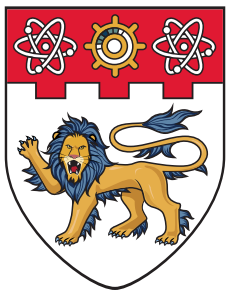


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC3020 Database Systems Principles

Project 2 Report

Name	Email	Matric Number
Cui Nan	C220133@e.ntu.edu.sg	U2221495L
Pu Fanyi	FPU001@e.ntu.edu.sg	U2220175K
Shan Yi	SH0005YI@e.ntu.edu.sg	U2222846C
Zhang Kaichen	ZHAN0564@e.ntu.edu.sg	U2123722J
Tian Yidong	YTIAN006@e.ntu.edu.sg	U2220492B

College of Computing and Data Science
Nanyang Technological University, Singapore

2024/2025 Semester 1

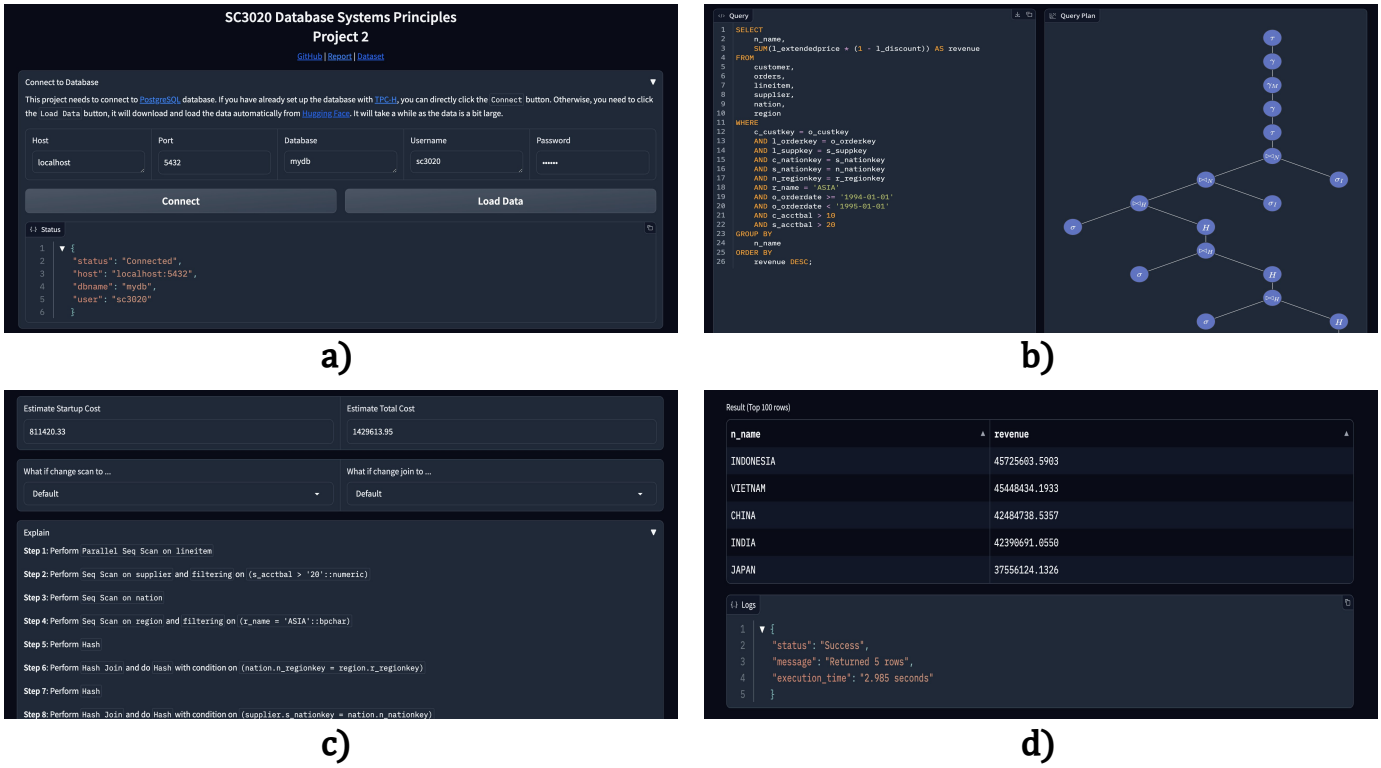


Figure 1: An overview of our designed UI. **a)** We provide an interface to allow you to connect to the SQL server running in backend with your own config. By clicking Load Data, we will fetch the data we use and create data in your selected db. **b)** We allow you to write query and visualize an interactive query plan. **c)** We estimated startup cost and total cost with explanation. You can change the config with the dropdown. **d)** We also allow you to execute the SQL query and log the result.

1 Introduction

In real-world applications, writing SQL to retrieve information from a DBMS has become a routine task. However, since I/O costs significantly impact query performance, it is often necessary to optimize the query plan based on specific requirements. In PostgreSQL [6], the query execution plan (QEP) is generated from a large number of alternative query plans (AQPs). Therefore, it is essential to provide the following capabilities:

- Retrieve and visualize the QEP for a given SQL query
- Support "what-if" queries on the QEP
- Retrieve the estimated cost of the AQPs

To achieve these objectives, we developed a user interface (UI) that allows users to interactively modify their QEPs based on their needs. The functionalities of our application are demonstrated in Fig. 1. The application enables users to connect to an existing database they have created and creates tables from the prepared data fetched from the Internet. Users can then write SQL queries and visualize the corresponding query plans. Additionally, they can view the estimated costs, modify the types of scan or join methods, and optimize the plan based on their requirements. Once satisfied with the cost, users can execute the commands, log the results, and evaluate the performance.

The remainder of this report is structured as follows:

- An overview of the pipeline and the structure of our application
- A detailed explanation of the algorithms
- Case studies demonstrating the application

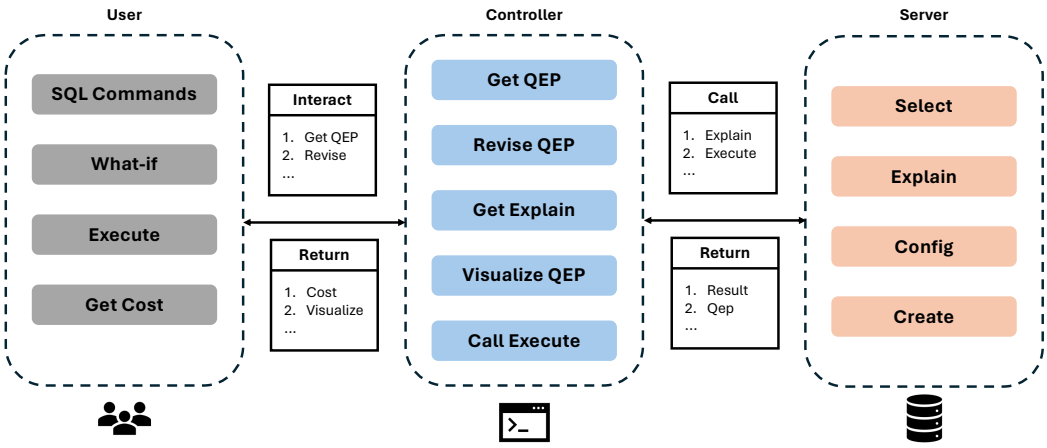


Figure 2: A overview of our pipeline. Where user can perform different operations and receive results from the controller and the server side. The user can freely decide when to execute the commands

2 Overview

In this section, we will give a brief introduction on how our application functions.

2.1 Pipeline Overview

Our application mainly consist of three key components: **1)** A user-interface to allow user to interact with. **2)** A controller component that post the commands to the server and get the result. **3)** A database server that actually execute the commands and store the data.

Our application follows a modular pipeline that integrates a backend for query processing and a frontend for user interaction. The pipeline consists of several stages:

Database Connection Users first connect to the PostgreSQL database through the GUI by providing connection details. If necessary, they can load a prepared dataset from the Internet and create the corresponding table in the database. This operation is performed only once and stores the *tpch* dataset on the server.

Query Plan Retrieval Users can input SQL queries or select examples through the GUI. Once a valid command is entered in the code panel, the QEP visualization is displayed. The execution and retrieval process algorithm is implemented efficiently, providing real-time details on costs, explanations, and tree visualizations. Further details about the algorithm are available in Section 3.

Interactive What-If Analysis Users can modify the query execution plan by changing join types or scan methods through the GUI. A list of options is provided, and changes are reflected in real time in the cost estimation and tree visualization. Planner settings are dynamically adjusted using SQL commands (e.g., `SET enable_*`) based on the query planning configuration. Details of the algorithm are provided in Section 3.4.

Visualization and Explanation After retrieving the QEP or the revised AQP from the server, the system parses it into a tree where each node represents an operation and its condition. This parsed tree is used to calculate the coordinates of the visualized QEP tree. We also provide natural language explanations derived from the tree using a post-order traversal. Each step is parsed according to a set of predefined rules and formats, then presented to the user.

Comparison and Results Once the user is satisfied with their query plan, they can execute it to view the actual query results and query log. The system displays the query execution time, enabling users to validate performance.

2.2 Application Structure

The application is organized into the following key components:

Frontend Gradio [1] is used as our frontend framework. The GUI accepts user inputs (connection details, SQL queries, and what-if modifications) and displays query plans, costs results and logs to the user. It also provides interactive dropdown menus for what-if analysis.

Backend We adopted FastAPI [7] as the framework to provide the backend server logic. The backend contacts the database in SQL commands, with Psycopg 3.

Database System PostgreSQL [6] is selected as the Database system to manage our database. The database is established from TPC-H dataset [5].* To make it easier to reproduce our experiments on different devices, we have uploaded the data to Hugging Face Datasets [4], which could be accessed by this link.

2.3 Key Interaction Flow

Database Connection User connects to the database via GUI → FastAPI establishes the connection → PostgreSQL confirms the connection → Download the dataset from Hugging Face → Load data to the database.

Query Execution User inputs query → Gradio sends request to FastAPI → FastAPI receives the query and send to PostgreSQL → PostgreSQL generates QEP and computes results and costs, then returns them to the FrontEnd.

What-If Analysis User modifies join/scan types on the FrontEnd → Gradio sends modification request → FastAPI adjusts planner settings and send to PostgreSQL → PostgreSQL generates AQP and estimated costs.

Result Display Results, costs, explains and logs are visualized in the GUI.

3 Implementation Details

3.1 Parsing Query Plans and Tree Genreating

A QEP is generated by PostgreSQL with the SQL statement:

```
1 EXPLAIN <QUERY>;
```

A parser is created to parse the output of the generated QEP. We assume $\textsc{Parse}(p_i)$ function can extract the information of the plan and convert it to a node, $\textsc{Depth}(p_i)$ function can retrieve the depth of the plan. $\textsc{Depth}(p_0) = 0$.

The whole process generation QEP tree is DFS with backtracking, described in Algorithm 1.

The tree structure is explained in Section 3.2 and 3.3.

3.2 ExecutionTreeNode Class

The ExecutionTreeNode class represents a single node in the execution tree. Each node corresponds to a query operation (e.g., Hash Join, Seq Scan) and includes metadata like costs and conditions associated with that query operation.

Table 1 lists the key attributes and methods for ExecutionTreeNode.

*Version 3.0.1, retrieved from this link, compiled with `Visual Studio 2022 version 17.10`.

Algorithm 1 GENERATEEXECUTIONTREE(p)

Require: A list of strings representing query plans p from PostgreSQL
Ensure: An EXECUTIONTREE representing the QEP

$\mathcal{T} \leftarrow$ Empty Tree
$o \leftarrow \text{PARSE}(p_0)$ ▷ o is the current node
$\text{ROOT}(\mathcal{T}) \leftarrow o$ ▷ Set o to be the root of $\mathcal{T}$
$d \leftarrow 0$ ▷ Currently depth is 0
for $i \in 1 \ldots \text{LEN}(p) - 1$ **do** ▷ Enumerate remaining plans
    $o' \leftarrow \text{PARSE}(p_i)$ ▷ Current node
    $d' \leftarrow \text{DEPTH}(p_i)$ ▷ Current depth
    while $d' \leq d$ **do** ▷ Backtrack to Find Parent of o
        $o \leftarrow \text{PARENT}(o)$
        $d \leftarrow d - 1$
    end while
    $\text{PARENT}(o') \leftarrow o$ ▷ Set the parent of o' to o
end for
return $\mathcal{T}$

Attributes / Methods	Description
Operation	The query operation.
Conditions	Filters or keys associated with the operation.
Startup Cost	Startup cost for the operation.
Total Cost	Total cost for the operation.
Children	Child nodes representing sub-operations.
Set Operation	Parses the operation and perform estimation.
Natural Language	Converts the operations into a human-readable format.
Explain	Returns the detailed estimation to the operation.

Table 1: Key Attributes and Methods for EXECUTIONTREENODE

3.3 EXECUTIONTREE Class

The EXECUTIONTREE class organizes query operations into a hierarchical structure, allowing traversal and cost analysis. Table 2 lists key attributes and methods of EXECUTIONTREE.

Attributes / Methods	Description
Root	The root node of the tree.
Traversal	Return nodes based on the order of operations.
Get Cost	Aggregates the startup and total costs of all nodes.

Table 2: Key Attributes and Methods for EXECUTIONTREE

The TRAVERSAL method uses post-order traversal methods to return nodes. $\text{TRAVERSAL}(o, \ell)$ means the current node is o, and the answer list is ℓ. shown in Algorithm 2. The total cost and startup cost inside a tree is calculated by summing all the total cost and startup cost for each node in it.

3.4 Interactive What-If Analysis

We explored generating alternative query plans by applying user-defined modifications to join and scan methods. PREPAREJOINCOMMAND and PREPARESCANCOMMAND generate `SET` commands to control PostgreSQL planner configurations. For instance, when a user selects `HASH_JOIN`:

```
SET enable_hashjoin = ON;
SET enable_mergejoin = OFF;
SET enable_nestloop = OFF;
```

Algorithm 2 TRAVERSAL(o, ℓ)

for $s \in$ SON(o) **do**
  TRAVERSAL(s, ℓ)
end for
APPEND(ℓ, o)

This ensures that PostgreSQL uses only the specified join type when generating the QEP. Similarly, we can disables other scan methods and enables only index-based scanning by producing:

```
SET enable_bitmapscan = OFF;
SET enable_indexscan = OFF;
SET enable_indexonlyscan = ON;
SET enable_seqscan = OFF;
```

3.5 QEP Visualizer

QEP Visualizer converts a EXECUTIONTREE to a plot. Firstly, We consider the tree as a planar graph and calculate the coordinates of each node using igraph [2]. The specific calculation method is mentioned in [8]. Then, Based on these coordinates, we perform scaling transformations to adapt them to the screen. Using Plotly [3], we plot the nodes and edges in a Cartesian coordinate system. Finally, we remove the coordinate axes to generate an image of the tree.

To display information for each node concisely, we use symbols to represent each node, with the specific mappings listed in Table 3. Detailed information about that node is displayed when the mouse hovers over a specific node.

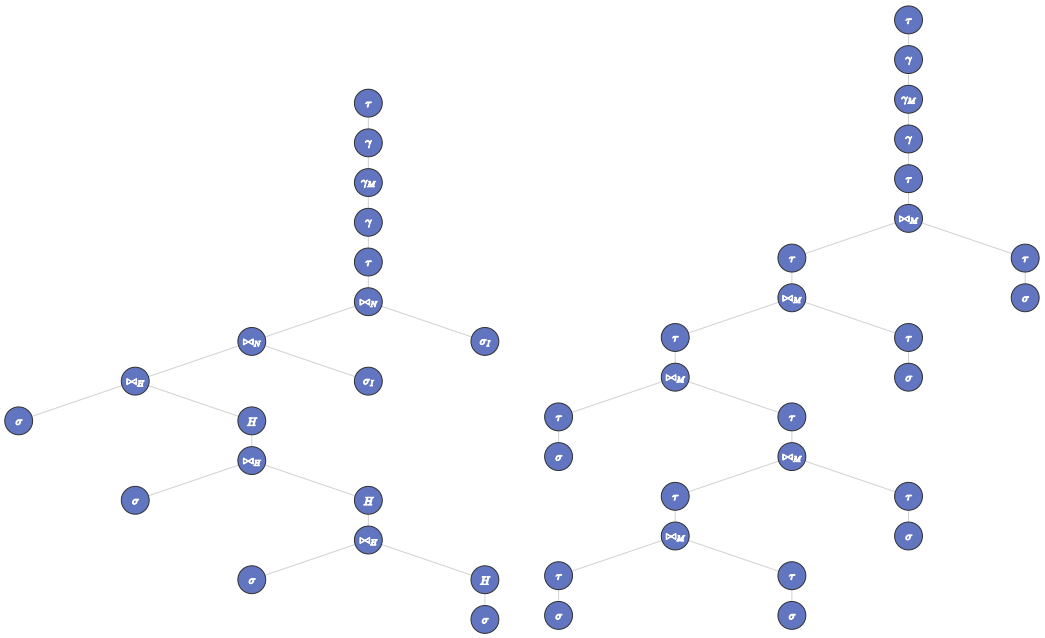
Operation	Symbol	Description
Aggregate	γ	Performs aggregation operations like `SUM`, `AVG`, etc.
Hash Join	$\bowtie_H$	Joins two tables using a hash-based method.
Merge Join	$\bowtie_M$	Joins two sorted tables using a merge-based approach.
Seq Scan	σ	Sequentially scans all rows in a table.
Index Scan	σ_I	Scans rows using an index for faster lookup.
Bitmap Heap Scan	σ_B	Uses a bitmap index for efficient range scans.
Sort	τ	Sorts the rows based on specified columns.
Hash	H	Creates a hash table for efficient data access.
Gather Merge	γ_M	Merges results from multiple parallel workers.
Materialize	M	Stores intermediate results in memory for reuse.
Append	$\cup$	Combines rows from multiple sources into one output.
Unique	δ	Removes duplicate rows.
Group	γ	Groups rows by specific column(s) for aggregation.
Window	ω	Computes window functions like `RANK`, `ROW_NUMBER`, etc.
Limit	L	Restricts the output to a specified number of rows.
Unknown Operation	o	Represents an operation that is not recognized.

Table 3: Operations and their corresponding symbols.

4 Case Study

In this section, we present case studies demonstrating the use of our project. One complex example is highlighted to showcase the correctness and effectiveness of our application. We present more examples that can also be executed directly using our application. In application, you can also hover the mouse on each node for more infomation.

```sql
SELECT
    n_name,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND l_suppkey = s_suppkey
    AND c_nationkey = s_nationkey
    AND s_nationkey = n_nationkey
    AND n_regionkey = r_regionkey
    AND r_name = 'ASIA'
    AND o_orderdate >= '1994-01-01'
    AND o_orderdate < '1995-01-01'
    AND c_acctbal > 10
    AND s_acctbal > 20
GROUP BY
    n_name
ORDER BY
    revenue DESC;
```



(a) The visualization of the example query plan using the default settings. The database can freely choose to use which scan type or join type.

(b) The visualization of the example query plan by setting the scan method to Sequential Scan and join method to Merge Join.

Figure 3: Comparison of the example query plan visualizations.

QEP and AQP comparisons The original Query Execution Plan (QEP) for the above case study example uses scanning method of Index Scan and Sequential Scan to perform selecting; it also

	Startup Cost	Total Cost
Original	815194.52	1434838.82
Revised	6592161.29	6813958.6

Table 4: The estimated cost for different QEP

performs join using Nested Loop Join and Hash Join; the QEP is shown in Fig. 3a. Then the user tests the changes that can be caused by modifying scan method to Sequential Scan and join method to Nested Loop Join, which generates an AQP; the AQP is shown in Fig. 3b. As observed from the above graphs for query plans, the AQP is different from the initial QEP in tree structure and operator sequences. The revised QEP is forced to use sequential scan and merge join for each of the node. We show their estimation cost in Section 4.

5 Conclusion and Limitation

Our software effectively utilizes an interface to allow users to interactively invoke the PostgreSQL database system for query execution and cost evaluation, as well as to visualize the effect of changing the join or the scanning methods on query execution plan and cost of plans.

By leveraging structured tree representations and intuitive visualizations of plan in tree structure view, it helps the users to understand the complex task of query generation and optimization. Make the understanding process less difficult through visual demonstration of query execution process.

Limitations Our application leverages PostgreSQL's configuration options for query plans to enforce the use of specific scan or join methods when generating the QEP. However, this approach cannot precisely modify individual nodes within the QEP, as it uniformly applies the specified method to all join or scan nodes. To fine-tune the processing logic for each node, extensions such as *pg_hint_plan* can be utilized.

6 Use of AI tools

We used ChatGPT to help us refine the paragraph, and picked the grammar mistakes in the report. All the report draft and graphs were written and designed by us.

Part of the hardcode logics, such as if-else statement for the node symbol, which require repeated work are being done by the code pilot. We coded all of our core components by ourselves.

References

[1] Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. Gradio: Hassle-free sharing and testing of ml models in the wild. *arXiv preprint arXiv:1906.02569*, 2019.

[2] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[3] Plotly Technologies Inc. Collaborative data science, 2015.

[4] Quentin Lhoest, Albert Villanova Del Moral, Yacine Jernite, Abhishek Thakur, Patrick Von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, et al. Datasets: A community library for natural language processing. *arXiv preprint arXiv:2109.02846*, 2021.

[5] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[6] PostgreSQL Global Development Group. Postgresql source code repository. https://github.com/postgres/postgres. Accessed: 2024-11-17.

[7] Sebastián Ramírez and the FastAPI Contributors. Fastapi: A modern, fast, web framework for python, 2018. Accessed: 2024-11-17.

[8] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on software Engineering*, (2):223–228, 1981.