CZ4031 - Database System Principles
Project 2 Report - Query Visualiser

Bachhas Nikita - U1921630F

Kam Chin Voon - U1922278G

Kundu Koushani - U1922997B

Nanyang Technological University
November 2021

Table of Contents

Introduction

Large world real scale applications often consist of a huge amount of data stored in the form of a database and make use of query processing languages such as SQL to update or retrieve data according to a given query. A query execution plan (QEP) is a sequence of steps used to access data in a relational database management system.

In order to improve the efficiency of query execution many modern database management systems make use of query optimizers. A query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query execution plans.

In this project, we have used Python as the programming language and PostgreSQL for the DBMS. The user can input a sql query and the outline of the query plan will be the output in natural language which the layman can understand.

The main aim of this project is to understand and visualize the performance of a query optimizer through the following tasks:

a) Generate and describe the plan for a set of queries using the TPC-H benchmark data and queries.
b) Design and implement a GUI that takes as input a query, retrieves its query execution plan and output query execution plan in natural language.

Tools Used

The following are a few tools that are used to help in the implementation of this project.

- **PostgreSQL**

  PostgreSQL is a relational database management system that supports SQL querying. It is often used as a backend database for websites and a transaction database for applications and products. PostgreSQL supports popular languages like Python, Java, and C++.

  In this experiment PostgreSQL is used as the database management system. In order to set up the PostgreSQL, each of the tables are copied one by one in sequence, as stated in the project description and then the .csv file is imported into the database.

- **TPC-H benchmark**

  The TPC-H benchmark data consists of a database and a set of queries that are designed to reflect the functionalities of real world online complex business analysis applications.

  The database consists of eight different tables, which will be elaborated in the section below.

- **psycopg2 library**

  Psycopg2 is a PostgreSQL database adapter implemented in C++ and designed for Python. Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety which enables several threads to share the same connection. It also features client-side and server-side cursors, asynchronous communication and notifications, “COPY TO/COPY FROM” support.

  Psycopg2 is used in the project to run the PostgreSQL queries in Python.

Objective

An output that allows the user to understand the steps and algorithm used by the query optimizer to better understand how the query is carried out. It should also generate a query plan in a tree format for easier visualisation for the user. This can allow the user to better plan the query that they input to the DBMS in order to allow more efficient retrieval and manipulation of records.

One such example is taking note of where sequential scans are occurring and the amount of time for the scan. If we note that the query is spending most time on a sequential scan of the database, adding indexes would help to speed up the query execution plan.

Approach

PostgreSQL also contains the EXPLAIN method which, when run, is able to generate the query execution plan for the specific query. The python scripts which connect to the database and get the relevant query execution plan in JSON will utilise this function in PostgreSQL. After retrieving the query in json, we build a tree from the json file, which will be able to show the flow of the query execution. The leaf nodes are the operations conducted on the database tables while the parent nodes are operations that use the children nodes and are therefore executed after the children nodes are executed.

Algorithm Design

The key algorithms that we implement in this project have been split into separate python files, namely: interface.py, annotation.py, preprocessing.py, and project.py. Each of these files are responsible for carrying out different parts of the functionalities required.

We have decided to visualise the query plan in a tree format, with each node representing the respective query plan carried out by the query plan optimizer at various times. The root node represents the last query plan executed while the deepest children node represents the query plan executed on the database.

These are a few of the classes and functions implemented:

interface.py

This file contains code for the GUI, and shows our implementation of the user interface using the tkinter package in python.
When executed, it prompts the user to enter their password and details of the database that they are querying on, after which, it outputs the corresponding Query Execution Plan (QEP) as a png file to the user.

project.py

This is the entry point of the program whereby, when run, calls on interface.py which is where the logic flow of the program starts from.

preprocessing.py

This file connects the user to PostgreSQL and retrieves the json formatted QEP from Postgresql and converts it into a tree format for visualisation.

Method	Function
def get_tree_node_pos(G, root=None, width=1.0, height=1, vert_gap=0.1, vert_loc=0, xcenter=0.5)[1]	Generating a hierarchical graph from networkx
class Node(query_plan, curr_node, count)	Takes in a query plan generated by postgresql in json, the parent node and an integer count. This class contains all attributes of the nodes, getting their types and creates an explanation of the

	query when called, etc.
class QueryPlan(query_json, raw_query)	Takes in the json query plan generated by PostgreSQL and raw query input by the user in the GUI. Draw the graph for the QEP.
class QueryRunner(pwd, hostname = 'localhost', database = 'TPC-H', username = 'postgres', port_img_nameid ="5432")	Takes in user input values for pwd, hostname, database, username and portid in order to set up connection with the TPC-H database in PostgreSQL

annotation.py

This contains code for generating the annotations.
Upon execution, this outputs a step-by-step process describing each statement in the query and how every operation is carried out.

Method	Function
def explainer_map(query_plan, node)	Takes input from create_explaination method in class Node; the query plan, as well as the instance of the node. Generates an explanation based on the node type of the query plan passed

Flow of program

When the user first starts the program, they will be prompted to input their password as well as the database they wish to query. By default, if the attributes of the database are left blank, the local host TPC-H database with port id 5432 will be queried.

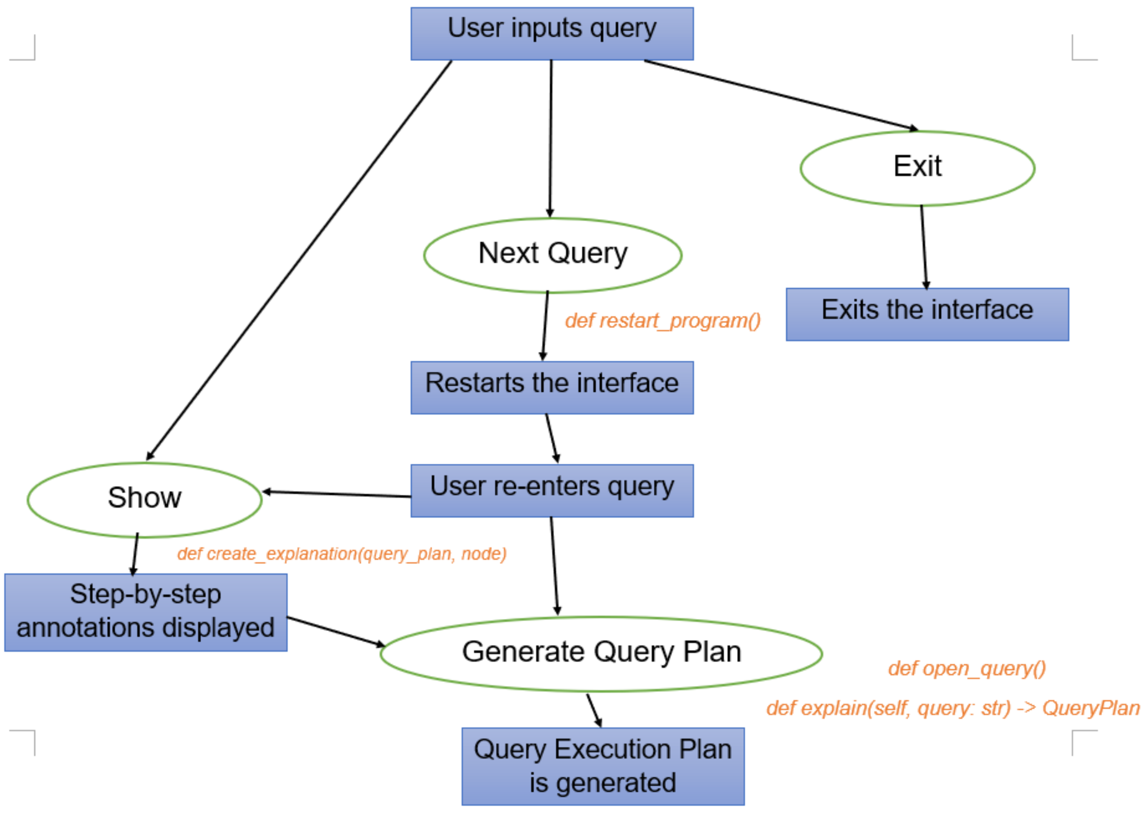


Fig.1 Steps involved to generate the output by the GUI

The functions illustrated in the figure above include:

- def restart_program(): this function restarts the current program and cleans up all the file objects
- def create_explanation(query_plan, node): creates a step-by-step annotation of the query which is displayed to the user upon clicking the 'Show' button
- def open_query(): this function opens the png image file for the specific Query Execution Plan, once the user clicks on the 'Generate Query Plan' button.
- def explain(self, query: str) -> QueryPlan: this will take the query as the input string and then generate the Query Execution Plan for the specific query

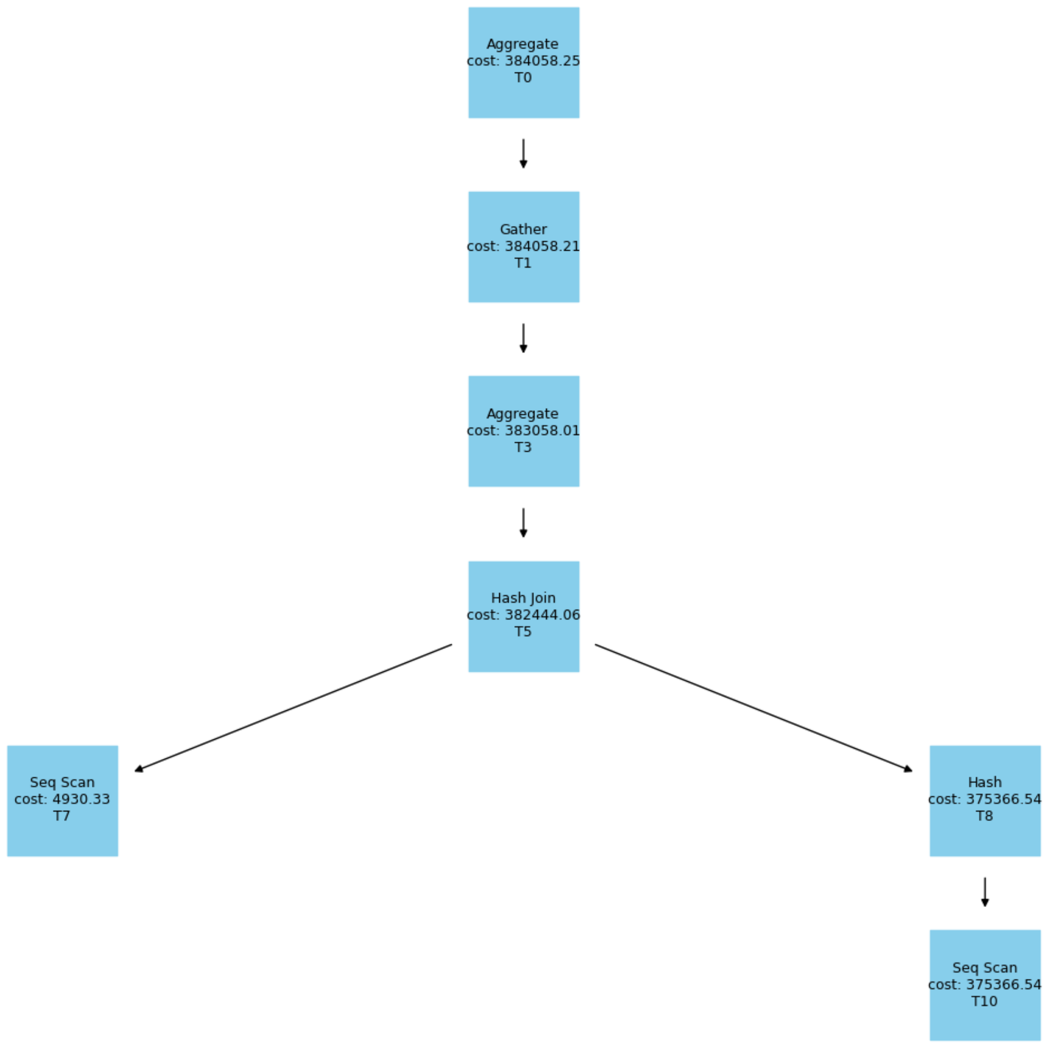
Test

PostgreSQL contains the database (TPC-H) that we will be querying. The database consists of the following eight different tables:

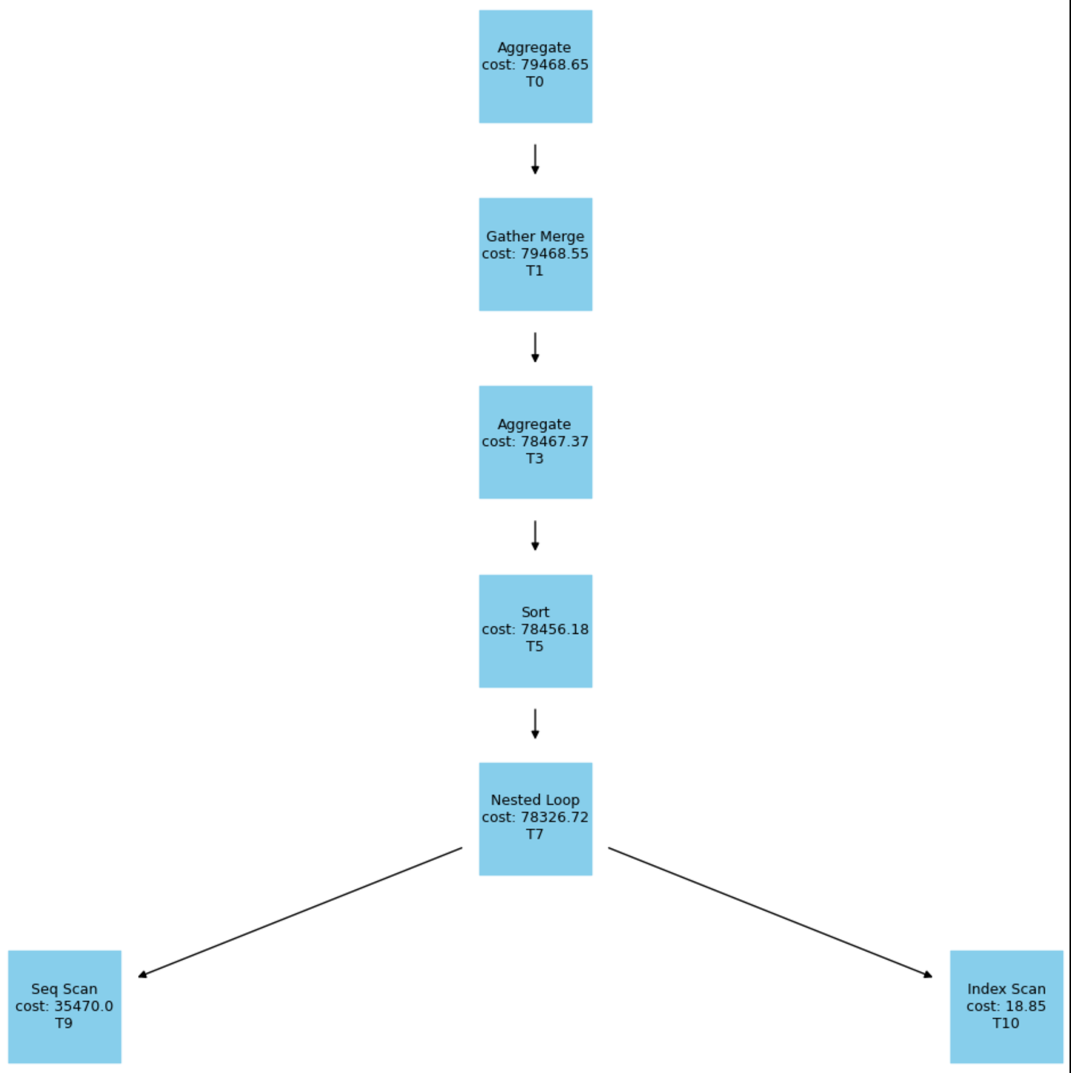
- Customer
    - custkey (primary key)
    - name
    - address
    - nationkey (foreign key on nation nationkey)
    - phone
    - acctbal
    - mktsegment
    - comment
- Lineitem
    - order key, part key, supp key, line number (primary key)
    - quantity
    - extendedprice
    - discount
    - tax
    - returnflag
    - linestatus
    - shipdate
    - commitdate
    - receiptdate
    - shipinstruct
    - shipmode
    - comment
- Nation
    - nationkey (primary key)
    - name
    - regionkey (foreign key on region regionkey)
    - comment
- Orders
    - orderkey (primary key)
    - custkey (foreign key on customer custkey)
    - orderstatus
    - totalprice
    - orderdate
    - orderpriority
    - clerk
    - shippriority
    - comment

- Part
    - partkey (primary key)
    - name
    - mfgr
    - brand
    - type
    - size
    - container
    - retailprice
    - comment
- Partsupp
    - partkey (foreign key on part (p_partkey)), suppkey (foreign key on supplier (s_suppkey)) (primary key)
    - availqty
    - supplycost
    - comment
- Region
    - regionkey (primary key)
    - name
    - comment
- Supplier
    - suppkey (primary key)
    - name
    - address
    - nationkey (foreign key on nation nationkey)
    - phone
    - acctbal
    - comment

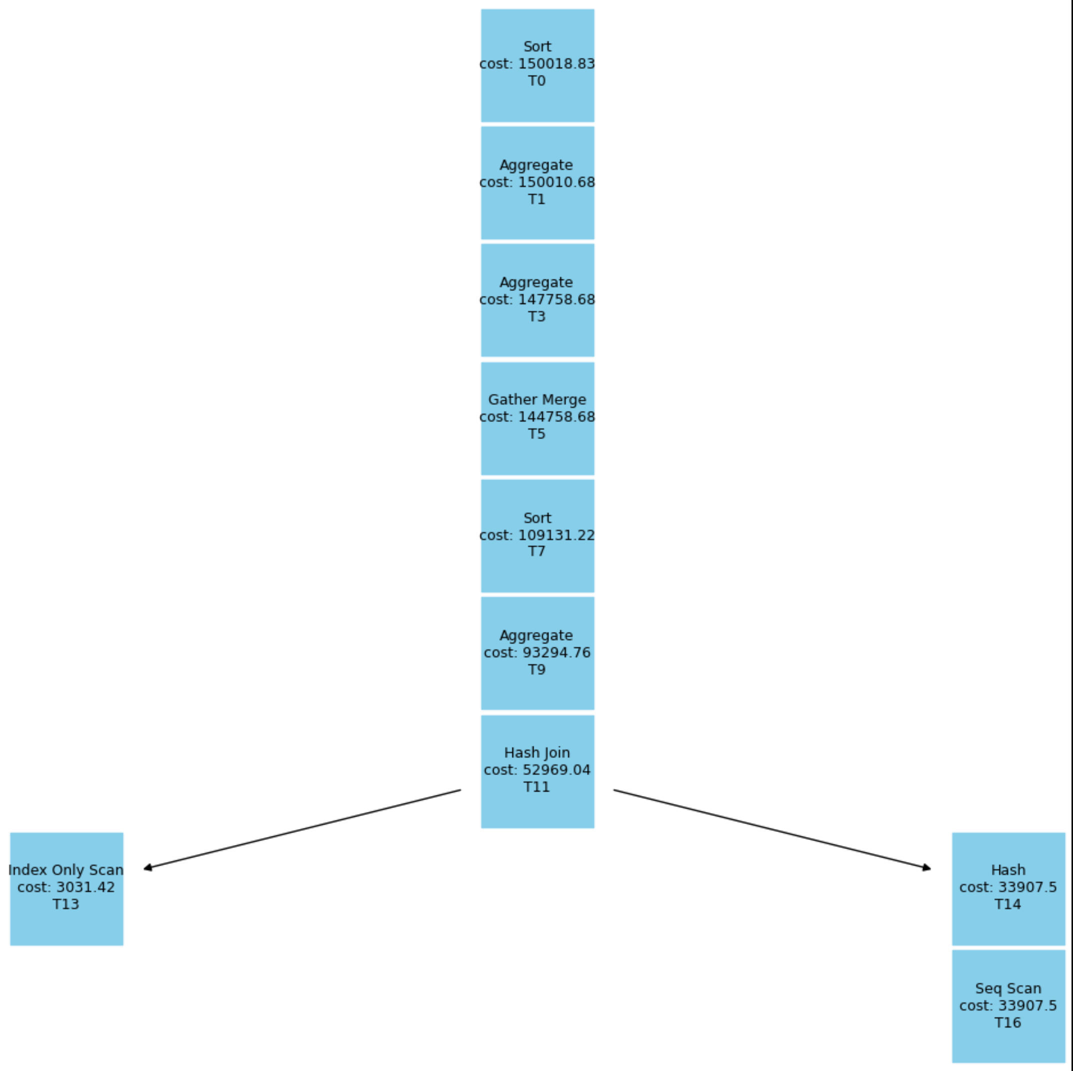
Test Cases and Output

Query no.	Query plan visualisation and Sample query
1.	 Aggregate cost: 384058.25 T0 Gather cost: 384058.21 T1 Aggregate cost: 383058.01 T3 Hash Join cost: 382444.06 T5 Seq Scan cost: 4930.33 T7 Hash cost: 375366.54 T8 Seq Scan cost: 375366.54 T10 select 100.00 * sum(case when p_type like 'PROMO%' then l_extendedprice * (1 - l_discount) else 0 end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue from lineitem, part where l_partkey = p_partkey and l_shipdate >= date '1994-03-01' and l_shipdate < date '1994-03-01' + interval '1' month;

2.



```
select
        o_orderpriority,
        count(*) as order_count
from
        orders
where
        o_orderdate >= date '1997-07-01'
        and o_orderdate < date '1997-07-01' + interval '3' month
        and exists (
                select
                        *
                from
                        lineitem
                where
                        l_orderkey = o_orderkey
                        and l_commitdate < l_receiptdate
        )
group by
        o_orderpriority
```

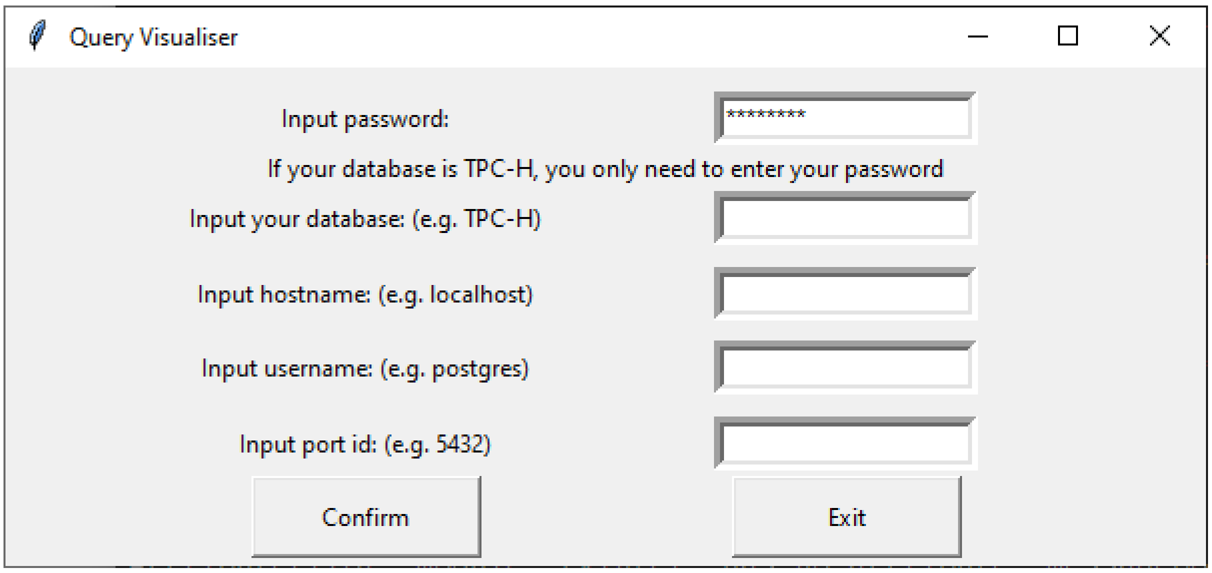
	order by o_orderpriority;
3.	 Sort cost: 150018.83 T0 Aggregate cost: 150010.68 T1 Aggregate cost: 147758.68 T3 Gather Merge cost: 144758.68 T5 Sort cost: 109131.22 T7 Aggregate cost: 93294.76 T9 Hash Join cost: 52969.04 T11 Index Only Scan cost: 3031.42 T13 Hash cost: 33907.5 T14 Seq Scan cost: 33907.5 T16 select c_count, count(*) as custdist from (select c_custkey, count(o_orderkey) from customer left outer join orders on c_custkey = o_custkey and o_comment not like '%express%packages%' group by c_custkey) as c_orders (c_custkey, c_count) group by c_count

	order by
	        custdist desc,
	        c_count desc;

Visualisation of SQL Queries

Query visualisation is important so that we can achieve a better understanding of how the database interprets the database schema so that as the developers of the database, we can more efficiently speed up queries. In order to visualise the SQL Queries in this project, we have designed and implemented a user-friendly graphical user interface (GUI) using the tkinter library in Python. The GUI is designed in such a way that it will take in a query as input from the user, and then output the corresponding step by step Query Execution Plan (QEP) by PostgreSQL. The QEP is visualised into a graphical representation of the operations performed by the database engine in order to return the information required by the user's query.

The screenshot below shows an illustration of the GUI that we have designed:



Input password: ********

If your database is TPC-H, you only need to enter your password

Input your database: (e.g. TPC-H)

Input hostname: (e.g. localhost)

Input username: (e.g. postgres)

Input port id: (e.g. 5432)

Fig 2. Prompt user for database information (can be left blank and default will apply)

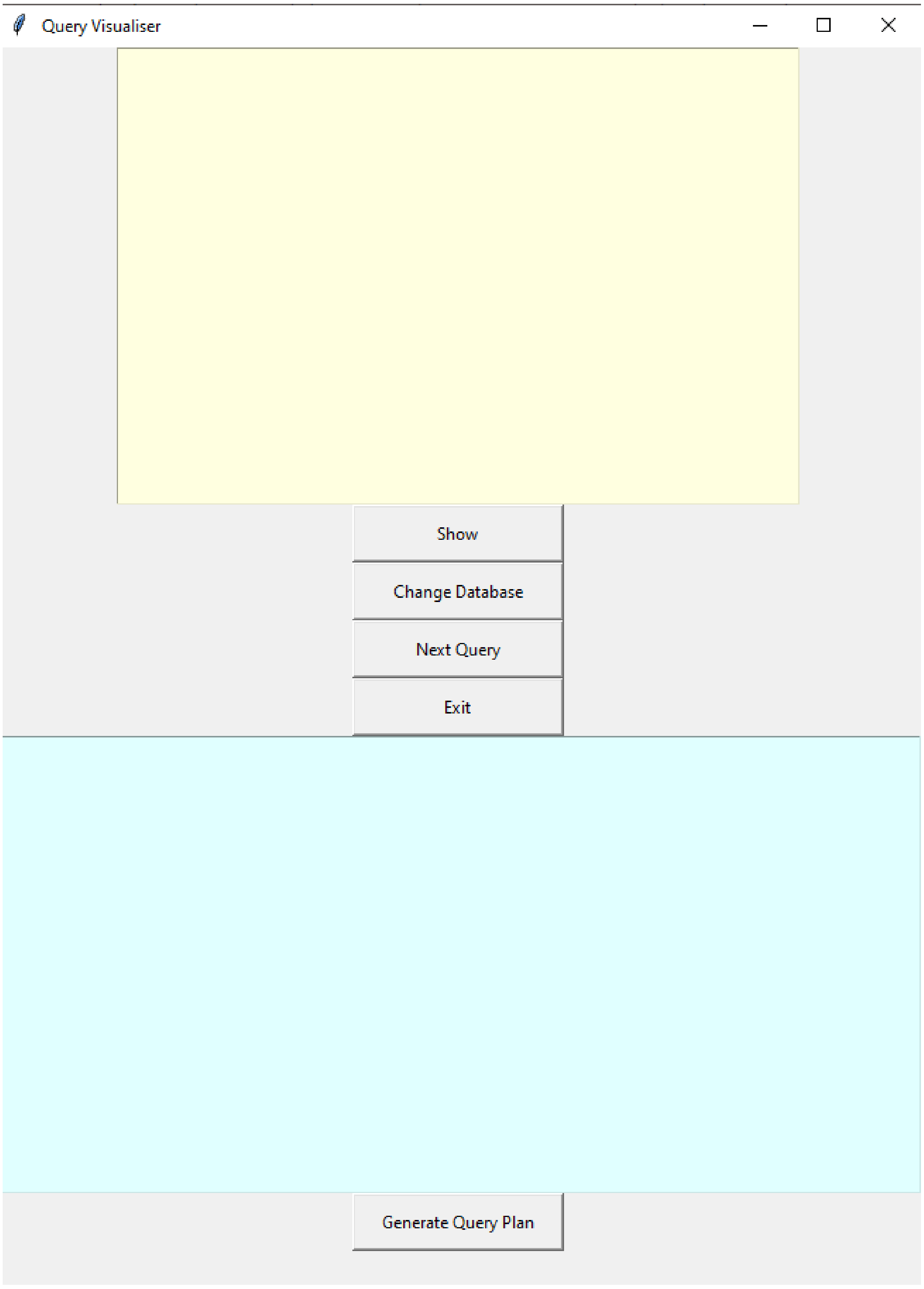
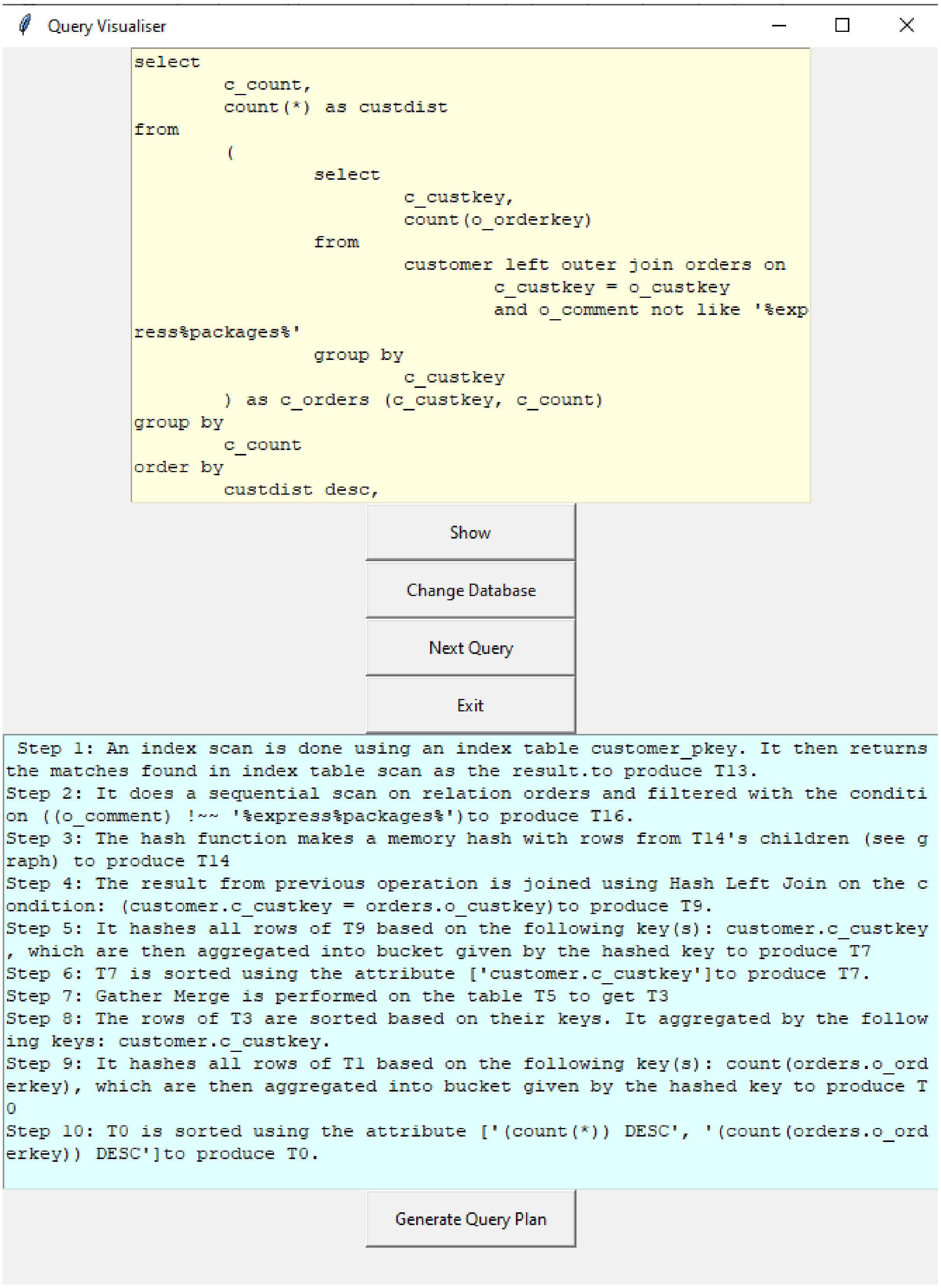


Figure 3: Main Application Window for Users to Input in their Queries
Yellow Box: Input
Blue Box: Output



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

```
 Step 1: An index scan is done using an index table customer_pkey. It then returns
the matches found in index table scan as the result.to produce T13.
Step 2: It does a sequential scan on relation orders and filtered with the conditi
on ((o_comment) !~~ '%express%packages%')to produce T16.
Step 3: The hash function makes a memory hash with rows from T14's children (see g
raph) to produce T14
Step 4: The result from previous operation is joined using Hash Left Join on the c
ondition: (customer.c_custkey = orders.o_custkey)to produce T9.
Step 5: It hashes all rows of T9 based on the following key(s): customer.c_custkey
, which are then aggregated into bucket given by the hashed key to produce T7
Step 6: T7 is sorted using the attribute ['customer.c_custkey']to produce T7.
Step 7: Gather Merge is performed on the table T5 to get T3
Step 8: The rows of T3 are sorted based on their keys. It aggregated by the follow
ing keys: customer.c_custkey.
Step 9: It hashes all rows of T1 based on the following key(s): count(orders.o_ord
erkey), which are then aggregated into bucket given by the hashed key to produce T
0
Step 10: T0 is sorted using the attribute ['(count(*)) DESC', '(count(orders.o_ord
erkey)) DESC']to produce T0.
```

Figure 4: With Input and Output

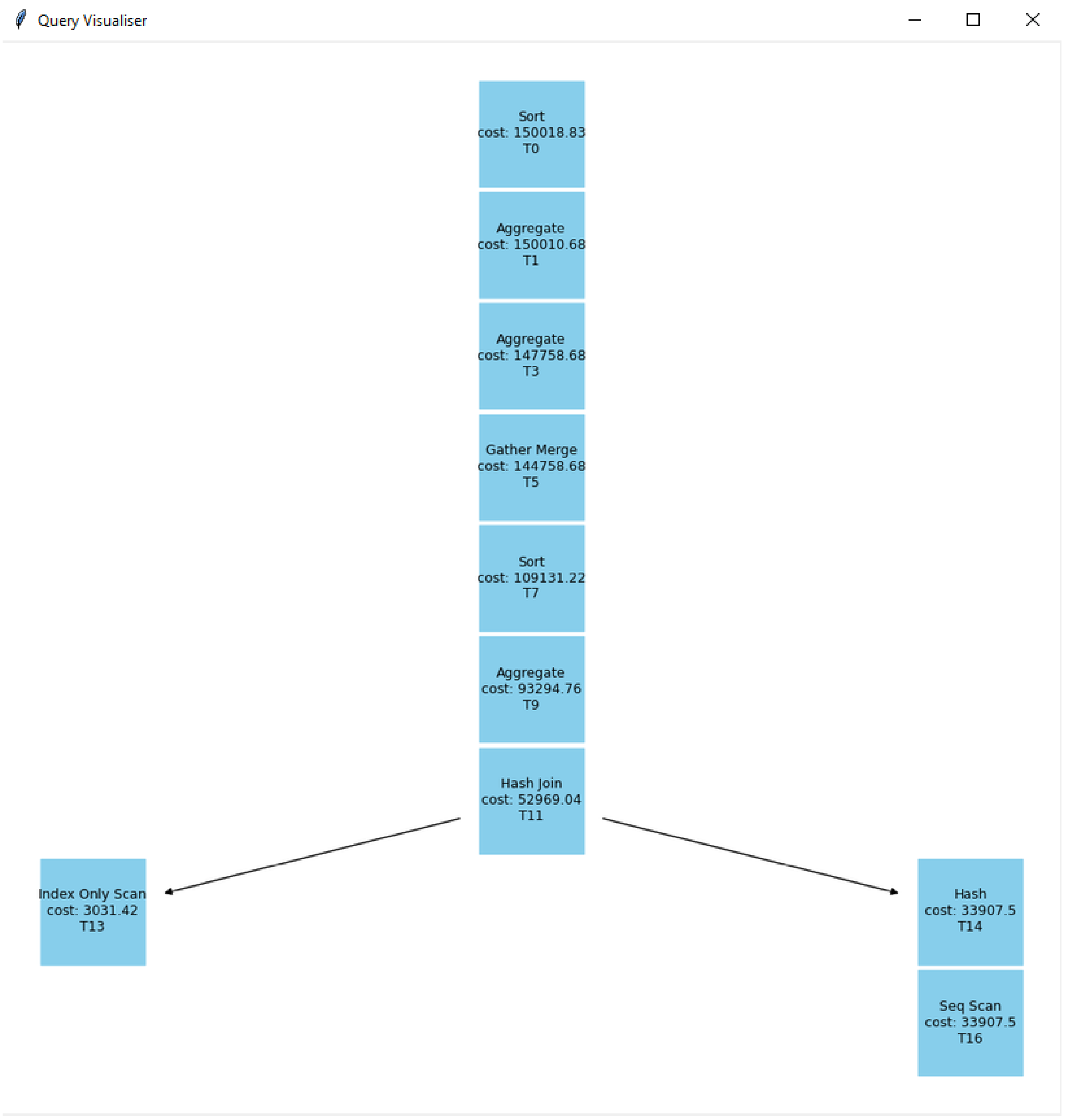


Figure 5: Generated Query Plan

Limitations of Software

The following are a few limitations of the software that we have identified:

1. Tested only on TPC-H Database
    a. The algorithm designed and implemented was only tested on the data in the TPC-H database. It would have been more ideal to test the algorithm on a range of databases to better observe the algorithm’s accuracy and efficacy
    b. A simple solution for solving this limitation would be to have different databases and SQL queries and run against them.

2. Limited Number of Error Checks
    a. Due to the limited number of error checks implemented, it is difficult to identify the location of the error.
    b. A possible solution for this would be to implement more exception handling and error handling by breaking the code into smaller chunks.

References

[1] From Joel's answer at https://stackoverflow.com/a/29597209/2966723.