****

**CZ4031 Database System Principles**

AY 2022/2023 Semester 1

**Project 2 Report**

Group 62

| **Name** | **Matriculation Number** | **Contribution** |
| --- | --- | --- |
| Goh Jia Rui, Bryna | U1822465A | Preprocessing |
| Goh Zheng Ying | U2021431A | Annotation |
| Ng Rui Hao Alexander | U2021730A | Preprocessing |
| Govindaraj Pranav | U2021117D | Annotation |
| Tan Caken | U2023173A | Interface |

# **Table of Contents**

[**Table of Contents**](#_fbc973eqcgxw) **2**

[**1. Introduction**](#_fc5f2nvto7c) **3**

[1.1 Description](#_5v5qrtwhhyo6) 3

[1.2 Project Overview](#_f523jpx002tb) 3

[1.3 Installation Guide](#_2kxtfugg8ixq) 3

[1.4 Database Guide](#_abfo5eh44dpr) 4

[**2. Interface**](#_1pzdr8hd9vor) **5**

[2.1 Query](#_mik27lo99mgo) 5

[2.2 Results](#_glfdqsjh6d56) 6

[2.3 Plot](#_v9hkwql2mftq) 7

[**3. Preprocessing**](#_eq69icknp4me) **8**

[3.1 SqlQuery](#_h6gjtsetht95) 9

[3.2 Node](#_s87q73lzah4t) 12

[3.3 Tree](#_u5uynevqn5w5) 13

[**4. Annotation**](#_4up1j7ohzs1z) **16**

[4.1 Annotate Class](#_7fps3set5rrl) 16

[4.2 Annotation Functions](#_ggbga9b7ez88) 16

[4.3 NodeAnnotation](#_mrp1kmt6umvh) 17

[4.4 MapAnnotation](#_eyfwh39voalu) 18

[**5. Examples**](#_9kb9u7z87j4r) **20**

[1. Example 1 (10.sql)](#_3btoyp83rvgz) 20

[2. Example 2 (19.sql)](#_uowr8hfuuwve) 21

[3. Example 3 (21.sql)](#_pw8bhtv272bx) 22

[**6. Limitations**](#_read6qlux0tk) **23**

# 

# **1. Introduction**

## **1.1 Description**

This project designs and implements three components of a web application that visualizes a database query plan. The project will be implemented with the Python language.

## **1.2 Project Overview**

These following lists the key sections of the report:

* **Interface** - Implementation of a simple GUI to take in a query and display a step-by-step explanation with a diagram.
* **Preprocessing** - Implementation of algorithms to process the QEP returned by the PostgreSQL database.
* **Annotation** - Implementation of algorithms to generate explanations for the operations performed in the query plan.

## **1.3 Installation Guide**

1. Unzip the source code zip file, or clone this [repository](https://github.com/GohZhengYing/CZ4031-Assignment-2) by running the command below:



1. Check and ensure that the machine has the latest Python version.
2. Ensure that you are in the ‘cz4031’ folder. Install the required package to run project.py by running the command below:



1. Run the executable by running .\project.py,



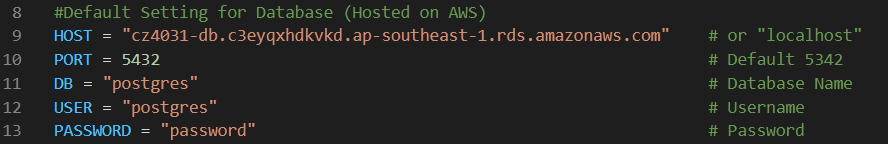
1. Run url given by the flask by copy the [link](http://127.0.0.1:8000/) to browser



## 1.4 Database Guide

Below are steps to change connection to other database:

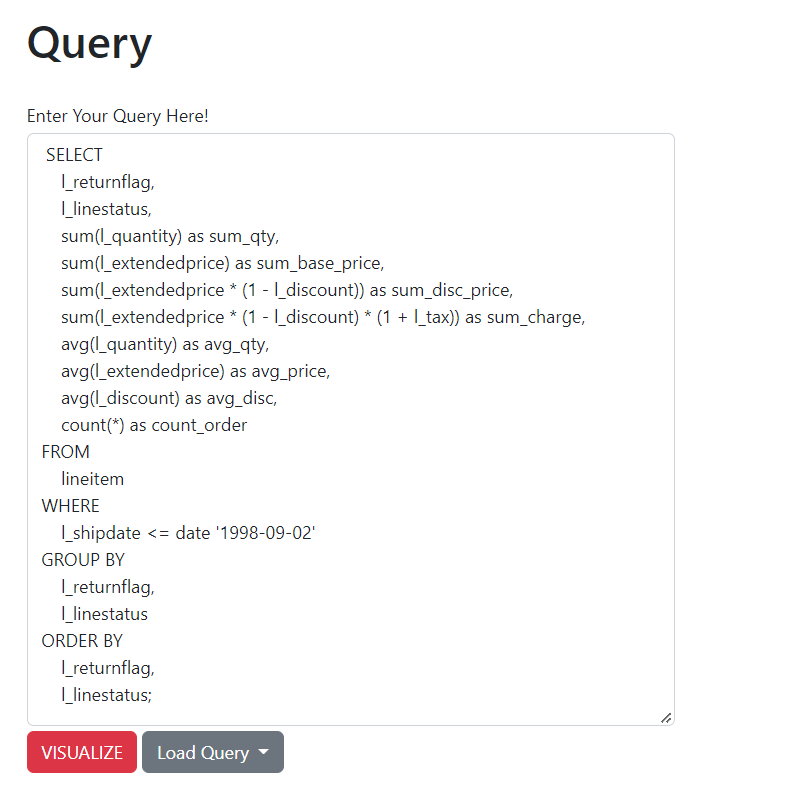
1. Open interface.py
2. You will see these settings between line 8 to 13.



1. Change settings to database that you want to connect.

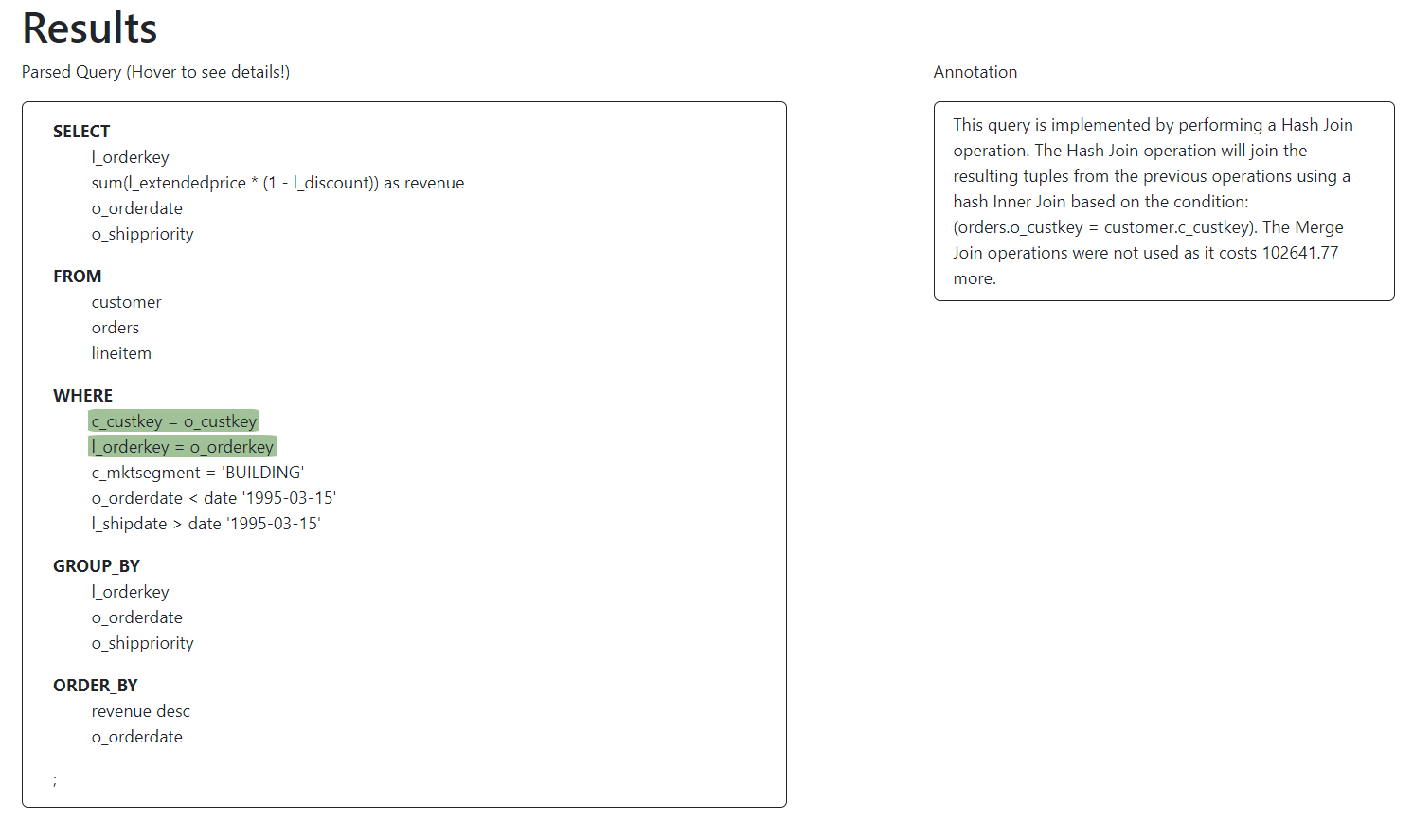
# **2. Interface**

## 2.1 Query



Query section consists of a textbox area where user-input query is located. Below the query section, there exists a “VISUALIZE” button and a “Load Query” button. Pressing **“VISUALIZE” button** will generate annotation for query inputted and show plot in later sections. **Load Query** is a dropdown menu where pre-defined query can be loaded to textbox area with help of a click.

## 2.2 Results



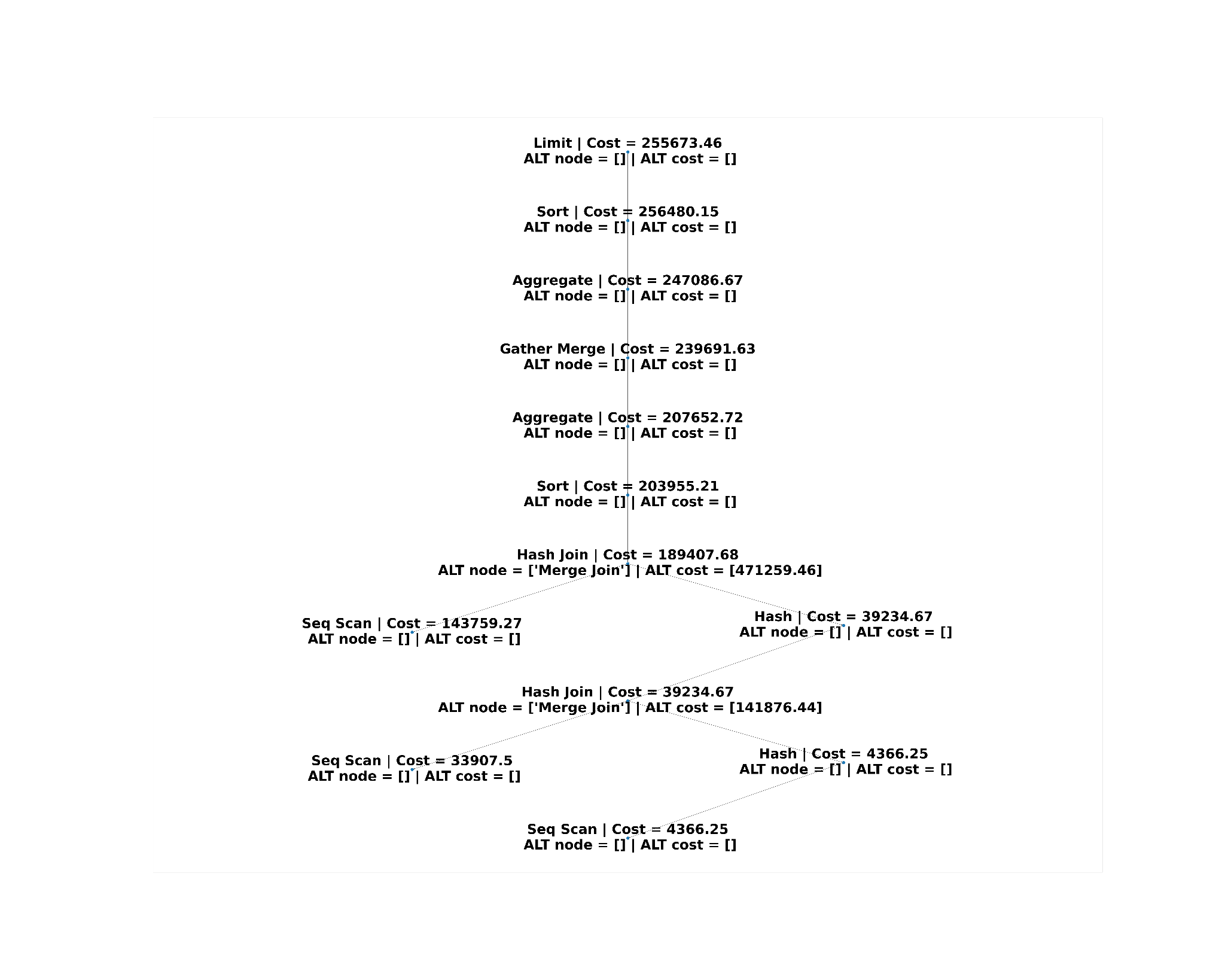
After user has successfully input a query, preprocessig.py will start to parse query according to requirements of annotation.py. Result of annotation.py should be dictionary. Result will be passed to interface to visualize and show the results.

Hover over green section in parsed query will allow annotation to display methods chosen to execute and reasoning behind choosing such methods, by comparing QEP with AQPs. More information will be explained in annotation.py

## 

## 

## 2.3 Plot



Plot section includes a networkx plot. Each node contains a node type to indicate an action and a cost variable. Each node can contain multiple alternate obtain for other AQP trees. This shows possible node types and costsfor a node.

## 

# 

# **3. Preprocessing**

Our preprocessing stage can be divided into 3 phases:

In phase 1, After the user has entered the SQL Query in the interface, the interface will create a query plan by passing the query into the explain\_query\_2 function. This function will execute the query in the database with EXPLAIN and will output as a query plan

In phase 2, we will then format our query into a SqlQuery object for the annotation segment.

In phase 3, we will create a main QEP tree based on the current query plan. We can then call the generate\_altnodes method from outside the tree to create the various AQP trees and map the alternate nodes to the nodes in the main QEP tree

We use the following 3 classes for our preprocessing stage:

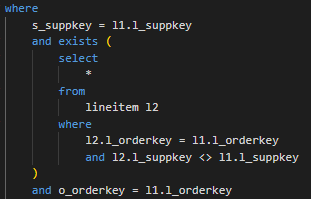
## 3.1 SqlQuery

Our SqlQuery class parses a given SQL query, extracting individual clauses in the query into separate data structures that can be easily accessed when performing annotation.

To parse a given SQL query, we first standardised the SQL format, as users may input SQL statements in various formats. This is done in the *\_standardise\_sql()* method. As such, we remove all semicolons, new lines and tab formatting, and standardised the whitespace to single spaces. The query was also lowercased, with the exception being text in quotes — these were left with their original casing, to allow them to still be matched with their PostgreSQL counterpart.



The next step was to handle any subqueries present in the query — these can pose problems for us when trying to parse the next keyword of the query. This is done in the *\_extract\_subqueries()* method.



For example, having a SELECT in the WHERE clause of an SQL statement could cause us to split the query subsections incorrectly. All subqueries, identified with brackets and nested SELECTs, are extracted to a list, and replaced with “SUBQUERY\_x”, where x is the index of the subquery in the list. Each of these subqueries will later be instantiated to generate new SqlQuery objects.

The SQL query is then split into their individual clauses. This is done using the *\_get\_next\_section()* method, which, given the beginning clause identifier (e.g. ‘FROM’ for the from clause), and a list of next possible clause identifiers (e.g. ‘WHERE’, ‘GROUP BY’, ‘ORDER BY’ or ‘LIMIT’ can all follow the FROM clause), identifies the string for the entire clause (e.g. “FROM customer, orders”). If none of the next clause identifiers are found, it is assumed that the end of the query has been reached, and the whole string from the beginning clause identifier to the end of line is returned.

Each clause string is then processed with individual methods, to account for the different structures of each clause. For example, aliases are parsed in the SELECT clause to dictionaries, with aliases as their keys and the column name as the value. If there is no alias, the column name will be used as both the key and value for that entry.

| **SQL Query** |  |
| --- | --- |
| **Parsed SELECT Clause** |  |

Another example is how the joins are parsed in the FROM clause. For joins in the FROM clause, only the join conditions are extracted to a list. For non-joins, the table names are extracted to a dictionary in a format similar to the SELECT dictionary.

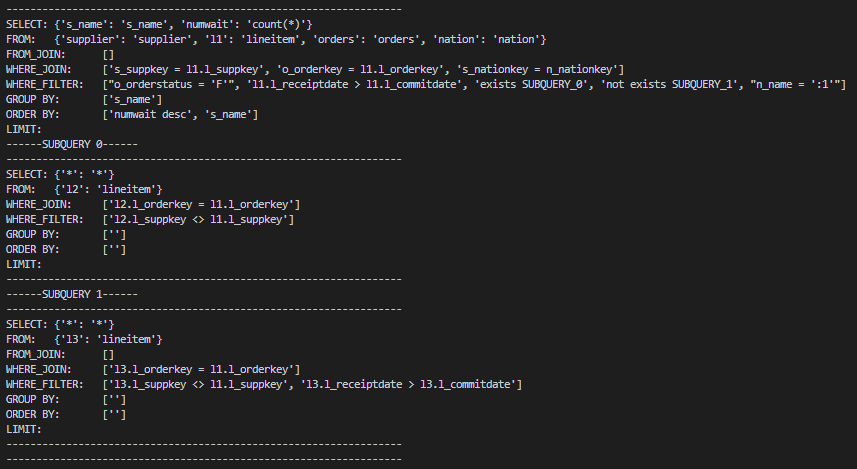
| **SQL Query** |  |
| --- | --- |
| **Parsed FROM Clause** |  |

Conditions in the WHERE clause are separated into *where\_joins* and *where\_filters*. *where\_joins* are conditions that include “=”, and has a format like *p\_partkey = l\_partkey*. *where\_filters* are all remaining conditions that are not *where\_joins*. BETWEEN conditions are also deconstructed into two separate *where\_filters*, to match PostgreSQL’s query plan:

| **SQL Query** |  |
| --- | --- |
| **Parsed WHERE Clause** |  |

The GROUP BY and ORDER BY clauses are processed into separate lists of column names, extracted by splitting the string with the “,” delimiter.

The output of the parsing of an example query (containing subqueries) is as shown below — some query information was not considered to be useful for our mapping to the PostgreSQL output, and was thus omitted:



## 3.2 Node

* node\_index - Index of node for node mapping
* data - Output of the EXPLAIN query for each node such as Node Type and Cost
* annotation - Explanation for each node
* alt\_node - List of corresponding alternate node in an AQP
* alt\_cost - List of corresponding alternate cost of node in the AQP

Each Node represents a node in the query plan provided by PostgreSQL.

When build\_tree() is called, the nodes are instantiated and are linked to each other, forming the tree. Node.index will be updated and node.data will updated with the data from each row of the query plan

When the node\_mapper() function is called, the mapped alternate node and alternate cost from the AQP will be appended to the alt\_node list and alt\_cost list for every node

When the add\_alt\_attributes(cur, combined\_df, tree) function is called, the annotation attribute will be updated with the required annotation for each node in the tree.

## 3.3 Tree

* Root - root node of tree
* edgeList - edges between nodes in the tree
* nodesList - nodes of the tree that have 5 properties to be passed to annotation for visualisation
  + Node\_index
  + Total cost
  + Node Type
  + Alt\_node
  + Alt\_cost
* fullNodesList - list of nodes in the tree with the full data for each node used for mapping nodes to the corresponding aqp nodes
* Parsed\_query -
* All\_annotations - list of annotations for each node in the tree
* query\_string - raw query string from the interface
* Operations - list of notable scan and join operations in the tree

Each Tree object represents a query plan generated by PostgreSQL, and is made of nodes, where each individual node is mapped to an annotation, given the corresponding node’s cost in other AQPs, and the original parsed SQL query.

Functions in the tree:

1. build\_tree (self, node)

Builds the tree based on the parsed query attribute in the tree

1. get\_networkx\_info (self, cur, num\_indent=0)

Populates the nodesList and the edgeList to which are passed to the networkx to generate the graph

1. display\_tree (self, cur, num\_indent=0)

Displays the built tree

1. get\_operations\_list(self)

Creates a list of scans and joins which are stored as an attribute operations

1. get\_multiple\_aqp(self, query\_string, operations, cur)

Generates multiple query plans by turning off different scans or joins and stores them in a list. It is called by the generate\_altnodes function

1. generate\_altnodes (self, tree, cur)

Function that is called outside the tree to generate the AQPs based on the current QEP. It firstly calls the get\_multiple\_aqp function to generate a list of AQP query plans and then iterates through that list to build multiple AQP trees and store it in list named aqp\_trees\_list. It then iterates through the aqp\_trees\_list to map the nodes of the AQP to the nodes in the QEP

Other functions in the preprocessing file include:

1. join \_mapper(list1, list2)

Inputs list1 and list2 are the fullNodesList from the main QEP tree and an AQP, it iterates through both lists to perform 1-1 mapping of the join nodes and outputs a corresponding dataframe

1. scan\_mapper(list1, list2)

Inputs list1 and list2 are the fullNodesList from the main QEP tree and an AQP, it iterates through both list to perform 1-1 mapping of scan nodes and outputs a corresponding dataframe

1. node\_mapper(tree, aqp\_tree)

Combines the outputs of the join\_mapper and scan\_mapper into a single dataframe. It will then call add\_alt\_attributes function to individually add the alternate nodes and costs to the alt\_node and alt\_cost list of each node

1. add\_alt\_attributes(cur, combined\_df, tree)

A function that is used to recursively travel the tree and append the alternate nodes to the alt\_node list for each node, and append the alternate cost to the alt\_cost for each node in the main QEP tree. It will also call the annotation function to update the node.annotation attribute for every node

# 

# **4. Annotation**

## 4.1 Annotate Class

**class Annotate(object)**

The Annotate class maps the node type of the plan to the annotation function for that node type. All the annotation functions are stored in a dictionary named annotation\_dict and the functions are mapped with their specific node types as their key.

**Example:**

plan['Node Type'] = ‘Hash’

**Input:** Annotate().annotation\_dict.get(plan['Node Type'],default\_annotate)(plan)

**Output:** The Hash operation reads data into a hash table, where it can easily be looked up by the hash key.

## 4.2 Annotation Functions

A function to generate an annotation is created for each node type, each function accepts the query plan of the node and returns the annotated string.

The following is the list of annotation functions and their individual node type:

1. Aggregate : aggregate\_annotate(plan)
2. Append : append\_annotate(plan)
3. CTE Scan : cte\_annotate(plan)
4. Function Scan : function\_scan\_annotate(plan)
5. Group : group\_annotate(plan)
6. Gather Merge : gather\_merge\_annotate(plan)
7. Index Scan : index\_scan\_annotate(plan)
8. Index Only Scan : index\_only\_scan\_annotate(plan)
9. Limit : limit\_annotate(plan)
10. Materialize : materialize\_annotate(plan)
11. Unique : unique\_annotate(plan)
12. Merge Join : merge\_join\_annotate(plan)
13. SetOp : setop\_annotate(plan)
14. Subquery Scan : subquery\_scan\_annotate(plan)
15. Values Scan : values\_scan\_annotate(plan)
16. Seq Scan : seq\_scan\_annotate(plan)
17. Nested Loop : nested\_loop\_annotate(plan)
18. Sort : sort\_annotate(plan)
19. Hash : hash\_annotate(plan)
20. Hash Join : hash\_join\_annotate(plan)
21. Default : default\_annotate(plan)

A default annotation function is created to annotate nodes that could not be mapped by the Annotate class.

**Example:**

**Input:** hash\_annotate(plan)

**Output:** The Hash operation reads data into a hash table, where it can easily be looked up by the hash key.

## 4.3 NodeAnnotation

The NodeAnnotation function helps us in comparing the alternative node, the query plan’s suggested node and the difference in Total costs if the alternative node was used instead.

**Example:**

Input: NodeAnnotation(postgres\_tree,alt\_node,alt\_cost)

Output:

' This query is implemented by performing a Hash Join operation.\n The Hash Join operation will join the resulting tuples from the previous operations using a hash Inner Join based on the condition: (supplier.s\_nationkey = nation.n\_nationkey). \n The Nested Loop operation was not used as it costs 98.08 more.'

## 4.4 MapAnnotation

The MapAnnotation function helps us to annotate the different sql statements to the relevant nodes in the query plan generated to identify which part of the query plan was produced due to the sql statement used.

The query inputted gets parsed in preprocessing to generate a parsed query in the following format:

{

“select” : {sql statements in the select clause} ,

“from”:{sql statements sql in the from clause},

“from\_join”:{sql statements in the from clause which are used for joins},

“where\_join”:{sql statements in the where clause that are used for joins},

“where\_filter”:{sql statements in the where clause used for filtering values},

“group\_by”:{sql statements in the group by clause},

“order\_by”:{sql statements in the order by clause},

“limit”:{sql statements in the limit clause},

“subqueries”:{sql statements that are due to subqueries},

}

A **mapped queries** dictionary gets returned which maps the relevant keys as stated in the format above, for instance “where\_join”, with the node generated in the Query plan and its necessary annotations. Also, the alternative query plan’s node and the difference in cost in doing this operation is also added to the annotation when calling this function as shown below. This is done so with the help of the NodeAnnotation function too within MapAnnotation.

**Example:**

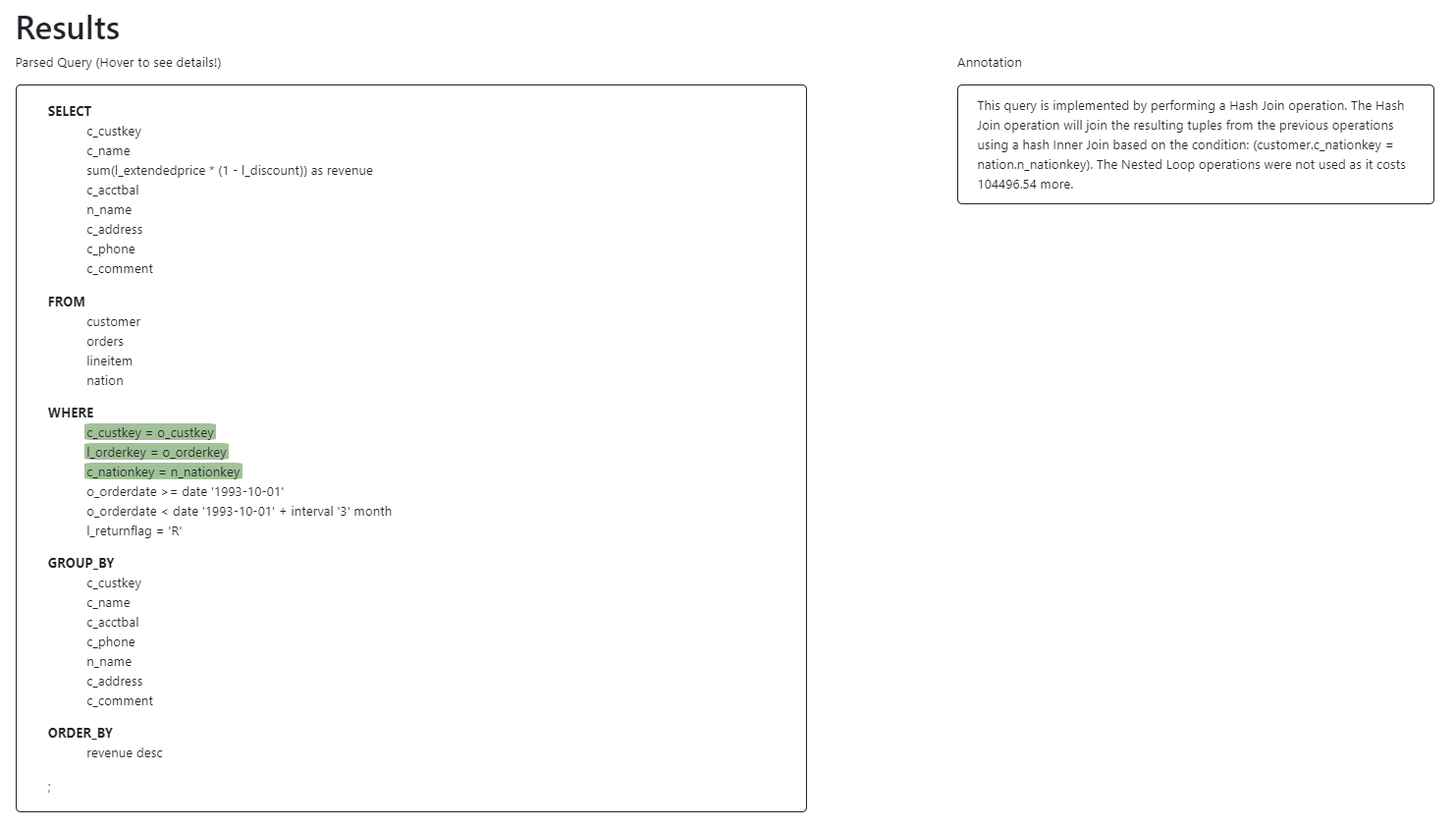
**Input:** annotation.MapAnnotation(parsed\_query,postgres\_tree, alt\_node,alt\_text**)**

**Output:**

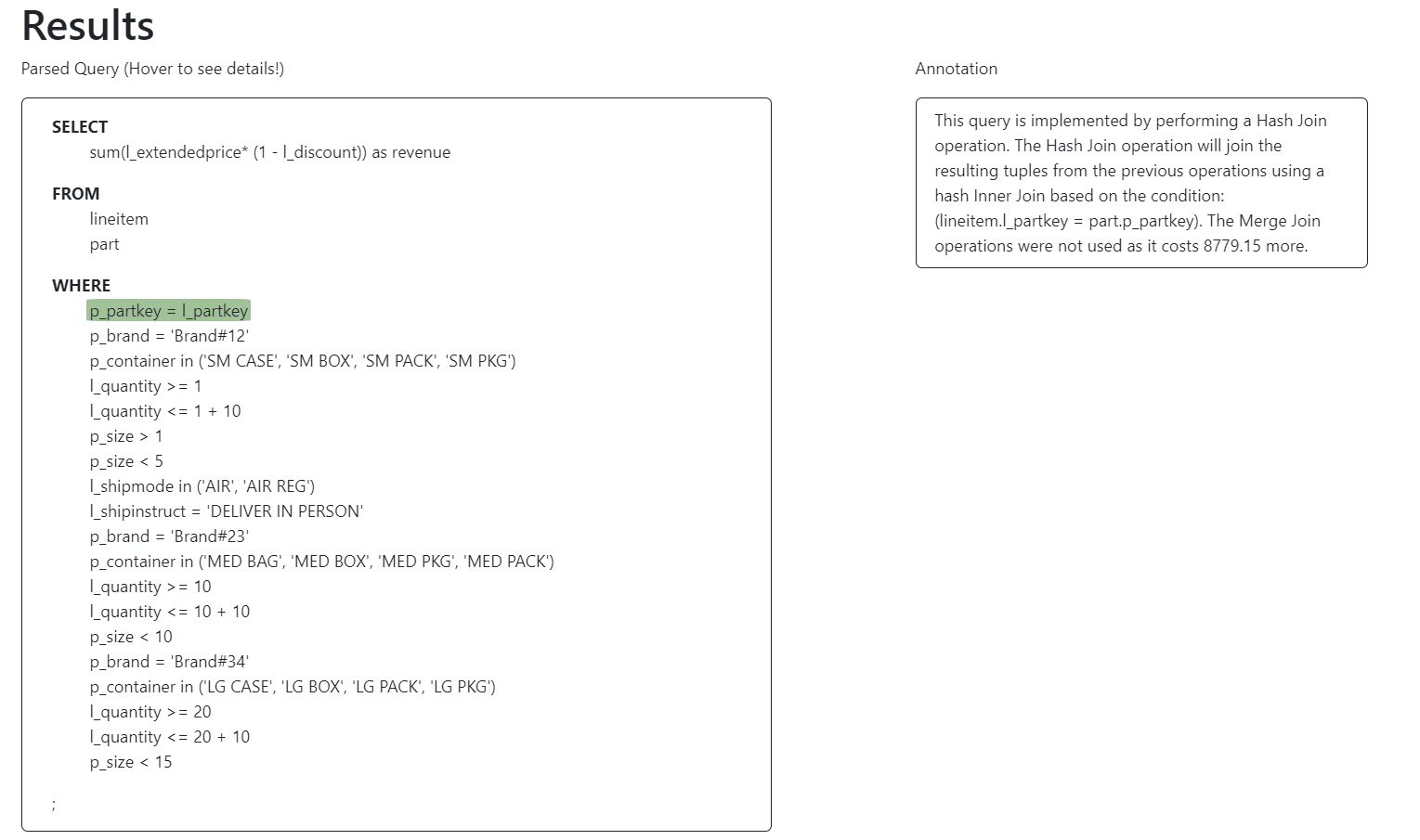
{'where\_join': {'s\_nationkey = n\_nationkey': ' This query is implemented by performing a Hash Join operation.\n The Hash Join operation will join the resulting tuples from the previous operations using a hash Inner Join based on the condition: (supplier.s\_nationkey = nation.n\_nationkey). \n The Nested Loop operation was not used as it costs 98.08 more.'}}

# **5. Examples**

## Example 1 (10.sql)



## Example 2 (19.sql)



## Example 3 (21.sql)

## 

# **6. Limitations**

| **Index** | **Limitations** |
| --- | --- |
| 1 | Mapping in preprocessing only works for scans and joins. |
| 2 | Unable to map clauses in sql within a subquery since it requires a recursive function to traverse each subquery and it will slow down the application. |
| 3 | Mapping for EXISTS and NOT EXISTS clauses to the relevant nodes in the query plan do not work. |
| 4 | Processing time was slower than expected |