

NANYANG
TECHNOLOGICAL
UNIVERSITY

CZ4031 Database System Principles

Project 2 - Connecting Your Query with Query Plans

Group 36

Gabriel Low Zhi You - U2022715G

Sanskriti Verma - U2023954E

Japhet Tan Zhi Rong - U2022037F

Peng Weixing - U1921133E

Zou Zeren - U2022422H

Abstract

Modern off-the-shelf database software does not provide annotations and visualisation when the user is writing or executing a SQL query, making it hard for Database System Principles students to understand what's happening behind each query or for database engineers to optimise their queries. In this project, we utilised PostgreSQL's built-in Query Planner module with different combinations of Planner Method Configurations to generate a Query Execution Plan (QEP) and Alternative Query Plans (AQPs). The plans are then stored in a tree-like data structure which preserves hierarchical information of the query execution sequence—each node in the tree stores cumulative execution costs and annotations generated by comparing QEP and AQPs. Lastly, the query execution tree, annotations, and plan cost are visualised via a user-friendly GUI written in Python. The software is tested using the TPC-H dataset.

Acknowledgement

We would like to thank our supervisors, Professor Sourav Saha Bhowmick and Professor Long Cheng, for their support and guidance throughout this project, which would not have been possible without them. We would also like to thank our friends and family who supported us in our 3 - 4 years of university and provided us with emotional and financial support.

Software Execution Instructions

Installing

It's recommended to create a virtualenv for the project first before installing the dependencies. In your console, run:

```
pip install -r requirements.txt
```

Running the project

Make sure you have python3 installed (software tested with python ver. > 3.6).

When running under macOS, please make sure dark mode is off.

CD to the root of the project folder and run:

```
python ./project.py
```

Connecting to database

You will be presented with a login screen on program start. Please refer to Chapter 3 for a detailed guide on our software.

Please contact peng0099@e.ntu.edu.sg if you face issues running the software.

Table of Contents

Abstract	2
Acknowledgement	3
Software Execution Instructions	4
Table of Contents	5
1. Introduction	6
1.1 Background and Motivation	6
1.2 Report Organization	6
2. Algorithm	7
2.1 Query Execution Plan Generation	7
2.2 Query Plan Preprocessing	7
2.3 Alternative Query Plans Generation	9
2.4 Proposed Algorithm	10
3. Graphical User Interface	12
3.1 Design and Implementation	12
4. Dataset & Database Schemas	18
5. Conclusion	19

1. Introduction

1.1 Background and Motivation

SQL query writers often write in the blind, i.e. they write SQL queries without understanding what's happening behind the scene. Students and junior database engineers often made the mistake of writing working but not optimised SQL queries. This is partially due to the lack of query visualisation tools available. In this project, we propose a user-friendly Graphical User Interface (GUI) for visualising SQL queries. Four steps achieve this:

- 1) User specifies a SQL query to visualise.
- 2) Generating QEP and AQPs using PostgreSQL Query Planner based on the query.
- 3) Comparing QEP with each AQP generates annotations to explain how each component in the query is executed and why a specific operator is chosen to execute the component.
- 4) Visualise the plans in a hierarchical tree, with each operation as a node of the tree and fully annotated.

Our software is then tested using the TPC-H database schemas and dataset, loaded in a PostgreSQL database.

1.2 Report Organization

The report is organised as follows:

1. Chapter 1 provides a brief overview of the problem and the motivation behind this problem.
2. Chapter 2 discusses the preprocessing steps and the proposed algorithm.
3. Chapter 3 introduces the GUI of the software.
4. Chapter 4 describes the dataset and schemas used in this project.
5. Chapter 5 concludes the project by providing some limitations of the project.

2. Algorithm

This section first discusses the plan generation step and the preprocessing steps we took to process SQL queries. We then elaborate on the algorithm we used to generate AQPs and annotations.

2.1 Query Execution Plan Generation

PostgreSQL provides an *EXPLAIN* command that displays the execution plan that the Query Planner generates for the supplied SQL query. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and, if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table. The generated query can be rendered in plain text or JSON format.

	QUERY PLAN text
1	Limit (cost=151603.20..151603.20 rows=1 width=16)
2	-> Sort (cost=151603.20..151603.70 rows=200 width=16)
3	Sort Key: (count(*)), (count(orders.o_orderkey))
4	-> HashAggregate (cost=151600.20..151602.20 rows=200 width=16)
5	Group Key: count(orders.o_orderkey)
6	-> Finalize GroupAggregate (cost=111347.75..149350.20 rows=150000 width=12)

Figure 2.1.1: A query plan in text format generated from EXPLAIN command

In this project, query plans are generated without the *ANALYZE* option enabled, as *ANALYZE* will execute the query, and the process can be very slow as we are generating multiple alternative query plans at the later stage.

2.2 Query Plan Preprocessing

Plan generation using EXPLAIN

The generated plan from the *EXPLAIN* command can be rendered in JSON format. Each plan contains critical information such as the operator, cost, rows returned, and its child plans. Child plans are plans that are executed before the plan. For example, in figure 2.2.1, the

“Sort” operator will be executed first before the “Limit” operator. The query execution hierarchy and order are preserved this way.

```
{
  'Plan': {
    'Node Type': 'Limit',
    'Parallel Aware': False,
    'Async Capable': False,
    'Startup Cost': 291255.13,
    'Total Cost': 291255.14,
    'Plan Rows': 1,
    'Plan Width': 24,
    'Output': [
      'orders.o_orderpriority',
      '(count(*))'
    ],
  },
  'Plans': [
    {
      'Node Type': 'Sort',
      'Parent Relationship': 'Outer',
      'Parallel Aware': False,
      'Async Capable': False,
      'Startup Cost': 291255.13,
      'Total Cost': 291255.15,
      'Plan Rows': 5,
      'Plan Width': 24,
      'Output': [
        'orders.o_orderpriority',
        '(count(*))'
      ],
      'Sort Key': [
        'orders.o_orderpriority'
      ],
    },
  ],
}
```

Figure 2.2.1: A query plan in JSON format, showing the “child plan” hierarchy

PlanNode Data Structure

The raw JSON query plan returned from PostgreSQL contains a lot of redundant information, and the “child plan” structure is hard to interpret. In this project, we created our data structure “PlanNode” to represent the query plan returned from PostgreSQL. PlanNode is a binary tree-like data structure with three key attributes: cost, rows and children. Cost stores the plan cost, rows attribute stores number of output rows, while children store the child plans. With this data structure, each node in a tree can be easily accessed via depth-first search or breadth-first search.

```
Wei Xing
class PlanNode:
    """
    Base class for Plan nodes
    """
    type = "Base"

    Wei Xing
    def __init__(self, cost: float):
        self.cost = cost

        self.children: ["PlanNode"] = []
```

Figure 2.2.2: Implementation of “PlanNode” data structure

We also noticed that each node type could contain different attributes. For example, a Hash Join node type includes the attribute “Hash Cond” while a Merge Join node type contains a different attribute “, Merge Cond”. Different node types are also going to have different annotations on them. To handle this, we created multiple child classes of specific node types inherited from the parent class PlanNode. The child classes also implement the abstract method “get_annotation” to output possible annotations for that particular node type.

```

Wei Xing
class SortNode(PlanNode):
    type = "Sort"

Wei Xing
def __init__(self, cost: float, sort_keys: list):
    """
    Sort node
    :param cost: Total cost
    :param sort_keys: list of sort keys
    """
    super().__init__(cost)
    self.sort_keys = sort_keys

Wei Xing
def get_annotations(self) -> str:
    text = sort_annotation(self.sort_keys)

    return text

```

Figure 2.2.3: Example of “Sort” node

With the PlanNode data structure, raw query plans can be parsed easily and ready for use for the next step.

Node Annotation

Each node type has its annotations. These annotations are stored as templates in annotation.py. We created the annotation by referring to content from lectures and from:

<https://www.pgmustard.com/docs/explain>

2.3 Alternative Query Plans Generation

PostgreSQL does not provide a function to generate AQPs out of the box. Fortunately, it provides a “Planner Method Configuration” method, which turns a specific set of operators on and off when generating a query execution plan. These settings can be referred to at

<https://www.postgresql.org/docs/current/runtime-config-query.html#RUNTIME-CONFIG-QUERY-ENABLE>.

For example, adding “Set enable_hashjoin off;” before a SQL query can disable the usage of the Hash Join operator when generating the QEP. This way, we can generate multiple alternate versions of the original QEP of the query.

2.4 Proposed Algorithm

```
join_types = ["Hash Join", "Nested Loop", "Merge Join"]
scan_types = ["Seq Scan", "Bitmap Heap Scan", "Index Scan"]
unique_types = PlanNode.get_unique_node_types(self.qep)

# Generate AQP by limiting 1 operation type per plan
for t in join_types + scan_types:
    if t in unique_types:
        constraints = self.__prepare_constraints_query([t])
        q = f'{{constraints}} SET max_parallel_workers_per_gather = 0; EXPLAIN (VERBOSE, FORMAT JSON) ' + self.sql_query
        plan = self.__get_sql_query_plan(q)
        root = self.__build_tree_from_raw_plan(plan)
        self.alt_plan_names.append(t)
        PlanNode.compare_trees(self.qep, root) # mark diff

    if t in join_types:
        other_ops = [x for x in join_types if x != t]
    else:
        other_ops = [x for x in scan_types if x != t]
    annotation = extra_annotation(op=t, other_ops=other_ops, reduction=root.cost/self.qep.cost)
    self.extra_annotation[t] = annotation
    self.aqp[t] = root
```

Figure 2.4.1: Code snippet of our proposed algorithm

Our proposed algorithm for generating AQPs and annotation is as follows:

- 1) Use the SQL query provided by the user to generate the QEP.
- 2) Preprocess the QEP
- 3) Identify key operators in the generated QEP that could impact the plan cost significantly. In this project, we classified “Hash Join, Merge Join, and Nested Loop” as key operators for JOINS and “Seq Scan, Bitmap Heap Scan, and Index Scan” as key operators for SCANS.
- 4) For each key operator identified in point 3, we disable that operator using the Planner Method Configuration described in section 2.3, generating a different plan without

using that operator. This way, we can compare the effects of not using the operator with the original QEP.

- 5) The generated AQP will be compared with the QEP for structural differences. This is done by traversing each of the plan tree to identify node differences and mark the node that's different.
- 6) The comparison results from point 4 are saved into "extra_annotation". In this project, our comparison result is the cost difference between the original QEP and the AQP when a specific operator is disabled, as illustrated in figure 2.4.2. The "extra_annotation" will be appended to the specific operator of comparison in the QEP.

```
Hash Join is faster than using Nested Loop, Merge Join here, and achieved a cost  
reduction by at least 1.6x  
Seq Scan is faster than using Bitmap Heap Scan, Index Scan here, and achieved a cost  
reduction by at least 60596.8x
```

Figure 2.4.2: Example of comparison annotations for Hash Join and Sequential Scan

3. Graphical User Interface

3.1 Design and Implementation

In order to visualise the results, we designed and implemented a user-friendly graphical user interface (GUI). We first did an initial design using Figma as shown in Figure 3.1.1.

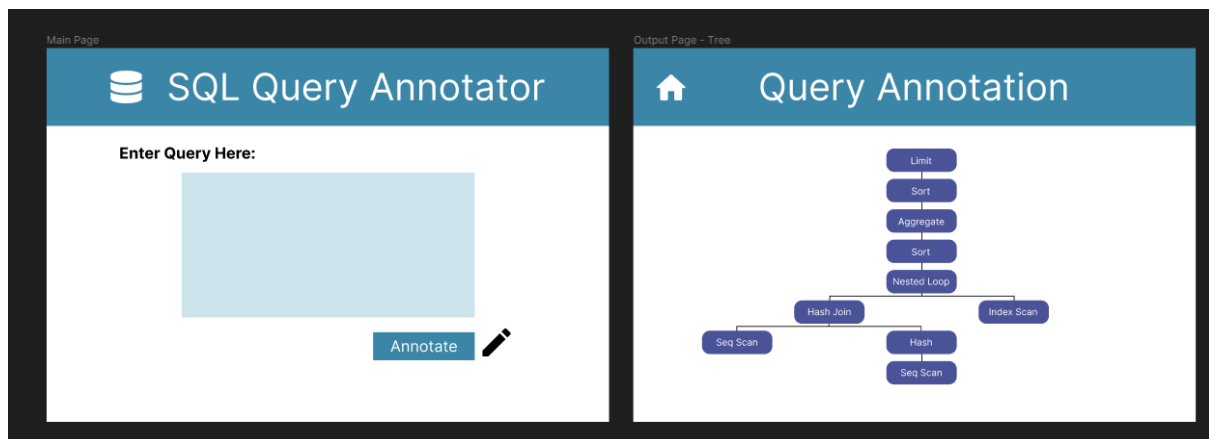
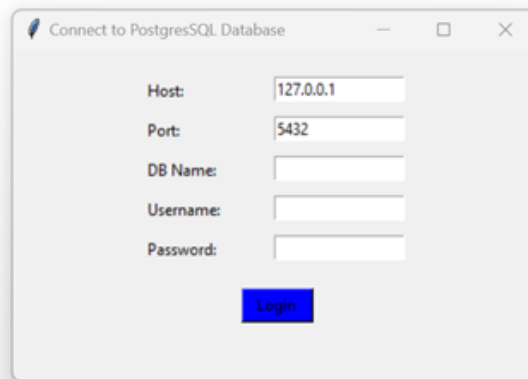


Figure 3.1.0: Figma of GUI Design

The goal was to create a simple, user-friendly design that any first-time user can navigate with ease. This was achieved with the help of accurately labelled page titles, interactive labelled buttons, scrollbars, and texts with instructions. We used Tkinter, the standard GUI library for Python to implement this design.



A screenshot of a GUI window titled "Connect to PostgreSQL Database". The window contains five input fields for connection details: Host (127.0.0.1), Port (5432), DB Name, Username, and Password. A blue "Login" button is positioned below the input fields.

Host:	127.0.0.1
Port:	5432
DB Name:	
Username:	
Password:	

Login

Figure 3.1.1: Login page GUI

Once the project.py file is run, the user is redirected to a login page. The Login page GUI includes 5 input entries- Host, Port, DB, Username, and Password. These are used to establish the connection to the hosted database.

SQL Query Annotator

Enter Query Here:

Annotate

Example SQLs:

SQL Query 1

SQL Query 2

SQL Query 3

SQL Query 4

Figure 3.1.2: Home page GUI

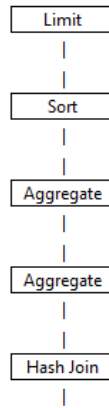
Figure 3.1.2, our Home page GUI, shows four components.

1. SQL Query Annotator's UI text.
2. The Input Box for creating custom SQL queries.
3. The user interface button for forwarding the input query to the preprocessing algorithm
4. A list of example SQL queries for presentation purposes.

[Back](#)

Query Annotation

Plan cost: 165029.78



```
select
  c_count,
  count(*) as custdist
from
  (
    select
      c_custkey,
      count(o_orderkey)
    from
```

Alt Plans:

[No Hash Join](#)[No Seq Scan](#)[Original Plan](#)

Figure 3.1.3: Output page GUI

The output page GUI contains three components.

1. The query plan component: With valid SQL entries into the Query Annotation tool, a query plan will be generated. The original query plan will be displayed as the default option.
2. The SQL entry component: Displaying the entry for the plans to be generated.
3. The alternative plan's components: consist of the alternative plan buttons for the sql query.

There are also horizontal and vertical scrollbars to ensure that the whole plan can be viewed, regardless of size.

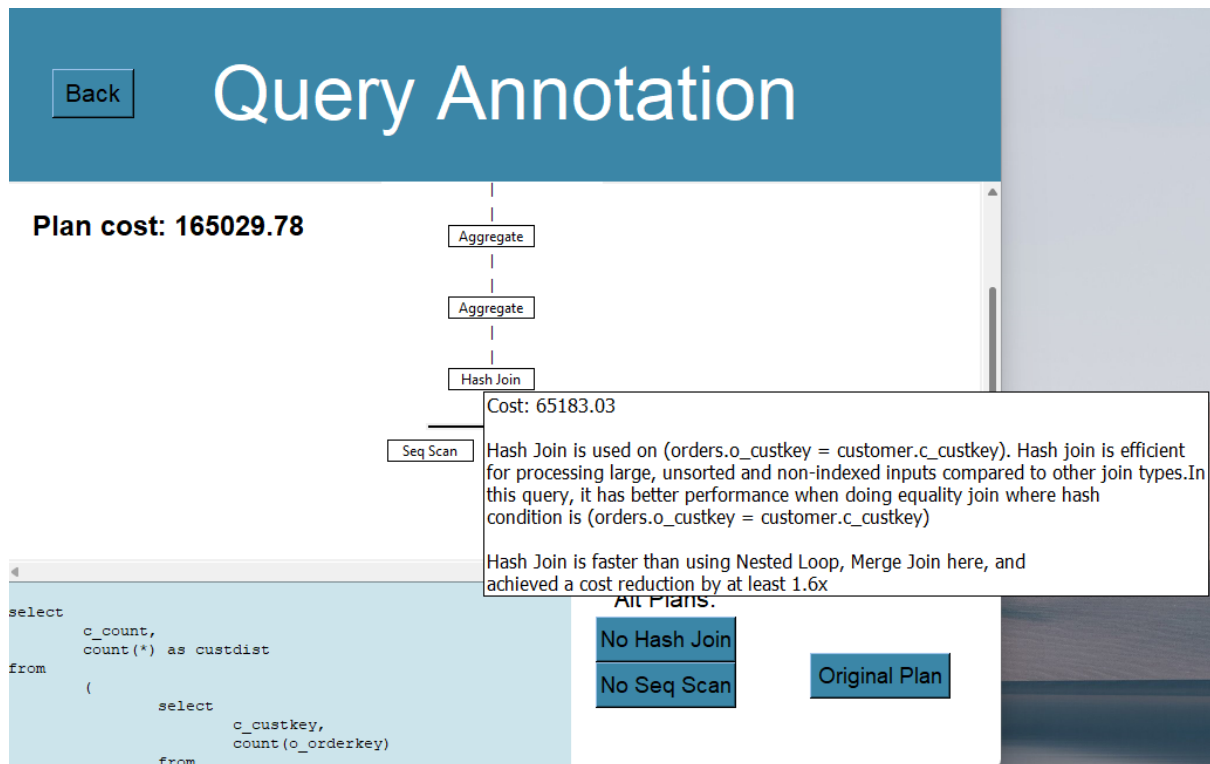


Figure 3.1.4: Query Annotation

Figure 3.1.4 shows how the query annotation is viewed on the Output page. We implemented the annotation display using a tooltip, whereby the annotation information of an operator will be visible when the cursor hovers over that specific operator. Additional information included in the tooltip is the cost, as well as the cost reduction calculated concerning other less efficient operators that can be used.

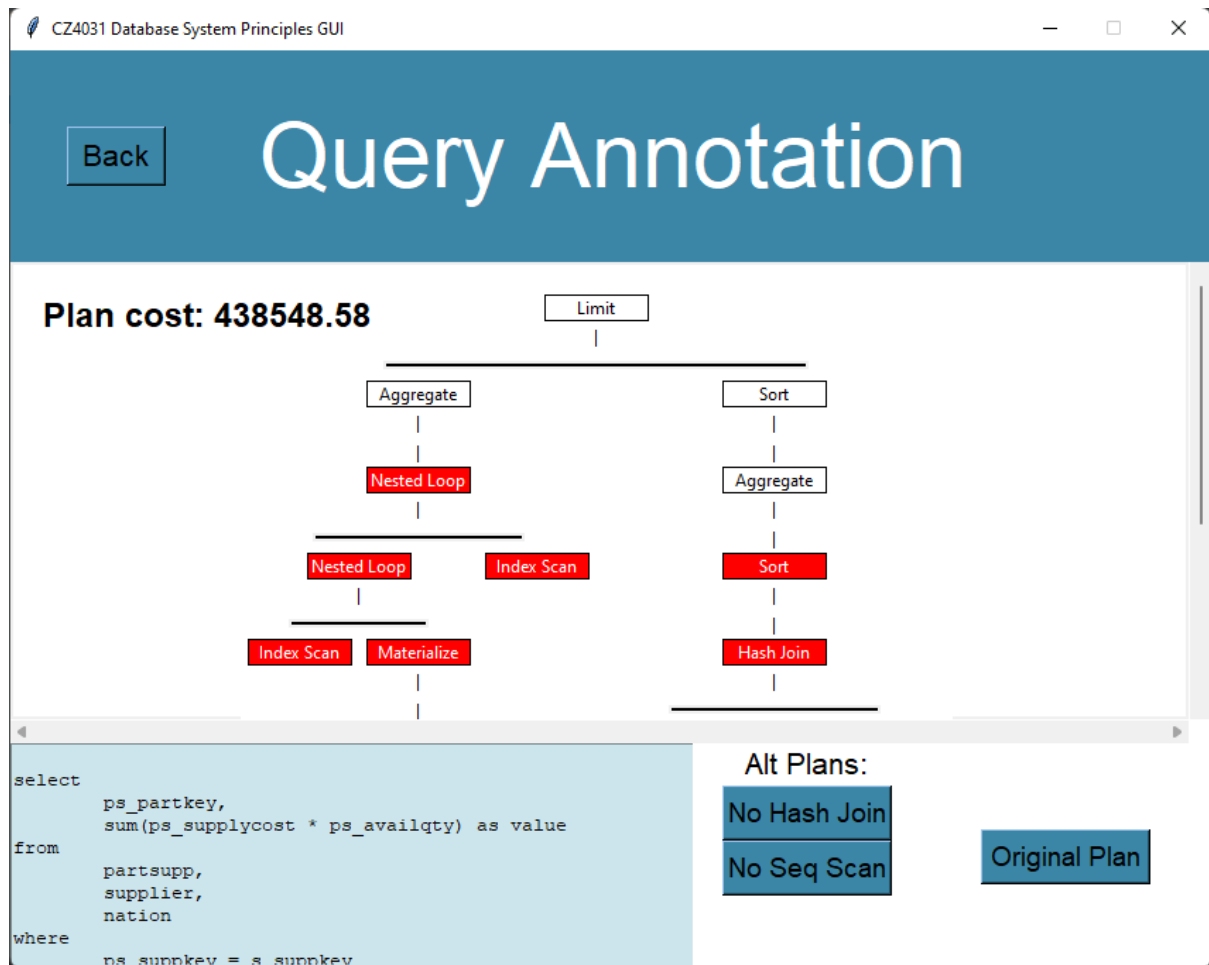


Figure 3.1.5: Viewing alternative plans

When clicking on any of the alternative plan buttons, the tree view will be reloaded with the tree of the AQP. Nodes in red highlights the structural difference from the original plan. When mouse over each node, no annotation will be displayed except the cost.

4. Dataset & Database Schemas

In this project, TPC-H dataset and schemas are used to populate the PostgreSQL database. TPC-H is a decision-making benchmark. It comprises a set of business-oriented ad hoc queries and concurrent data changes. The queries and data used to populate the database were chosen to be of broad industry relevance. This database is used to test our software.

TPC-H relations	Description	Cardinality(no.of rows approx)
REGION	Contains the continents that nations belong in.	5
NATION	Contains a list of all supported nations.	25
SUPPLIER	Contains details about the supplier of parts.	10000
CUSTOMER	Contains a list of all customers, including their names, address and other information.	150000
PART	Contains parts sold and their respective supplier and price.	200000
PARTSUPP	Contains information from the supplier about the part, including inventory and price.	800000
ORDERS	Contains details of the customer's orders, order status, and price of the order	1500000
LINEITEM	Contains all the items within an order, including the shipping dates and prices.	6001215

5. Conclusion

PostgreSQL has around 50 possible operators when generating a QEP. In this project we only annotated the most commonly used ones, such as Hash Join and Sequential Scan. That being said, our program utilised various object oriented programming principles to allow the addition of more operators easily.

For AQP generation, we limit 1 operator at a time, i.e turning off one type of operators at a time. The project can be extended by generating AQPs by turn off multiple combinations of operators to study the effect of such operation.