Nanyang Technological University
School of Computer Science and Engineering

CZ4031 Database System Principles
Academic Year 2022/2023 Semester 1

Project 2 Report

Database System Principles

Group 34

JAW LI SHENG - U1921172F

LIM JIA JING - U1922552A

MA CHUNYOU - U2023930A

MUHAMMAD IRFAN BIN NORIZZAM - U2021872E

XUE YUSHAN - U2022049H

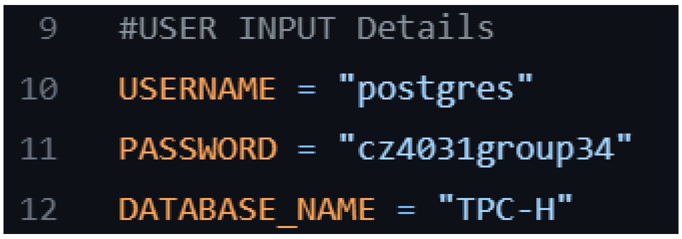
CONTRIBUTION FORM

Name	Contributions to Project 2
XUE YUSHAN	Everyone contributed equally
LIM JIA JING	Everyone contributed equally
MA CHUNYOU	Everyone contributed equally
MUHAMMAD IRFAN BIN NORIZZAM	Everyone contributed equally
JAW LI SHENG	Everyone contributed equally

After following the instructions to downloading the dependencies and running project.py:

MANUAL FOR OUR PROGRAM

1. Before running project.py, the User should first open preprocessing.py and edit their User Login Details frome lines 9-12 for their Postgres Database as shown below.

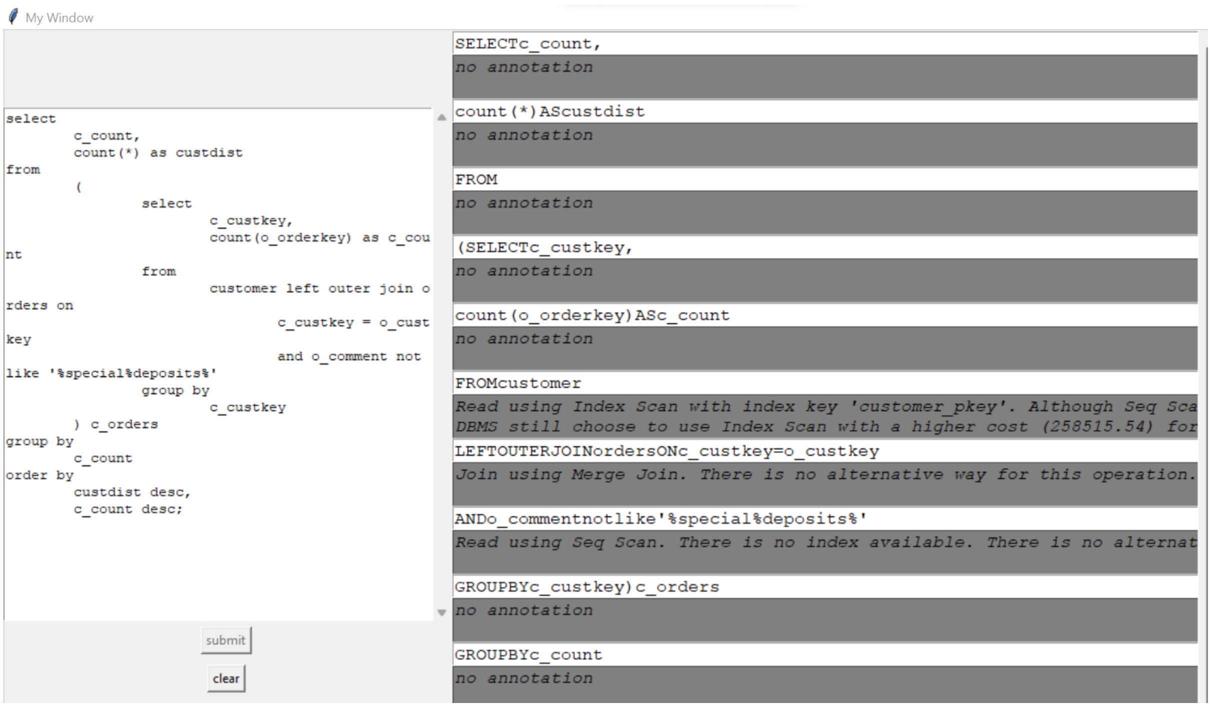


```
 9   #USER INPUT Details
10   USERNAME = "postgres"
11   PASSWORD = "cz4031group34"
12   DATABASE_NAME = "TPC-H"
```

2. After running project.py, there will be a pop-up window and the program will first ask the user to input their sql query on the left side of the UI as shown below and press submit. If the user's input details is incorrect, the pop-up window will exit.

3. After pressing submit the UI will display the output formatted query and the annotations on the right side of the UI as shown below.21

4. Before type in the new query, please click the clear button to clear the current output and query in the input box.



```
select
        c_count,
        count(*) as custdist
from
        (
                select
                        c_custkey,
                        count(o_orderkey) as c_cou
nt
                from
                        customer left outer join o
rders on
                                c_custkey = o_cust
key
                                and o_comment not
like '%special%deposits%'
                group by
                        c_custkey
        ) c_orders
group by
        c_count
order by
        custdist desc,
        c_count desc;
```

```
SELECTc_count,
no annotation
count(*)AScustdist
no annotation
FROM
no annotation
(SELECTc_custkey,
no annotation
count(o_orderkey)ASc_count
no annotation
FROMcustomer
Read using Index Scan with index key 'customer_pkey'. Although Seq Sca
DBMS still choose to use Index Scan with a higher cost (258515.54) for
LEFTOUTERJOINordersONc_custkey=o_custkey
Join using Merge Join. There is no alternative way for this operation.
ANDo_commentnotlike'%special%deposits%'
Read using Seq Scan. There is no index available. There is no alternat
GROUPBYc_custkey)c_orders
no annotation
GROUPBYc_count
no annotation
```

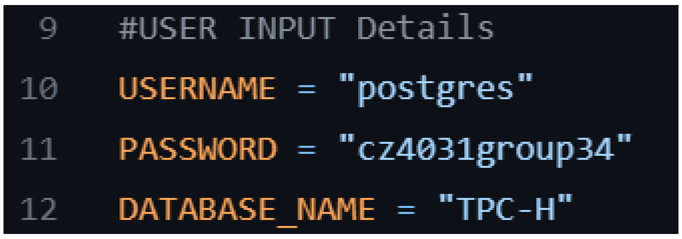
1) Introduction

This project involves the retrieval of relevant information from a QEP and AQP(s) to annotate the corresponding SQL query to explain how different components of the query are executed by the underlying query processor and why the operators are chosen among other alternatives. The source code of the project can be found at https://github.com/LimJiaJing/cz-4031-project-2.

2) Overall Flow of Program

Overall flow of the program will be as such:

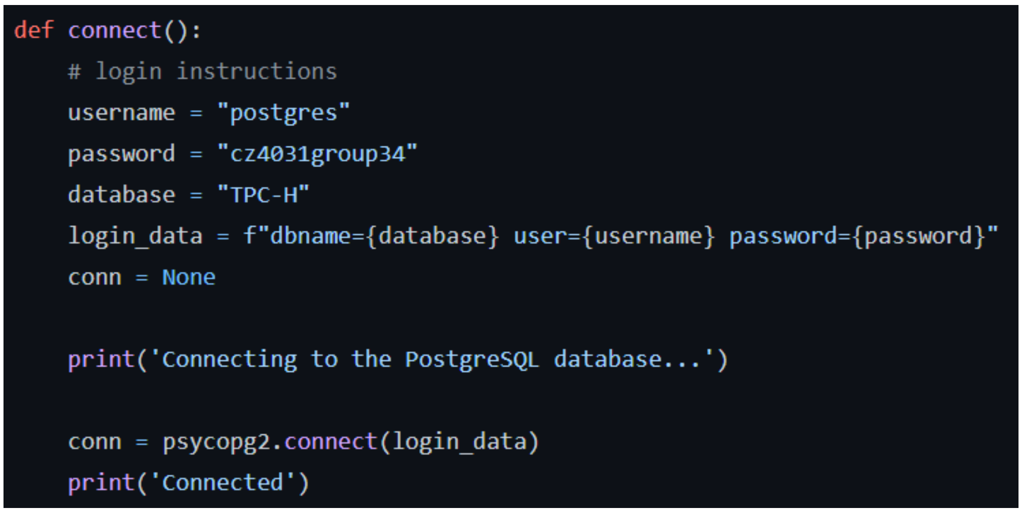
1. Before running the program, ensure all the dependencies are downloaded. The user will then have to access the preprocessing.py file and change lines 9-12 with the users Login Details for their Postgres Database. The user can then run the project.py file.



```
 9   #USER INPUT Details
10   USERNAME = "postgres"
11   PASSWORD = "cz4031group34"
12   DATABASE_NAME = "TPC-H"
```

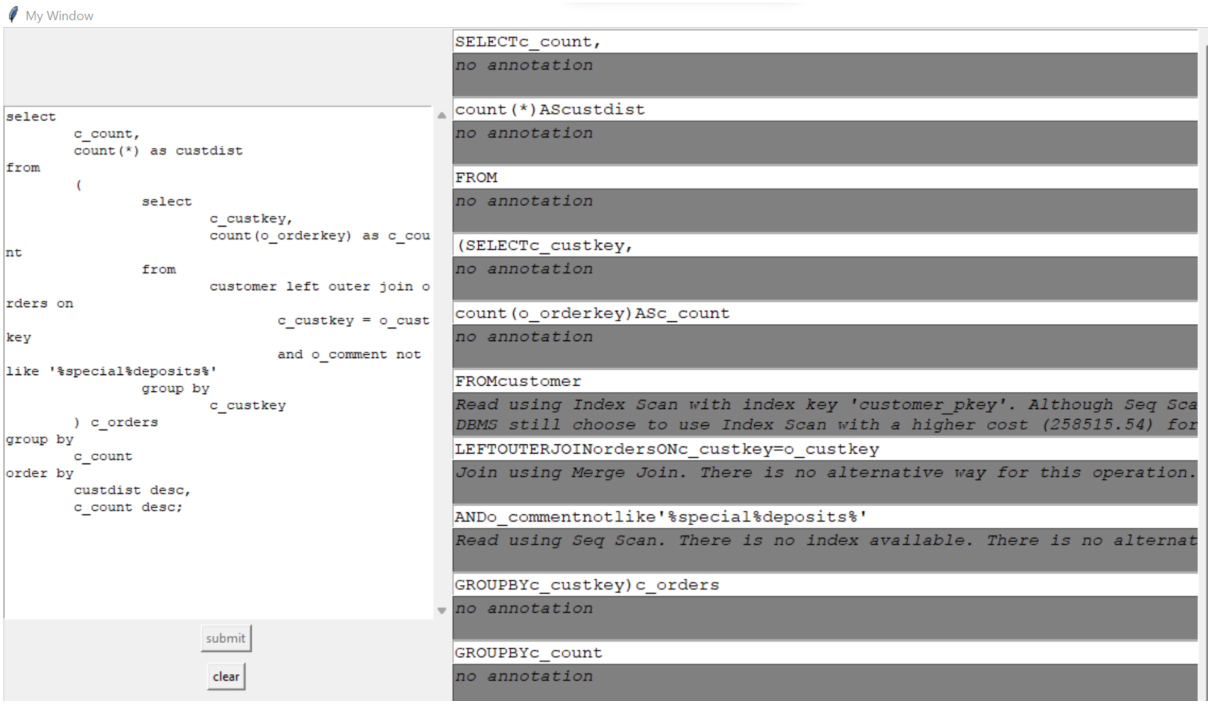
2. The Initial connection to the Postgresql TPC-H database is done via the connect()

function in preprocessing.py. After the connection is established, the connection and tables will be passed to the UI interface functions. An example of the inputs used is as shown below.



```python
def connect():
    # login instructions
    username = "postgres"
    password = "cz4031group34"
    database = "TPC-H"
    login_data = f"dbname={database} user={username} password={password}"
    conn = None

    print('Connecting to the PostgreSQL database...')

    conn = psycopg2.connect(login_data)
    print('Connected')
```

3. The following is a visual representation of the UI.



```
select
        c_count,
        count(*) as custdist
from
        (
                select
                        c_custkey,
                        count(o_orderkey) as c_cou
nt
                from
                        customer left outer join o
rders on
                                c_custkey = o_cust
key
                                and o_comment not
like '%special%deposits%'
                group by
                        c_custkey
        ) c_orders
group by
        c_count
order by
        custdist desc,
        c_count desc;
```

```
SELECTc_count,
no annotation
count(*)AScustdist
no annotation
FROM
no annotation
(SELECTc_custkey,
no annotation
count(o_orderkey)ASc_count
no annotation
FROMcustomer
Read using Index Scan with index key 'customer_pkey'. Although Seq Sca
DBMS still choose to use Index Scan with a higher cost (258515.54) for
LEFTOUTERJOINordersONc_custkey=o_custkey
Join using Merge Join. There is no alternative way for this operation.
ANDo_commentnotlike'%special%deposits%'
Read using Seq Scan. There is no index available. There is no alternat
GROUPBYc_custkey)c_orders
no annotation
GROUPBYc_count
no annotation
```

The UI can be divided into 2 different parts.

- The first section will be at the left side of the UI which will be where, the user will key in his SQL Query.

- The second section will be at the right side of the UI. This is where the relevant lines of the input query and its annotations will be displayed for the user's analysis.

4. After the user enters the query in the textbox and click the "Submit" button in the UI, the inputted query will be passed to the SQL parser to be cleaned (**SQL Parsing algorithm**).

5. At the same time, the query will also be passed to Postgresql where the Query Execution Plan (QEP) will be generated and outputted into a JSON file, "qep.json"

6. The qep.json file is then read and relevant node types such as "Index Scan, "Hash Join" will be identified if present. These node types are then used to generate the representative Alternative Query Plans (AQP) via the Planner Configuration Method. For each identified Node Type, its corresponding operation is disabled to generate an AQP. An aqp_info.txt is also generated, outlining which operations were disabled in each AQP file. The AQP files generated follows this naming convention - aqp_{i}.json.

7. Following the generation of the required QEP and AQPs, their JSON files are cleaned up to retain only the relevant keys.

8. The cleaned up JSON files are then read and parsed into one dictionary each via the **Summarize algorithm**. The format of the dictionary is as follows:
   a. Key: condition/relation to be annotated in the SQL
   b. Value: list of lists. Each inner list contains information about the operation used. It is a list that might be more than one due to duplicates.
      i. Content of each inner list: [Node Type, Index Key, Relation Name, Condition, Join Algorithm, Cost]
         1. Node Type: The value of the "Node Type" key in the QEP/AQP JSON file.
         2. Index Key: The value of the "Index Type" key in the QEP/AQP JSON file.
         3. Relation Name: The value of the "Relation Name" key in the QEP/AQP JSON file.
         4. Condition: The value of "Hash Cond"|"Index Cond"|Merge Cond" key in the QEP/AQP JSON file.
         5. Join Algorithm: The value of the "Node Type" key in the QEP/AQP JSON file
      ii. Based on each condition, some of these arguments might not be present/intentionally excluded. A 'None' is inserted in such a case.

9. For each of the summary dictionaries generated in **8**, the keys are matched to the

parsed SQL to obtain the annotation used for the corresponding part of the SQL query via the **Matching algorithm**. This generated the explanation for the SQL for each individual QEP/AQP. The explanations for each part of the SQP are then collated across the QEP and AQPs.

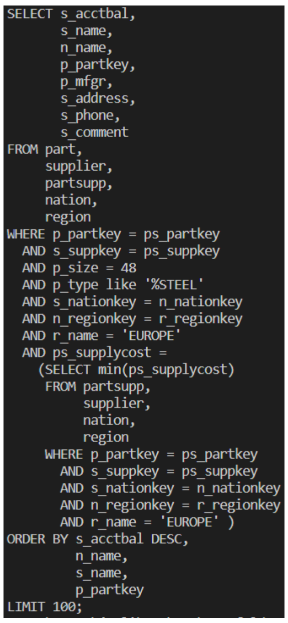
10. The **Comparison algorithm** then compares the QEP to the AQPs and chooses a representative AQP for each part of the SQL. An annotation is then generated comparing the QEP operation and the representative AQP.

11. The annotation is then showcased in the UI.

3) Key Algorithms used in the Project

SQL Parsing

- The input SQL query is first formatted by an external library called *sqlparse*. This is used to provide a uniform SQL format for comparison and annotation.
- An example of a formatted SQL query:

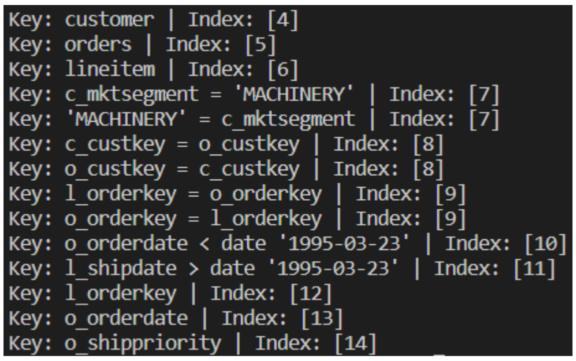


```
SELECT s_acctbal,
       s_name,
       n_name,
       p_partkey,
       p_mfgr,
       s_address,
       s_phone,
       s_comment
FROM part,
     supplier,
     partsupp,
     nation,
     region
WHERE p_partkey = ps_partkey
  AND s_suppkey = ps_suppkey
  AND p_size = 48
  AND p_type like '%STEEL'
  AND s_nationkey = n_nationkey
  AND n_regionkey = r_regionkey
  AND r_name = 'EUROPE'
  AND ps_supplycost =
    (SELECT min(ps_supplycost)
     FROM partsupp,
          supplier,
          nation,
          region
     WHERE p_partkey = ps_partkey
       AND s_suppkey = ps_suppkey
       AND s_nationkey = n_nationkey
       AND n_regionkey = r_regionkey
       AND r_name = 'EUROPE' )
ORDER BY s_acctbal DESC,
         n_name,
         s_name,
         p_partkey
LIMIT 100;
```

- After formatting, each line is checked to determine whether it is required for comparison and annotation. Only statements in the FROM, WHERE, and GROUP BY clauses are kept. The remaining lines are filtered out.
- After filtering, the remaining lines are either cleaned to remove unnecessary keywords or modified to ease comparison with the QEP and AQPs. This is achieved using

regular expressions from the *re* library.

- Examples of removing unnecessary keywords and modifying the line:
    - Example 1:
        - Before: **AND** c_nationkey = **n1**.n_nationkey
        - After: c_nationkey = n_nationkey
    - Example 2:
        - Before: **AND** o_orderdate **BETWEEN** date '1995-01-01' **AND** date '1996-12-31'
        - After: o_orderdate >= date '1995-01-01', o_orderdate <= date '1996-12-31'
- After cleaning up, the SQL parser will return a dictionary where the keys will be the cleaned up version of each SQL line and the values will be a list of indexes representing the line's position in the formatted SQL query
- An example of the output dictionary:



```
Key: customer | Index: [4]
Key: orders | Index: [5]
Key: lineitem | Index: [6]
Key: c_mktsegment = 'MACHINERY' | Index: [7]
Key: 'MACHINERY' = c_mktsegment | Index: [7]
Key: c_custkey = o_custkey | Index: [8]
Key: o_custkey = c_custkey | Index: [8]
Key: l_orderkey = o_orderkey | Index: [9]
Key: o_orderkey = l_orderkey | Index: [9]
Key: o_orderdate < date '1995-03-23' | Index: [10]
Key: l_shipdate > date '1995-03-23' | Index: [11]
Key: l_orderkey | Index: [12]
Key: o_orderdate | Index: [13]
Key: o_shippriority | Index: [14]
```

Summarize

The input to the summarize algorithm is a

- cleaned up JSON file of AQP/QEP

The output of the summarize algorithm is a dictionary:

- Key: condition/relation to be annotated in the SQL
- Value: list of lists. Each inner list contains information about the operation used. It is a list that might be more than one due to duplicates.
    - Content of each inner list: [Node Type, Index Key, Relation Name, Condition, Join Algorithm, Cost]
        - Node Type: The value of the "Node Type" key in the QEP/AQP JSON file.
        - Index Key: The value of the "Index Type" key in the QEP/AQP JSON file.
        - Relation Name: The value of the "Relation Name" key in the QEP/AQP JSON file.
        - Condition: The value of "Hash Cond"|"Index Cond"|Merge Cond" key in the QEP/AQP JSON file.
        - Join Algorithm: The value of the "Node Type" key in the QEP/AQP JSON file
    - Based on each condition, the arguments are intentionally chosen to be None or

having a valid value.

How the elements in the list are chosen to be None or to have a valid value:

- There are 4 cases. These cases are non-exclusive (i.e. multiple cases can be satisfied)
  Case 1: If the key is a filter condition based on a join algorithm
  Case 2: If the key is a filter condition on an index scan
  Case 3: If the key an Index scan condition
  Case 4: If the key is a table that is read using index scan

Additionally, Index Join are determined based on the following conditions (both must be satisfied):

1. The Node Type is an Index Scan
2. The Parent Node Type is a Nested Loop join

Matching

Assumptions: Duplicate conditions use the same operation.

The inputs to the matching algorithm are

- The output of the SQL Parsing algorithm, sql_dict
- The output of the Summary algorithm, plan_dict

The output to the matching algorithm is a dictionary

- Key: (sql_string_to_annotate, lines where this string appear)
- Value: Annotation

Iterate over the keys of the sql_dict and for each key, check if the key exists in plan_dict.

If the key does exist in the plan_dict: there are 2 cases:

- Case 1: There is more than one match for the given key, i.e. len(value)>1. In this case, we check if there is “Seq Scan” in one of the options. If there is, we take the “Seq Scan” option. This is because all other operations are built on top of the output of “Seq Scan” if “Seq Scan” is one of the options.
- Case 2: There is only one match.

Using chosen match, which is a list [Node Type, Index Key, Relation Name, Condition, Join Algorithm, Cost], we construct an explanation.

- Example explanation: Join using Index Join. The Index Join used relation 'nation' index key 'nation_pkey'.
- Based on the value of each element in the list, we can determine how the explanation should look like.

If there key does not exist in plan_dict: we check if the key has “ date “ in it. If there is, we find the keys in plan_dict that contain dates. Among these keys, we find the one that has the longest matching substring with the sql_dict key and use that as the matched key. We then repeat what we do when there is a matched key.

Comparison
The input dictionary for comparison is in the format as:
Keys: (*query_component*, *query_component_line_index*)
Values: ((*QEP_summary*, *QEP_cost*), (*AQP1_summary*, *AQP1_cost*), (*AQP2_summary*, *AQP2_cost*)...}

The comparison mainly focuses on join and scan operations. With keyword matching, the operation type (e.g. Seq Scan, Merge Join) will be extracted from the QEP/AQP summaries.

Then all the AQP tuples will be iteratively compared with the QEP on the cost. If there is a difference between QEP algorithm and AQP algorithm, an explanation annotation will be generated. Usually, the QEP algorithm will have lower cost than all different AQP algorithms. However, sometimes some AQP algorithm has lower cost than the corresponding QEP algorithm on an individual operation, possibly because QEP considers optimising consequent operations to minimise the total cost although it has higher cost at one stage. This special situation will be annotated as well.

The final annotation includes following information:

1. The summary for the algorithm in QEP
2. If there is no AQP available or all algorithms in AQP are the same as in QEP, annotate *"There is no alternative way for this operation."*
3. If there are different AQP operation(s), annotate the comparisons between QEP algorithm and different AQP algorithms e.g. "*Merge Join is not used because its cost is 1000, 10 times larger than Hash Join", "Although Hash Join has lower cost (100), DBMS still choose to use Merge Join with a higher cost (500) for some other consideration."(special case)*
4. Specially for sequential scan, if it is chosen in QEP, it indicates that there is no index available on the attributes involved in the filter condition. Then annotate *"There is no index available."*

The output of the comparison function is also a dictionary with format:
Keys: *query_component_line_index*
Values: *annotation*

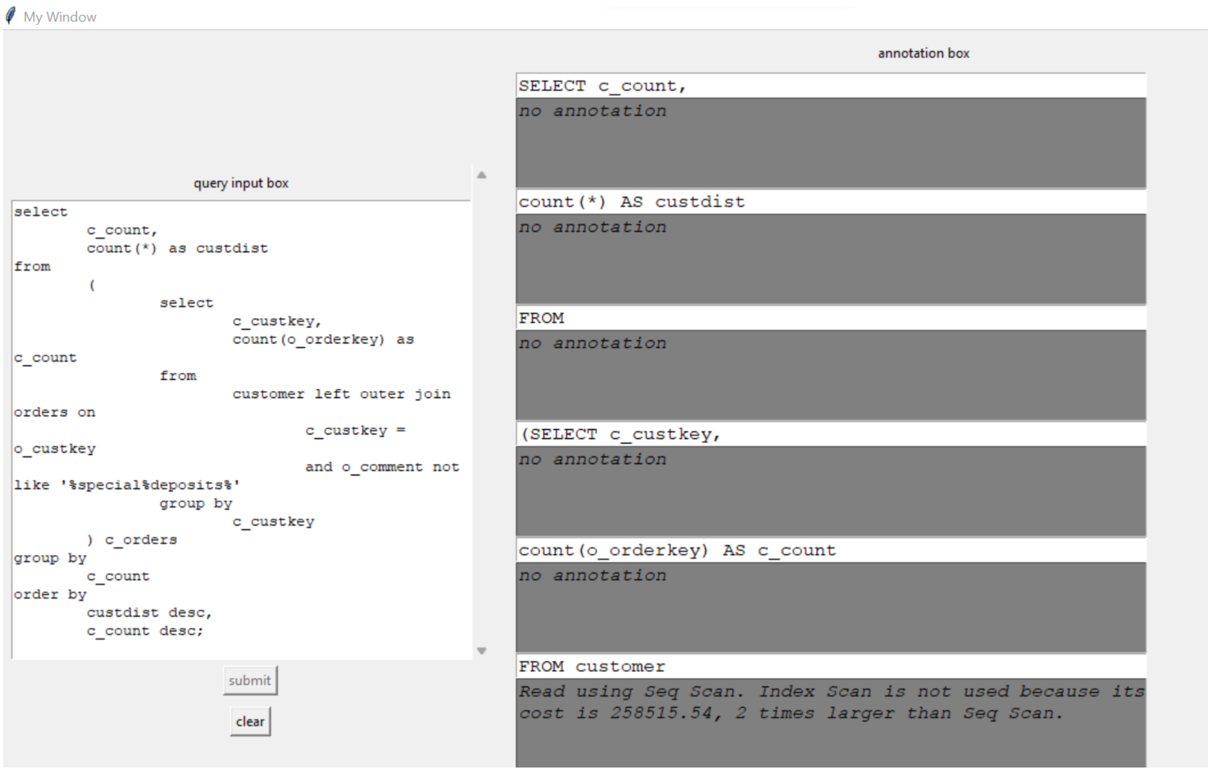
4) Limitations of our project

1. The first limitation of our project is that our project does not cover all possible Node Types currently available in Postgresql 15. As we are mainly exposed to sort and join functions in this Database System Principles Module, we mainly focus on those operations.
2. The UI set-up for this project does not come with any error-handling features. It will

not display any errors such as syntax errors in the SQL query on the UI itself.

5) **Examples of some output** (Once done ill just add 2 to the report)

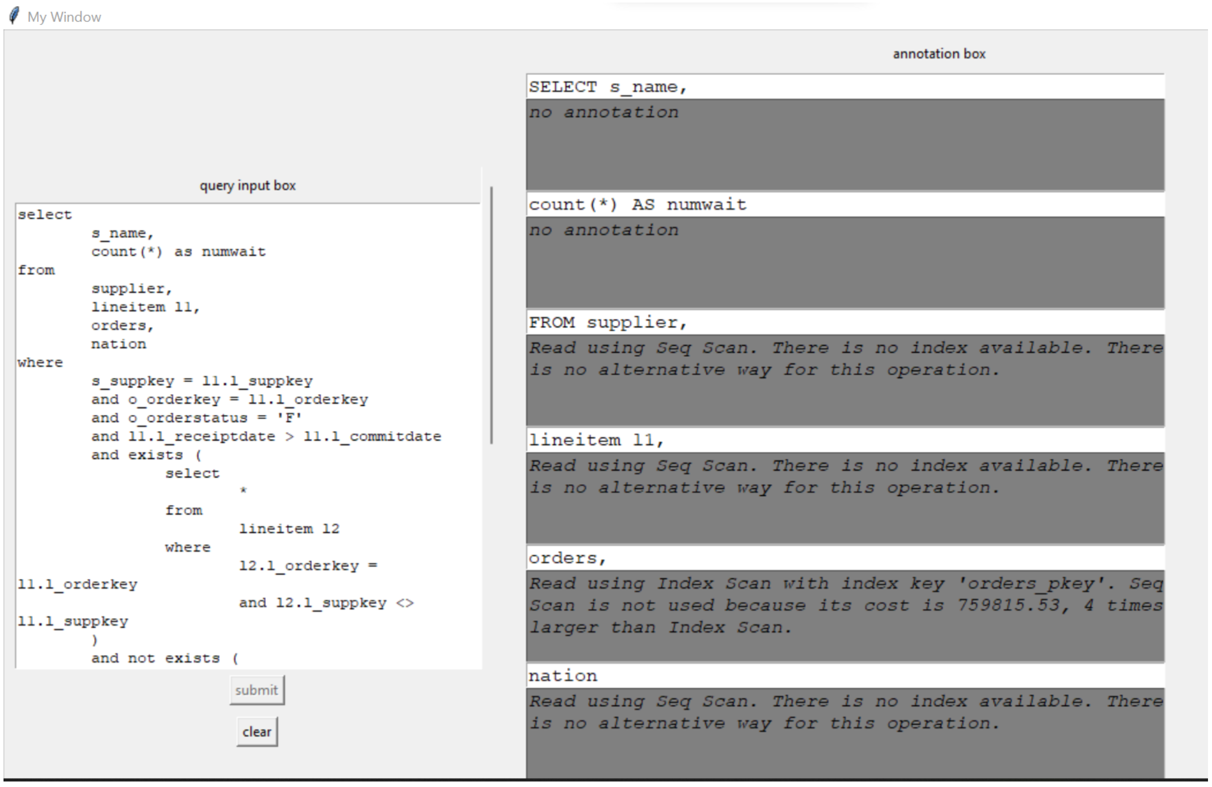
Example 1:



```
select
        c_count,
        count(*) as custdist
from
        (
                select
                        c_custkey,
                        count(o_orderkey) as
c_count
                from
                        customer left outer join
orders on
                                c_custkey =
o_custkey
                                and o_comment not
like '%special%deposits%'
                group by
                        c_custkey
        ) c_orders
group by
        c_count
order by
        custdist desc,
        c_count desc;
```

```
SELECT c_count,
no annotation

count(*) AS custdist
no annotation

FROM
no annotation

(SELECT c_custkey,
no annotation

count(o_orderkey) AS c_count
no annotation

FROM customer
Read using Seq Scan. Index Scan is not used because its
cost is 258515.54, 2 times larger than Seq Scan.
```

Example 2:



```
select
        s_name,
        count(*) as numwait
from
        supplier,
        lineitem l1,
        orders,
        nation
where
        s_suppkey = l1.l_suppkey
        and o_orderkey = l1.l_orderkey
        and o_orderstatus = 'F'
        and l1.l_receiptdate > l1.l_commitdate
        and exists (
                select
                        *
                from
                        lineitem l2
                where
                        l2.l_orderkey =
l1.l_orderkey
                        and l2.l_suppkey <>
l1.l_suppkey
        )
        and not exists (
```

```
SELECT s_name,
no annotation

count(*) AS numwait
no annotation

FROM supplier,
Read using Seq Scan. There is no index available. There
is no alternative way for this operation.

lineitem l1,
Read using Seq Scan. There is no index available. There
is no alternative way for this operation.

orders,
Read using Index Scan with index key 'orders_pkey'. Seq
Scan is not used because its cost is 759815.53, 4 times
larger than Index Scan.

nation
Read using Seq Scan. There is no index available. There
is no alternative way for this operation.
```