Project Report

Data Storage Paradigms, IV1351

Yuhui Gan

4th Jan 2025

Declaration:

By submitting this assignment, it is hereby declared that all group members listed above have contributed to the solution. It is also declared that all project members fully understand all parts of the final solution and can explain it upon request.

It is furthermore declared that the solution below is a contribution by the project members only, and specifically that no part of the solution has been copied from any other source (except for lecture slides at the course IV1351), no part of the solution has been provided by someone not listed as a project member above, and no part of the solution has been generated by a system.

1 Introduction

In this task, I transitioned from designing the database to working directly with SQL to create queries and views for the Soundgood Music School database. The primary goal of this task was to develop Online Analytical Processing (OLAP) queries to support the school's business analysis and reporting needs. There are four different exercises for me to get a first taste of creating OLAP queries. I worked independently on this task and the queries are stored in a Git repository, along with scripts to create and populate the database, enabling seamless testing and reproducibility.

2 Literature Study

For this project, I used PostgreSQL, a relational database management system, to create, manage, and query data. I read online documentation and watched the SQL lecture recommended on Canvas. Relational databases store data in structured tables with defined relationships, allowing efficient storage, retrieval, and manipulation. The database design follows a schema with primary keys for unique identification and foreign keys to establish relationships between tables. This structure ensures data integrity and supports complex operations.

Relational databases adhere to principles such as normalization, which eliminates redundancy and ensures consistency by organizing data into smaller, interrelated tables. Additionally, PostgreSQL ensures ACID (Atomicity, Consistency, Isolation, Durability) compliance, making it reliable for data transactions.

3 Method

For Task 3, I used PostgreSQL as the database management system (DBMS) to create, manage, and query relational databases. To develop and execute SQL queries, I used pgAdmin, a graphical user interface for PostgreSQL, and occasionally the psql command-line tool for direct interaction with the database.

To verify that my SQL queries worked as intended, I often ran each query in the query tool to check for syntax correctness and ensure expected outputs. Then I used generated sample datasets to validate query results against known inputs and outputs. And the graphs would tell me if I am on the right track with the exercises or not.

4 Result

In this exercise 1, I created a database schema for tracking lessons and instructors. The schema included two tables: instructors and lessons, with a foreign key relationship between them. To verify the schema, I executed queries to retrieve all instructors and lessons, ensuring the relationships were correctly established.

The goal here was to show how many lessons happen every month, and separate them into three types: individual, group, and ensemble. I created a table called schedule where each row represents a lesson. The table has columns for lesson_type, genre, day, and the number of seats.

To get the result, I used TO_CHAR and TO_DATE to format the month number into a month name. The CASE WHEN inside the SUM() function helps separate the lesson types.: In this exercise 2, I aggregated the lesson data to count the number of lessons each instructor conducted. The query used COUNT and GROUP BY to summarize lesson counts per instructor. This query needed to show how many students have no siblings, one sibling, or two or more siblings. To do this, I made a table called studentsibling with two columns: student_id and contact_person_name. Students with the same contact person are siblings.

I counted how many students share the same contact person and then used a CASE block to sort them into groups:

```
SELECT instructor_id, COUNT(*) AS no_of_lessons
FROM lessons
GROUP BY instructor_id;
```

Exercise 3 focused on retrieving instructors with more than a specified number of lessons in the current month. I used EXTRACT for filtering dates and HAVING to enforce the lesson threshold.

month	total	individual	group_lessons	ensemble
Apr	2	0	2	0
Aug	2	0	1	1
Dec	1	0	1	0
Feb	2	0	2	0
Jan	2	1	0	1
Jul	2	2	0	0
Jun	2	0	2	0
Mar	2	1	0	1
May	2	2	0	0
Nov	1	1	0	0
Oct	1	0	0	1
Sep	1	1	0	0

(12 rows)

Figure 1: Figure 1. Results for exercise 1

no_of_siblings	no_of_students
0	4
1	3
2	6

(3 rows)

Figure 2: Figure 2. Results for exercise 2

instructor_id	first_name	last_name	no_of_lessons
2	Pomona	Sprout	4
1	Albus	Dumbledore	3
3	Gilderoy	Lockhart	2
4	Minerva	McGonagall	2

(4 rows)

Figure 3: Figure 3. Results for exercise 3

In exercise 4 I worked on retrieving ensembles scheduled for the next week and categorizing available seats. The query used a CASE statement for seat categorization and filtered results based on the date range.

Link to Github: https://github.com/GerminalG/IV1351.git

```
 Day | Genre  | No of Free Seats
-----+--------+------------------
 Fri | Punk   | 1 or 2 Seats
 Fri | Rock   | No Seats
 Sun | Rock   | Many Seats
 Thu | Gospel | No Seats
(4 rows)
```

Figure 4: Figure 4. Results for exercise 4

5 Discussion

I believe that materialized views could improve performance in scenarios where the same aggregated data, such as lesson counts per instructor, is queried repeatedly. By pre-computing and storing these results, materialized views reduce run-time overhead for frequent queries. However, in this project, queries were lightweight enough that materialized views were not necessary to achieve acceptable performance.

No correlated sub-queries were used in any of the exercises. Instead, I relied on efficient techniques like WITH clauses (Common Table Expressions) and joins to structure queries. This avoided the performance pitfalls of correlated sub-queries, which can be slow due to their repeated execution for each row in the outer query. All queries were kept concise, avoiding unnecessary complexity such as redundant UNION clauses or deeply nested sub-queries. For instance, in Exercise 4, the CASE statement was used effectively to categorize seat availability without introducing additional sub-queries.

To assess the performance of the queries, I used the EXPLAIN ANALYZE command to analyze execution plans. In Exercise 3, most of the query's execution time was spent on the JOIN operation between the aggregated lesson counts and the instructors' table. This behavior was expected and reasonable given the purpose of the query. Indexing the instructor ID column in both tables could further improve performance, especially for larger datasets. Additionally, filtering lessons for the current month involved a sequential scan of the lessons table, which could also benefit from a partial index on the date column.