NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ4031 Database System Principles

Project 2 Report

Group 48

Team members

Lee Jun Wei (U1922896C)

Lim Kai Sheng (U2020321C)

Ma JiaXin (U2022011J)

Yeow Ying Sheng (U1722395B)

Zhu Zeyu (N2202577F)

All members contributed equally to this project

Table of Content

1. Introduction

1.1 Project Objective

This project aims to design and implement a user-friendly GUI application to retrieve and display information about the Query Execution Plan (QEP) and Alternative Query Plans (AQP) from a PostgreSQL database based on a user's SQL query. And annotate the corresponding SQL query to explain how the underlying query processor executes different components of the query.

1.2 Technologies

Since the project has to be completed with Python as the host language, we have decided to utilize the Flask web framework. Flask utilizes the Jinja templating engine that performs server-side rendering, allowing us to dynamically render HTML/CSS webpages to the user. A web-based front-end is more flexible and would enable us to develop more beautiful user interfaces compared to a python-based GUI library such as Tkinter.

The Docker Engine was also used to allow our team to provision local instances of the PostgreSQL database, loaded with the TPC-H dataset.

1.3 Project Structure

The project consists of the following 4 main program files as stated in the project brief:

- **interface.py**: This file is empty, details are explained in *Section 2.3 Interface*.
- **annotation.py**: To generate Natural Language Processing (NLP) on how different query components are executed.
- **preprocessing.p**y: Interfaces with the database to retrieve query plans and prepares the results.
- **project.py**: Main file that invokes all the procedures from the other files.

In addition to the above files, other files such as HTML templates, CSS stylesheets and javascript plugins are also included, as they are required for rendering the web page.

We employed the Procedural Oriented Programming model to structure this assignment into smaller components; each file has its purpose and utility functions. This makes the codes more understandable, as each file now serves to carry out a single purpose. The functional files are then called from and integrated with the views to serve requests from the clients.

The implementation of our application consists of the following steps:

Step 1: Generate execution plans

Step 2: Generate the natural language description of the query plan

Step 3: Present the results in the Graphical User Interface (GUI)

2. Key Algorithms

2.1 Preprocessing

2.1.1 Database connection

Various functions in our program require a database connection to the database. To avoid using a global variable or multiple instantiations of a database connection, the singleton pattern is used. This pattern ensures that at any one time, only one instance of the database connection exists, and if none exists at the time, one is created.

```
class DatabaseConnection:
   """
   Database connection singleton class.
   Call get_conn() to get a database connection.
   """

   _conn = None

   @classmethod
   def get_conn(cls):
       if cls._conn is None:
           cls._conn = psycopg2.connect(
               host="localhost",
               database="TPC-H",
               user="postgres",
               password="password123",
               port="5432",
           )
       return cls._conn
```

2.1.2 Natural Explanation for a Query Plan

```
def get_natural_explanation(plan):
    natural_explanation = []

    # queue for bfs
    q = deque([plan])
    while q:
        n = len(q)
        for _ in range(n):
            node = q.popleft()
            natural_explanation.append(natural_explain(node))
            if "Plans" in node:
                for child in node["Plans"]:
                    q.append(child)
    return natural_explanation[::-1]
```

The function takes a query plan and returns a list of strings.

1. The Breadth-First Search (BFS) algorithm is used to traverse all nodes of the query plan.
2. For each node, we get the natural explanation from the natural_explain() function by passing in the node and subsequently append the results to the natural_explanation list.
3. Finally, when all nodes have been traversed, we return the natural_explanation list in reverse order to place children's explanation first.

Details about the underlying natural explanation functions can be found in *Section 2.2.2 Natural Explanation*.

2.1.3 Query Plan Comparison

As there may be cases where PostgreSQL generates the same query plan even with different configurations disabled, we require a method for comparing query plans to avoid returning identical plans to the user.

```
def get_plan_comparison_attr(plan):
    inter_result = {"text": {"name": ""}}

    # get all key:value pair with string value
    for value in plan.items():
        if isinstance(value, str):
            inter_result["text"]["name"] += value + " "

    # remove last space from string
    inter_result["text"]["name"] = inter_result["text"]["name"][:-1]

    children_details = []

    if plan.get("Plans") is not None:
        for plan in plan.get("Plans"):
            children_details.append(get_plan_comparison_attr(plan))

    if len(children_details) != 0:
        inter_result["children"] = children_details

    return inter_result
```

The function takes a query plan and only retains important attributes such as the node types, relation names, conditions, etc. This simplified representation of a query plan can then be used for comparison with other query plans.

Algorithm:

1. The Breadth-First Search (BFS) algorithm is used to traverse all nodes of the query plan
2. For each node, we extract the important attributes and store them.
3. Finally when all nodes have been traversed, we return the simplified representation of the query plan.

2.1.4 Query Plans Generation

This is the main function that takes in a SQL query to generate the query plan, and subsequently the alternative query plans. The function receives the user's query as a parameter and returns a result dictionary that will be used to generate the data in the user interface.

```
def get_plans(user_query):
    """
    Get query plans for the given user query.

    Returns:
        - error boolean: True if error, False otherwise
        - result dictionary: contains message if error, else contains plan_data, summary_data
and natural_explain
    """

    # query string with placeholder
    query_str = f"EXPLAIN (ANALYZE false, SETTINGS true, FORMAT JSON) {user_query};"

    # dictionary to store results
    result = {
        "plan_data": [],
        "summary_data": [],
        "natural_explain": [],
    }

    try:
        # get database connection
        conn = DatabaseConnection.get_conn()

        # get base query plan
        with conn.cursor() as cur:
            try:
                # execute query and get results
                cur.execute(query_str)
                plan1 = cur.fetchall()[0][0][0].get("Plan")
            except ProgrammingError as e:
                print(e)
                cur.execute("ROLLBACK;")
                return True, {"msg": f"Invalid SQL query!"}
        # get alternative plans by passing in the previous plan
        plan2 = get_alt_plan(query_str, prev_plan=plan1)
        plan3 = get_alt_plan(query_str, prev_plan=plan2)

        # parse results for summary for plan 1
        summary1 = get_plan_summary(plan1)

        # get comparison attributes for plans
        plan_comp_attr1 = get_plan_comparison_attr(plan1)
        plan_comp_attr2 = get_plan_comparison_attr(plan2)
        plan_comp_attr3 = get_plan_comparison_attr(plan3)

        # get natural explanation for plan 1
        natural_explain1 = get_natural_explanation(plan1)

        # add plan 1 results to result object
        result["plan_data"].append(plan1)
        result["summary_data"].append(summary1)
        result["natural_explain"].append(natural_explain1)

        # check if plan2 is the same as plan1
        if plan_comp_attr2 != plan_comp_attr1:
            result["plan_data"].append(plan2)

            # parse results for summary for plan 2
            summary2 = get_plan_summary(plan2)
            result["summary_data"].append(summary2)

            # get natural explanation for plan 2
            natural_explain2 = get_natural_explanation(plan2)
            result["natural_explain"].append(natural_explain2)

        # check if plan3 is the same as plan1 or plan2
        if plan_comp_attr3 != plan_comp_attr1 and plan_comp_attr3 != plan_comp_attr2:
            result["plan_data"].append(plan3)

            # parse results for summary for plan 3
            summary3 = get_plan_summary(plan3)
            result["summary_data"].append(summary3)

            # get natural explanation for plan 3
            natural_explain3 = get_natural_explanation(plan3)
            result["natural_explain"].append(natural_explain3)

    except OperationalError as e:
        return True, {
            "msg": f"An error has occurred: Failed to connect to the database! Please ensure
that the database is running."
        }
    except Exception as e:
        return True, {"msg": f"An error has occurred: {repr(e)}"}

    # return False for no error, and results
    return False, result
```

Algorithm:

1. A database connection is obtained.
2. The base query plan is obtained by executing the query and getting the results without disabling any of the configuration parameters.
3. The first alternative plan is then obtained based on the base plan.
4. The second alternative plan is then obtained based on the first alternative plan.
5. The resultant plans are parsed for the summary, plan comparison attributes and NLP explanation information.
6. The resulting information is then added to the result if they are not duplicates and returned.
7. In the event of an error, an error message is returned instead.

2.1.5 Alternative Query Plan Generation

Outline of steps for AQP generation:

1. Get the plan for the query using the default settings (Optimal QEP)
2. Get a list of all the node types in the plan.
3. Map the node types to the configuration parameters and disable each configuration parameter.
4. Get the plan for each different configuration (Alternative QEP)
5. The configuration is reset after each query Alternative QEP.
6. Compare the plans and return the ones that are different to avoid duplicates.

To obtain alternative query plans, we first have to extract the operation types from a preceding query plan. The function returns a list of all node types used in a given plan.

```
def get_node_types(plan):
   # list to store all node types
   types = []

   # queue for bfs
   q = deque([plan])
   while q:
       n = len(q)
       for _ in range(n):
           node = q.popleft()
           types.append(node["Node Type"])
           if "Plans" in node:
               for child in node["Plans"]:
                   q.append(child)

   return types
```

This dictionary maps the Node Types to the Planner Method Configuration parameters. This dictionary is used to toggle the configuration parameters to get Alternative QEPs.

```
# mapping of node type to runtime setting name
params_map = {
   "bitmapscan": "enable_bitmapscan",
   "hashagg": "enable_hashagg",
   "Hash Join": "enable_hashjoin",
   "Index Scan": "enable_indexscan",
   "Index Only Scan": "enable_indexonlyscan",
   "Materialize": "enable_material",
   "Merge Join": "enable_mergejoin",
   "Nested Loop": "enable_nestloop",
   "Seq Scan": "enable_seqscan",
   "Sort": "enable_sort",
   "Tid Scan": "enable_tidscan",
}
```

Next, we disable all configuration parameters that are associated with these operations in order to force PostgreSQL to generate an alternate query plan.

```
def get_alt_plan(query_str, prev_plan):
   # Get all node type from prev_plan
   node_types = get_node_types(prev_plan)

   # Get list of query config settings parameters
   query_config_params = []
   for nt in node_types:
       if nt in params_map:
           query_config_params.append(params_map.get(nt))

   # get database connection
   conn = DatabaseConnection.get_conn()

   with conn.cursor() as cur:
       try:
           # Reset query plan configuration at the start
           cur.execute("RESET ALL;")

           # turn off each setting
           for item in query_config_params:
               cur.execute("SET LOCAL {} TO off;".format(item))
```

```
        # execute query and get results
        cur.execute(query_str)
        alt_plan = cur.fetchall()[0][0][0].get("Plan")

        # Reset query plan configuration at the end
        cur.execute("RESET ALL;")

    except Exception as e:
        # manage exceptions by rolling back transaction
        cur.execute("ROLLBACK;")
        # then re-raise exception from calling function to handle
        raise e

return alt_plan
```

Algorithm:

1. A previous query plan is passed into the function
2. A list of node types from the previous query plan is gotten from get_node_types()
3. The node types are then mapped to their corresponding Query Plan Configuration Parameters name.
4. The list of parameter names is looped through to toggle the Query Plan Configuration
5. The query is then executed to get the Alternative QEP
6. The Query Plan Configuration is then reset to the default configuration
7. Finally, the Alternative QEP is returned

2.2 Annotation

Two top-level functions are responsible for annotating the query plans generated by the database:

1. **`get_plan_summary()`** takes in a query plan as input and returns a summary.
2. **`natural_explain()`** takes in a query plan as input and returns a list of descriptions of the operations in natural language.

2.2.1 Plan Summary

The purpose of this function is to compute the total cost and number of operations of a query plan.

Similar to previous functions, the BFS approach is used to traverse all nodes of the plan.

```
def get_plan_summary(plan):
   """
   Produces a summary given a query plan.
   - Total cost of all nodes
   - Total number of nodes
   """

   summary = {
       "total_cost": 0,
       "nodes_count": 0,
   }

   # queue for bfs
   q = deque([plan])
   while q:
       n = len(q)
       for _ in range(n):
           node = q.popleft()
           summary["nodes_count"] += 1
           summary["total_cost"] += (node["Total Cost"])
           if "Plans" in node:
               for child in node["Plans"]:
                   q.append(child)

   return summary
```

2.2.2 Natural Explanation

The function takes in the query plan node and uses switch cases to invoke the corresponding explain functions and return the results of the function, which is a natural explanation string.

```
def natural_explain(plan):
    match plan["Node Type"]:
        case "Aggregate":
            return aggregate_natural_explain(plan)
        case "Append":
            return append_natural_explain()
        case "CTE Scan":
            return cte_natural_explain(plan)
        case "Function Scan":
            return function_scan_natural_explain(plan)
        case "Group":
            return group_natural_explain(plan)
        case "Index Scan":
            return index_scan_natural_explain(plan)
        case "Index Only Scan":
            return index_only_scan_natural_explain(plan)
        case "Limit":
            return limit_natural_explain(plan)
        case "Materialize":
            return materialize_natural_explain()
        case "Unique":
            return unique_natural_explain()
        case "Merge Join":
            return merge_join_natural_explain(plan)
        case "SetOp":
            return setop_natural_explain(plan)
        case "Subquery Scan":
            return subquery_scan_natural_explain()
        case "Values Scan":
            return values_scan_natural_explain()
        case "Seq Scan":
            return seq_scan_natural_explain(plan)
        case "Nested Loop":
            return nested_loop_natural_explain()
        case "Sort":
            return sort_natural_explain(plan)
        case "Hash":
            return hash_natural_explain()
        case "Hash Join":
            return hash_join_natural_explain(plan)
        case "Bitmap Heap Scan":
            return bitmap_heap_scan_natural_explain(plan)
```

As there are too many “explain” functions, one for each node type, we only include two functions in this report to keep the report concise. They all share the same concept and differ only in their explanations.

```
def sort_natural_explain(plan):
    result = f"The result is {bold_string('Sorted')} using the attribute "
    if "DESC" in plan["Sort Key"]:
        result += (
            bold_string(str(plan["Sort Key"].replace("DESC", "")))
            + " in descending order of "
        )
    elif "INC" in plan["Sort Key"]:
        result += (
            bold_string(str(plan["Sort Key"].replace("INC", "")))
            + " in ascending order of "
        )
    else:
        result += bold_string(str(plan["Sort Key"]))
    result += "."
    return result


def memoize_natural_explain(plan):
    result = f"The previous sub-operation result is then {bold_string('Memoized')}. This
means that the result is cached with cache key of {bold_string(plan['Cache Key'])}."
    return result
```

The function takes in the query plan node and returns a natural explanation string. The string is customised for each explanation function and built according to the node's parameters.

2.3 Interface

There is no function code for `interface.py` as we utilised a built-in function `render_template()` from Flask. This function helps render all views using a template file (HTML) and placeholders for dynamic data. Together with the Jinja template engine, we can render HTML pages that will be served to the user's browsers.

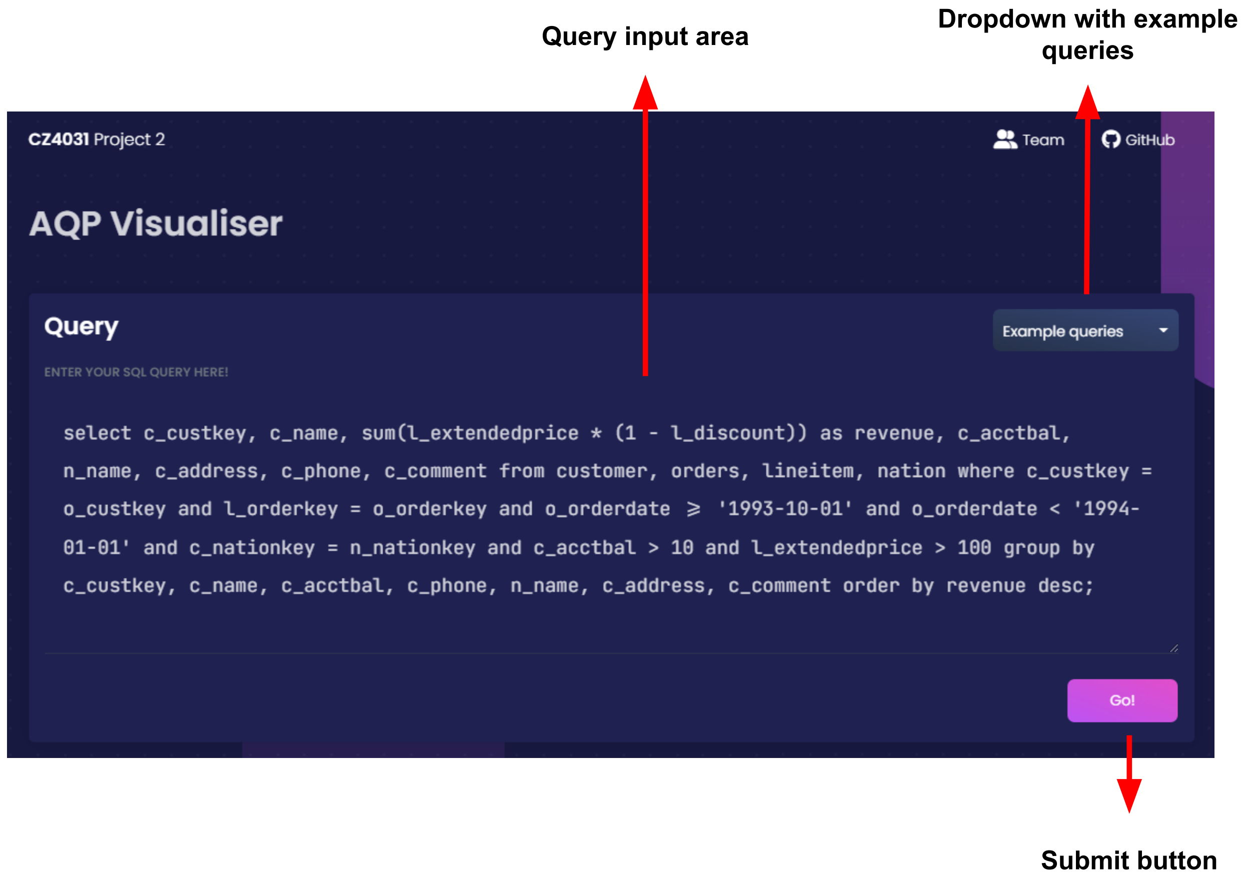
In our case, the entire interface.py can be re-looked into two components.

1. render_template() functions, as mentioned above, take in template HTML and dynamic data for rendering.
2. The HTML templates follow a modularised approach where we broke repeated components down, and made good use of inheritance. Specifically, the outline.html is our base template, and index.html inherits from it for content-based modification.

3. User Interface

3.1 Query Input Section

The query input section can be seen in Figure 1. The user can submit a query by entering it into the text box and clicking the “Go!” button. To improve user-friendliness and convenience, the user can also select an example query from the dropdown box.



```
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 ⩾ '1993-10-01' and o_orderdate < '1994-
01-01' and c_nationkey = n_nationkey and c_acctbal > 10 and l_extendedprice > 100 group by
c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment order by revenue desc;
```

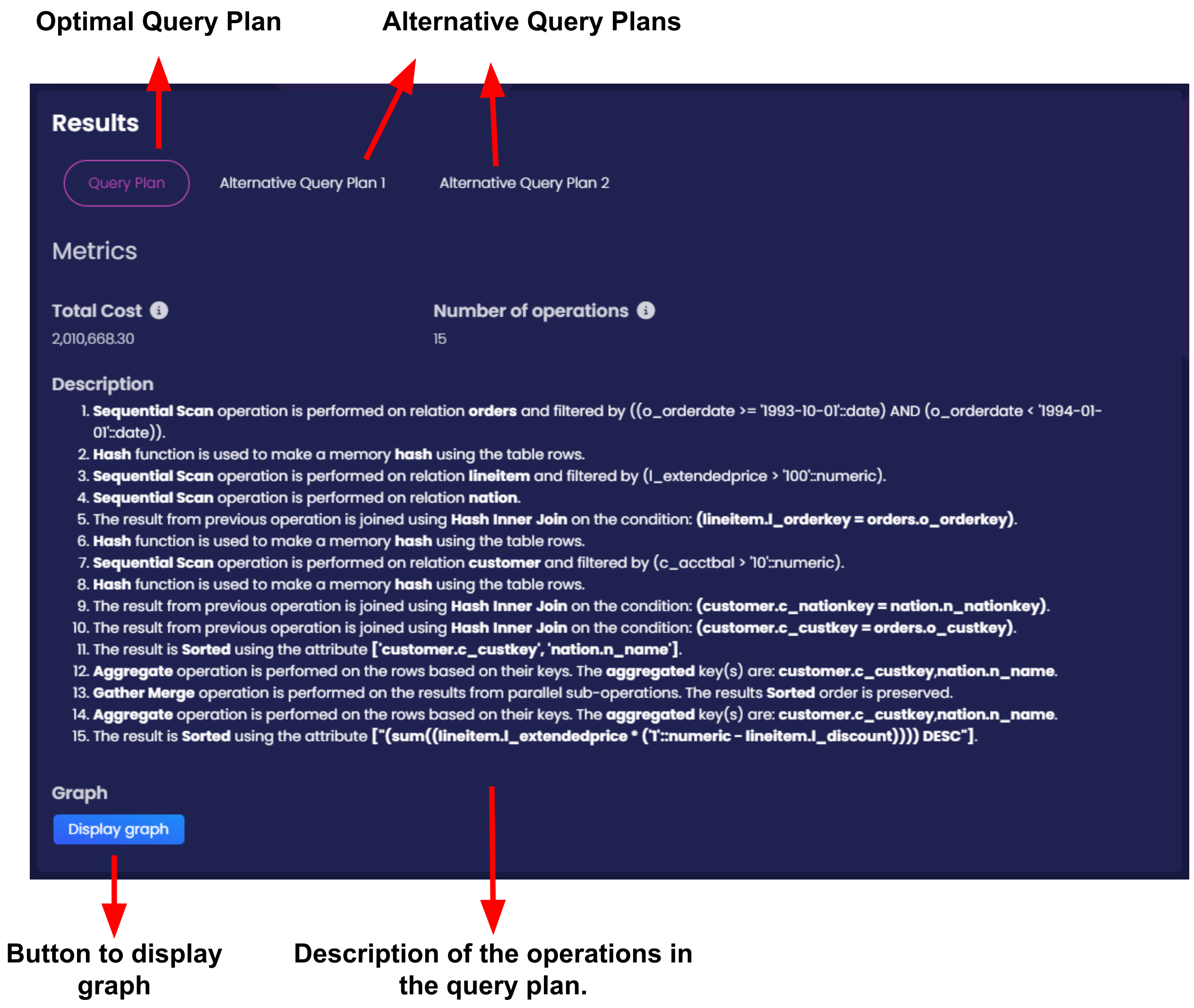
Figure 1: Query Input Section

3.2 Results Section

After the input has been processed by the server, the results will be displayed in the results section of the page (Figure 2). The user can view each of the generated plans by clicking on the respective tabs.

For each query plan, basic metrics such as the total cost and number of operations are shown. In addition, a step-by-step breakdown of the query plan is also included in the Description section. Finally, the user can also view a top-down tree diagram of the query plan by clicking on the

“Display graph” button. A modal with the graph will be displayed (Figure 3).



Results

Query Plan | Alternative Query Plan 1 | Alternative Query Plan 2

Metrics

Total Cost
2,010,668.30

Number of operations
15

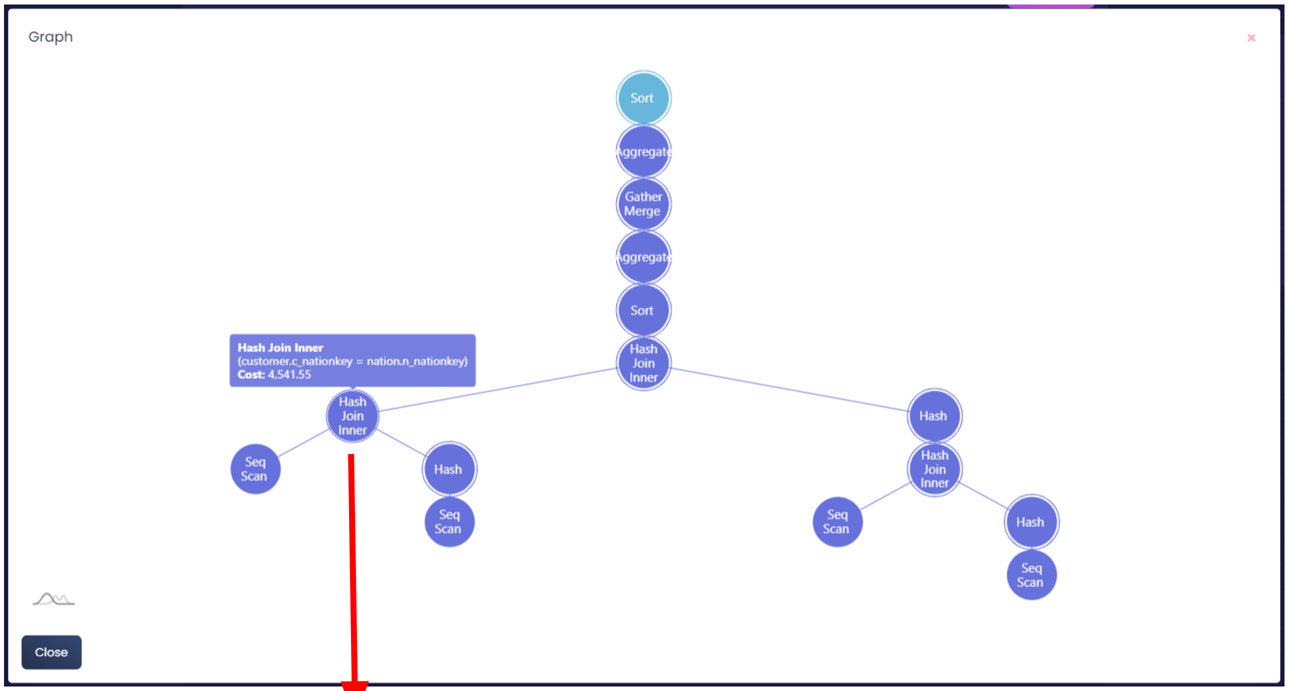
Description

1. **Sequential Scan** operation is performed on relation **orders** and filtered by ((o_orderdate >= '1993-10-01'::date) AND (o_orderdate < '1994-01-01'::date)).
2. **Hash** function is used to make a memory **hash** using the table rows.
3. **Sequential Scan** operation is performed on relation **lineitem** and filtered by (l_extendedprice > '100'::numeric).
4. **Sequential Scan** operation is performed on relation **nation**.
5. The result from previous operation is joined using **Hash Inner Join** on the condition: **(lineitem.l_orderkey = orders.o_orderkey)**.
6. **Hash** function is used to make a memory **hash** using the table rows.
7. **Sequential Scan** operation is performed on relation **customer** and filtered by (c_acctbal > '10'::numeric).
8. **Hash** function is used to make a memory **hash** using the table rows.
9. The result from previous operation is joined using **Hash Inner Join** on the condition: **(customer.c_nationkey = nation.n_nationkey)**.
10. The result from previous operation is joined using **Hash Inner Join** on the condition: **(customer.c_custkey = orders.o_custkey)**.
11. The result is **Sorted** using the attribute **['customer.c_custkey', 'nation.n_name']**.
12. **Aggregate** operation is perfomed on the rows based on their keys. The **aggregated** key(s) are: **customer.c_custkey,nation.n_name**.
13. **Gather Merge** operation is performed on the results from parallel sub-operations. The results **Sorted** order is preserved.
14. **Aggregate** operation is perfomed on the rows based on their keys. The **aggregated** key(s) are: **customer.c_custkey,nation.n_name**.
15. The result is **Sorted** using the attribute **["(sum((lineitem.l_extendedprice * ('1'::numeric - lineitem.l_discount)))) DESC"]**.

Graph

Display graph

Figure 2: Results Section



Mouseover a node to see the cost of the operation and other details

Figure 3: Tree Graph

4. Limitations

- The AQP generation method is to disable all algorithms used in the preceding plan whenever possible, as compared to turning off settings one by one in different combinations. Thus, the alternative query plans generated from this approach usually have a cost greatly more expensive than the optimal plan.

- Only up to two alternative query plans are generated.

- For natural language annotation, we were unable to name the input of merge operators. However, the tree diagram helps to identify the operators' input.

5. Installation guide

5.1 Database

As the TPC-H dataset is too large to be submitted, we assumed that an instance of the PostgresSQL database loaded with the TPC-H dataset is already available on the workstation for grading this project. Thus, the files required for setting up the database instance.

Internally, our team utilises Docker to run identical instances of the TPC-H database on our workstations. During creation, the Postgres docker container is configured to automatically import the TPC-H dataset by running a `.sql` script that is a complete database dump of the dataset.

To create the database dump, a fresh instance of PostgreSQL was spawned, and the dataset was manually loaded into the database by importing the various CSV files. The entire database was then dumped into a single `.sql` script for convenience. This approach reduces the number of steps required for each member of the group to perform, such as manually running scripts for table creation and importing the CSV files.

If you wish to set up the database instance, you can find the instructions and download link to the files in Appendix A.

5.2 Application

This section contains instructions on running the application. Alternatively, for your convenience, you can access the web application directly at this link: http://54.179.90.95/

The following instructions apply to Windows systems with **Python 3.10.x** already installed. For Mac systems, some commands may differ.

Steps:

1. Navigate to the `app/` directory.

2. Create a Python virtual environment.

```
python -m venv env
```

3. Activate virtual environment and install requirements.

   ```
   env\Scripts\activate
   pip install -r requirements.txt
   ```

4. Configure the database connection parameters to point to your Postgres instance in `app/preprocessing.py` (Lines 32-42)

   ```
   # Lines 32-42 snippet from app/preprocessing.py
   @classmethod
   def get_conn(cls):
       if cls._conn is None:
           cls._conn = psycopg2.connect(
               host="localhost",
               database="TPC-H",
               user="postgres",
               password="password123",
               port="5432",
           )
       return cls._conn
   ```

5. Run the project.

   ```
   python project.py
   ```

6. The web app will be accessible at http://localhost:5000/

Appendix A

Instructions for setting up a PostgreSQL database instance populated with the TPC-H dataset.

Pre-requisites: Docker Engine and Docker Compose installed.

1. Download `Dockerfile`, `docker-compose.yml` and `database-dump.sql` from OneDrive.

2. Ensure all three files are located in the same container.

3. Bring up the containers by running the command:
   ```
   docker-compose up -d
   ```

4. The database will be accessible on port 5432, and pgadmin will be accessible on port 8080.