NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

# CZ4031 Database System Principles

## Group 29 - Project 2 Report

| | |
|---|---|
| Lim Jin Yang | U1921682L |
| Tan Hu Soon | U1921282A |
| Vivien Chew | U1922718D |
| Tan Wen Xiu | U1921771H |

# Table of Contents

# 1. Introduction

A Structured Query Language (SQL) query is used to retrieve meaningful and relevant information from databases. Database management systems (DBMS) softwares such as SQL server, PostgreSQL or MySQL will execute a query execution plan (QEP) to process a query. The system will usually return only the results of the user's query with no indication on how these operations are being carried out and the steps being taken to process such a query. Although we can visualize the QEP using the DBMS software, the SQL query and the QEP view are disconnected from each other.

Hence, in this project, we aim to design an annotated SQL query that shows how the query and its query execution plan (QEP) are connected. The annotation shows how each of the different components of the query are being executed with relevant details displayed, giving our users a better understanding of the actual process being carried out. To achieve this, our group has implemented algorithms to identify and process the raw query plan output in addition to using visualization tools such as tkinter to come up with easily understandable graphs, representing the query plans used. More of this will be explained in the later part of the report.

## 1.1 The TPC-H Dataset

The dataset we will be using is the TPC Benchmark H (TPC-H) which contains 8 separate individual tables. The TPC-H dataset comprises a set of business queries designed to exercise system functionalities which are representative of complex business analysis applications. A summary of the dataset is as follows:

| **TPC-H relation** | **Description** | **Cardinality (No.of rows)** |
|---|---|---|
| PART | Contains list of parts | 200000 |
| SUPPLIER | Contains list of suppliers | 10000 |
| PARTSUPP | Contains information about parts sold | 800000 |
| CUSTOMER | Contains list of customers and | 150000 |
| ORDERS | Contains orders and customer information | 1500000 |
| LINEITEM | Contains order items in each order | 6000000 |
| NATION | Contains list of nations | 25 |
| REGION | Contains list of regions that nations are in | 5 |

We will also use the queries from the TPC-H dataset file given to test out our algorithm.

# 2. PostgreSQL Database Server

## 2.1 Dataset Pre-processing

After retrieving the .tbl files, we converted them to .csv files with our tbl_to_csv function, to be loaded into the PostgreSQL database. This enables us to execute queries for the database on PostgreSQL.

## 2.2 Connecting PostgreSQL database server

Psycopg 2 allows client programs to pass queries to the PostgreSQL backend server and receive the results of these queries. We will use the Psycopg2 package to connect to the PostgreSQL database server using Python as our programming language.

Within our get_json function, we will connect to the database server with our credentials. After successful connection, we are able to pass input queries to PostgreSQL.

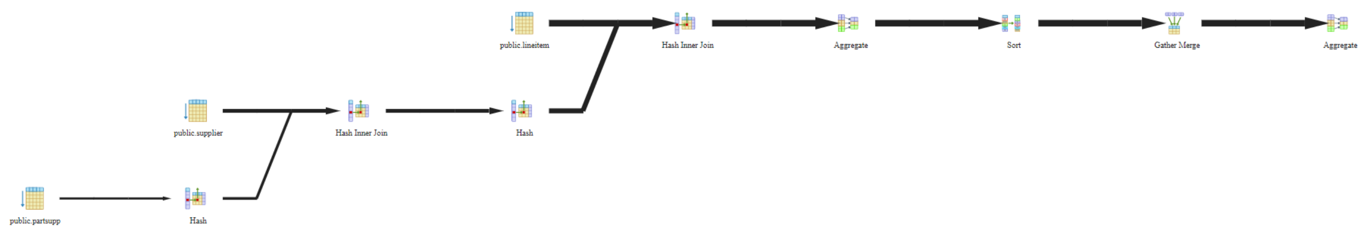
## 2.3 Retrieval of QEP from PostgreSQL

The DBMS software generates a QEP to execute the query. We are able to retrieve the QEP used by PostgreSQL through using the built-in EXPLAIN ANALYZE command such that the output is returned in a JSON format.

The EXPLAIN command displays the execution plan that the PostgreSQL planner generates for the query. The execution plan includes information about how the relation tables that are referenced by the query will be scanned and how operations are carried out. The estimated statement execution cost, which is an estimation of how long it will take the statement to run will be shown. This is the most crucial information as it is how the planner chooses an algorithm for an operation. The planner will choose the algorithm with the smallest estimated start-up time to be used. We are only allowed to access what PostgreSQL believes to be the most optimal plan and this is the QEP that we will make use of in the subsequent sections.

The ANALYZE option allows the statement to be actually executed, not only planned. The algorithm used and total elapsed time for the operation to be executed is returned.

After execution of the query using the EXPLAIN ANALYZE command, the graphical analysis of the query will be outputted on postgreSQL. By default, PostgreSQL uses a right-deep join tree.



Graphical Analysis of a Query

We execute the query using an additional command: EXPLAIN(ANALYZE, VERBOSE, FORMAT JSON) to retrieve the json file of this output and use it in the creation of our tree and annotations.

## 3. Algorithm

### 3.1 Visualization of QEP (in a tree-structured view)

We did the search by using built-in pre-order iteration through the Query Plan, in JSON format that was previously retrieved from POSTGRESQL.

Firstly, we build a tree using the json file. A plan is interpreted as a node in the tree. The first plan is the root node of the tree. The build_tree function is used recursively to retrieve all the child nodes of a previous node. We adopted the same algorithm used by PostgreSQL to build our tree.

### 3.2 Retrieval of Annotation for each Node

Given a node, we want to retrieve the relational tables and the attributes that were involved in the operation. For example, if a "Hash Join" method was used to execute a join, we want to find out which attributes were joined using this method.

We note that there are a few types of common methods used by PostgreSQL, hence we create a static dictionary object to keep track of the attributes that we want to annotate. For example, for a join operation, the methods "Hash Join" or "Merge Join" are frequently used.

In the static dictionary ATTRIBUTE, the keys of the dictionary are the methods used, while the values contain a list of related attributes that we would want to retrieve from the query plan (Json format). In the case of the "Hash Join" method, we will retrieve "Hash Cond" which displays the condition on which the hash was done and "Output" which tells us the attribute that was returned from the join. This static dictionary, created for all the frequently seen methods, will be stored in the node_types.py file.

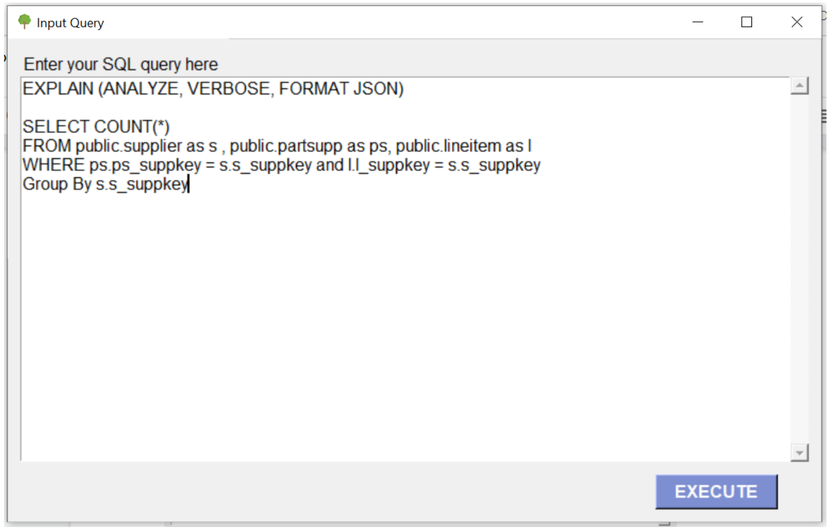
By creating a few dictionaries under node_types.py that are essential in our analysis of the query, it will then return us a list of matched nodes or None if no node was matched in that of our dictionary.

```
ATTRIBUTE = {'LIMIT': ['Plan Rows'], 'SORT': ['Sort Method', 'Sort Key'], 'NESTED
LOOP': [], 'MERGE JOIN': ['Merge Cond'],
           'HASH': ['Output'], 'HASH JOIN': ['Hash Cond', 'Output'], 'AGGREGATE':
['Group Key'], 'HASHAGGREGATE': ['Group Key'],
           'SEQ SCAN': ['Relation Name'], 'INDEX SCAN': ['Index Cond'], 'GATHER
MERGE': ['Output'],
           'INDEX ONLY SCAN': ['Index Cond'],
           'BITMAP HEAP SCAN': ['Recheck Cond'],
           'BITMAP INDEX SCAN': ['Index Cond'],
           'CTE SCAN': ['Index Cond']}
```

## 4. Graphical User Interface (GUI)

For our GUI, it is developed to be as intuitive as possible for the user to minimize any possible confusion that might arise when using the program. We used Tkinter which is the standard GUI library for Python, to develop our interface.

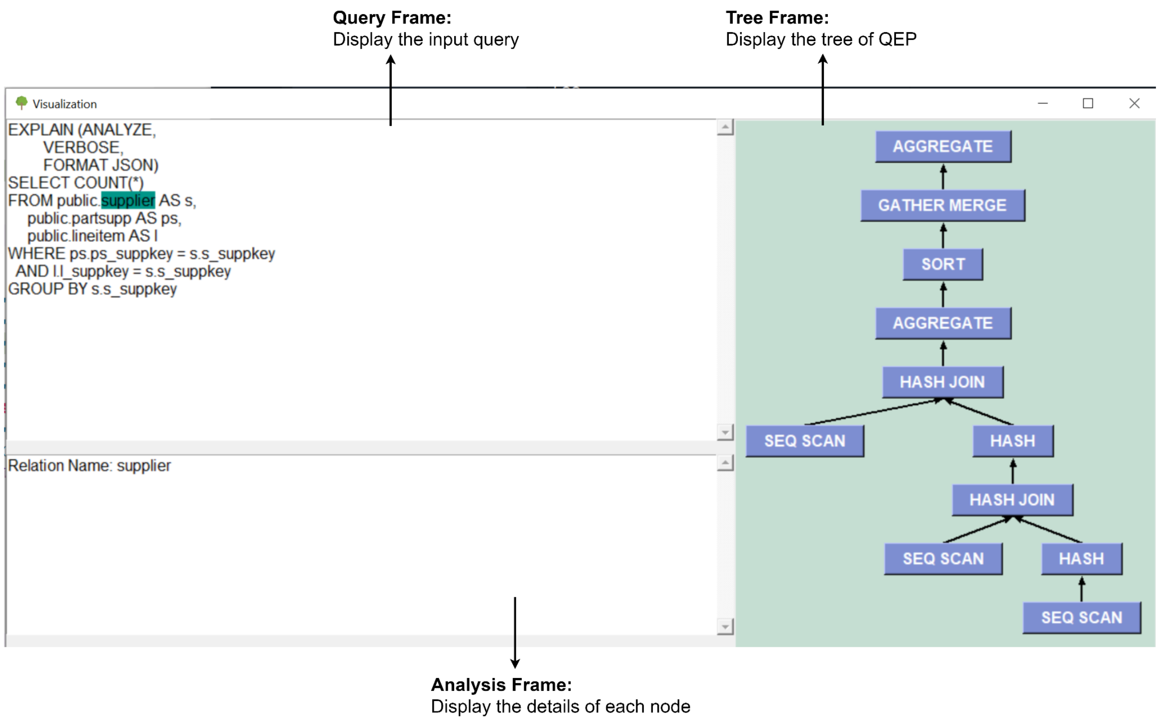
Upon running the program, the 'Input Query' window will appear, allowing the user to enter their SQL query. The program will then output the results of the query when the user clicks on the 'Execute' button. On the visualization window, we divided the page into 3 frames to view the details of query results. Upon selection on each node, analysis details will be displayed. When hovering over some nodes, the corresponding component of the query which was retrieved from our algorithm described in the previous part, will be highlighted when possible.



```
EXPLAIN (ANALYZE, VERBOSE, FORMAT JSON)

SELECT COUNT(*)
FROM public.supplier as s , public.partsupp as ps, public.lineitem as l
WHERE ps.ps_suppkey = s.s_suppkey and l.l_suppkey = s.s_suppkey
Group By s.s_suppkey
```

'Input Query' Window

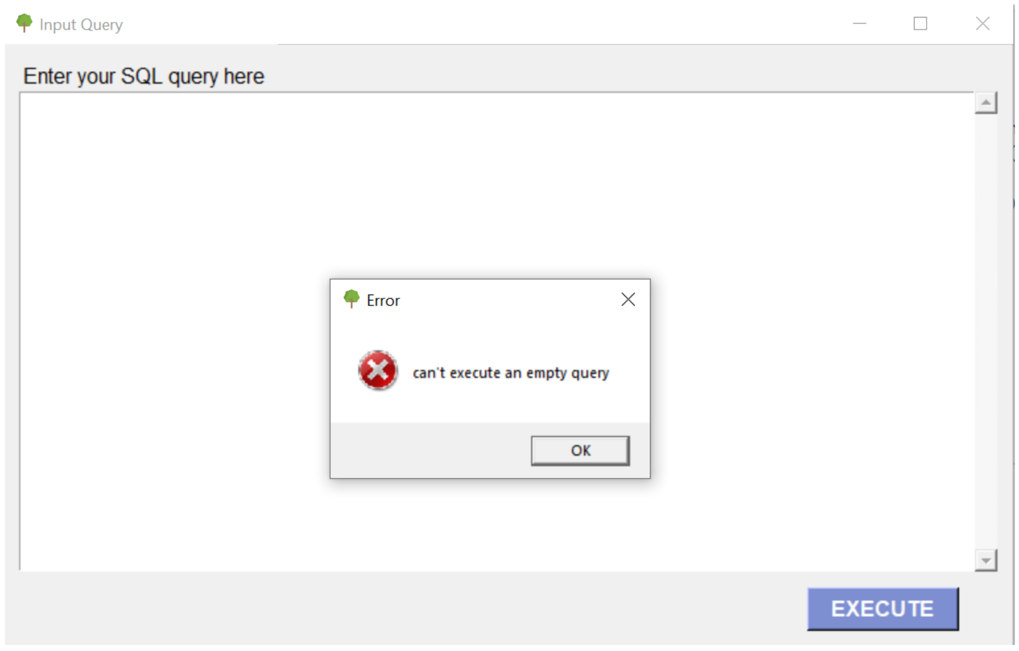


```
EXPLAIN (ANALYZE,
     VERBOSE,
     FORMAT JSON)
SELECT COUNT(*)
FROM public.supplier AS s,
   public.partsupp AS ps,
   public.lineitem AS l
WHERE ps.ps_suppkey = s.s_suppkey
 AND l.l_suppkey = s.s_suppkey
GROUP BY s.s_suppkey
```

Relation Name: supplier

'Visualization' Window

We used a try-except block to catch all possible errors. In the event there is an error, a message will be displayed to inform the user of the error. As we can see below, when we try to click the 'Execute' button with an empty input, a pop-up error message box will appear.



can't execute an empty query

'Error Message' Pop-Up

## 5. Limitations of the Software

There are many possible query plans that can be derived from a single query and the limitation of our software is that we are unable to determine the most optimal query plan to be used based on our own metrics. As such, our software assumes that the query plan presented by PostgreSQL is the most efficient and optimal solution possible.

There are 2 components in query processing, one being the selection of a suitable query plan and the other being the execution of the actual query. However, with the combinations of possible query plans available, it is not possible to iterate through every single plan, measuring their efficiency and optimality based on our own metrics. The cost and time saved upon finding the most optimal query plan will be outweighed by the time taken to complete such a task, making the whole overall process an inefficient one. There must therefore be certain tradeoffs when it comes to the selection of a suitable query plan, making use of heuristics to obtain a query plan which satisfies some level of efficiency that we are agreeable with and moving onto execution instead of searching through all possible plans. A method like this would minimize the costs taken for both the search and execution of the process and hence maximizing our query operation efficiency as a whole.

Another limitation our software faces is that the graph display is not compatible with some monitors. When the query plan chosen gets sufficiently complex with many nodes to be displayed, the graph will be cropped out if the screen is too small. A 24 inch monitor will be able to display the tree for all kinds of query plan without this problem as tested with all the queries given in the TPC-H file.

## 6. How to run the software

### 6.1 Libraries and dependencies

In Terminal, ensure you are in the folder of your repository. You will need to install all the following libraries:

1. pip install json
2. pip install tkinter
3. pip install sqlparse
4. pip install psycopg2
5. pip install anytree

All required packages must be installed to run the programme!

## 6.2 Loading the PostgreSQL Database

1. Create a new database named “TPC-H” in PostgreSQL
2. Create 8 relational tables with the query execution code included in appendix 7.1.
3. Import the data csv files from this google drive folder (https://drive.google.com/drive/folders/1ZTRsyR6EkxqbsNN5Es8uM0aQnmnazSut?usp=sharing) to populate the tables.
4. Update the password to your PostgreSQL account in the “interface.py” under the “connect” function **(need to key the password for your PostgreSQL in order to successfully connect and use the database)**:

```
        conn = psycopg2.connect(
            host="localhost",
            database="TPC-H",
            user="postgres",
            password="password")
```

## 6.3 Instruction Guide to run the software

1. Run the 'project.py' file
2. Enter your query into the input and press 'Execute' button
3. You can hover or click on any nodes of the tree to view the annotations.
4. When visualization and analysis is done, you can close the ‘Visualization’ window and enter another query of your choice in the ‘Input Query’ window.

# 7. Appendix

### 7.1 Create tables in PostgreSQL using the query execution code:

```
CREATE TABLE CUSTOMER(
            C_CUSTKEY INTEGER NOT NULL,
            C_NAME CHARACTER VARYING(25) NOT NULL,
            C_ADDRESS CHARACTER VARYING(40) NOT NULL,
            C_NATIONKEY INTEGER NOT NULL,
            C_PHONE CHARACTER(15) NOT NULL,
            C_ACCTBAL NUMERIC(15,2) NOT NULL,
            C_MKTSEGMENT CHARACTER(10) NOT NULL,
            C_COMMENT CHARACTER VARYING(117) NOT NULL
)
```

```
CREATE TABLE ORDERS  ( O_ORDERKEY       INTEGER NOT NULL,
                        O_CUSTKEY        INTEGER NOT NULL,
                        O_ORDERSTATUS    CHAR(1) NOT NULL,
                        O_TOTALPRICE     DECIMAL(15,2) NOT NULL,
                        O_ORDERDATE      DATE NOT NULL,
                        O_ORDERPRIORITY  CHAR(15) NOT NULL,
                        O_CLERK          CHAR(15) NOT NULL,
                        O_SHIPPRIORITY   INTEGER NOT NULL,
                        O_COMMENT        VARCHAR(79) NOT NULL);

CREATE TABLE LINEITEM ( L_ORDERKEY    INTEGER NOT NULL,
                          L_PARTKEY     INTEGER NOT NULL,
                          L_SUPPKEY     INTEGER NOT NULL,
                          L_LINENUMBER  INTEGER NOT NULL,
                          L_QUANTITY    DECIMAL(15,2) NOT NULL,
                          L_EXTENDEDPRICE  DECIMAL(15,2) NOT NULL,
                          L_DISCOUNT    DECIMAL(15,2) NOT NULL,
                          L_TAX         DECIMAL(15,2) NOT NULL,
                          L_RETURNFLAG  CHAR(1) NOT NULL,
                          L_LINESTATUS  CHAR(1) NOT NULL,
                          L_SHIPDATE    DATE NOT NULL,
                          L_COMMITDATE  DATE NOT NULL,
                          L_RECEIPTDATE DATE NOT NULL,
                          L_SHIPINSTRUCT CHAR(25) NOT NULL,
                          L_SHIPMODE     CHAR(10) NOT NULL,
                          L_COMMENT      VARCHAR(44) NOT NULL);

CREATE TABLE NATION  ( N_NATIONKEY  INTEGER NOT NULL,
                         N_NAME       CHAR(25) NOT NULL,
                         N_REGIONKEY  INTEGER NOT NULL,
                         N_COMMENT    VARCHAR(152));

CREATE TABLE REGION  ( R_REGIONKEY  INTEGER NOT NULL,
                         R_NAME       CHAR(25) NOT NULL,
                         R_COMMENT    VARCHAR(152));

CREATE TABLE PART  ( P_PARTKEY     INTEGER NOT NULL,
                       P_NAME        VARCHAR(55) NOT NULL,
                       P_MFGR        CHAR(25) NOT NULL,
                       P_BRAND       CHAR(10) NOT NULL,
                       P_TYPE        VARCHAR(25) NOT NULL,
                       P_SIZE        INTEGER NOT NULL,
                       P_CONTAINER   CHAR(10) NOT NULL,
                       P_RETAILPRICE DECIMAL(15,2) NOT NULL,
                       P_COMMENT     VARCHAR(23) NOT NULL );

CREATE TABLE SUPPLIER ( S_SUPPKEY     INTEGER NOT NULL,
                          S_NAME        CHAR(25) NOT NULL,
                            S_ADDRESS     VARCHAR(40) NOT NULL,
                            S_NATIONKEY   INTEGER NOT NULL,
                            S_PHONE       CHAR(15) NOT NULL,
                            S_ACCTBAL     DECIMAL(15,2) NOT NULL,
                            S_COMMENT     VARCHAR(101) NOT NULL);

CREATE TABLE PARTSUPP ( PS_PARTKEY     INTEGER NOT NULL,
                            PS_SUPPKEY     INTEGER NOT NULL,
                            PS_AVAILQTY    INTEGER NOT NULL,
                            PS_SUPPLYCOST  DECIMAL(15,2)  NOT NULL,
                            PS_COMMENT     VARCHAR(199) NOT NULL );
```