by the join methods used. For example, hash join could be associated with the hash operator that merge join does not use. Hence, for non-common operators of the two join, the cost could be combined together. This would give us two trees that match and we would then be able to do an operator by operator comparison which would give us a better cost estimate.