Project Report - Task 3: SQL

Data Storage Paradigms, IV1351

December 1, 2024

Project members:
Nils Ekenberg, nilseke@kth.se
Samuel Engbom, sengbom@kth.se
Peter Li, pli2@kth.se

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, we create Online Analytical Processing (OLAP), queries and views based on the Soundgood Music School database that was created in earlier tasks. Furthermore, we analyze the efficiency of one of the queries with *Explain Analyze*. In total, we want to make sure four different queries generates correct outputs:

- *Show the* ***number of lessons*** *given* ***per month*** *during* ***a specified year***
- *Show how many* ***students*** *there are with* ***no sibling****, with* ***one sibling****, with* ***two siblings****, etc.*
- *List* ***ids*** *and* ***names*** *of* ***all instructors*** *who has given* ***more than a specific number*** *of* ***lessons*** *during* ***the current month***
- *List* ***all ensembles*** *held during* ***the next week****.*

2 Literature Study

In order to solve this task, in addition to other literature material, we also gained knowledge the given lectures, lecture slides, and the book reading references that was instructed on Canvas.

3 Method

First we start by populating all necessary data such that the queries would output reasonable results. The queries were hand-crafted and tested using both the CLI and pgAdmin 4. To verify that the queries met the expected behavior, we began by looking at our relatively small dataset and manually determined the expected outcome, which the queries should reflect. We also tested the queries by excluding rows and giving incomplete data to test for gaps. In the case where we utilized subqueries, such as in the Sibling Count query, we could verify the logic by testing them separately.

4 Result

In this section, the tables are only for illustration, due to the limit of width of the document, column names are shortened on purpose.

Link to the GitHub repository containing the .sql files `https://github.com/yyIntsuki/data-storage-paradigms/tree/main/task-3`.

The YearlyLessonSummary view provides monthly summaries of lessons for a given year. First we extract the year and month from each lesson which allows us to group lessons by year and month, we then count the number of lessons for each specific lesson type.

The CASE statement checks if there is a match in the type of lesson and if true, adds 1 to the count, while also keeping track of the total. See table 1.

Month	Total	Individual	Group	Ensemble
1	1	1	0	0
2	1	0	1	0
3	1	0	0	1
...	...	...	...	...

Table 1: Output from the YearlyLessonSummary query

In the MonthlyInstructorWorkload view we summarize the number of lessons each instructor has taught, then in the accompanying query we can utilize this view to retrieve instructors' workload for the current month and filter for instructors who taught more than a given number of lessons, specified by comparing to a set value. We join tables lesson and instructor to see who taught what lesson and then instructor and their person details. We group lessons by instructor ID and name, ensuring one row per instructor per month and then year and month to separate lessons by calendar period. See table 2.

Instructor Id	First Name	Last Name	No of Lessons
2	Anna	Johansson	5
3	Lina	Bergström	5
4	Oskar	Lindgren	5
1	Erik	Svensson	3
5	Emma	Karlsson	3
...	...	...	...

Table 2: Output from the MonthlyInstructorWorkload query

The SiblingCount query calculates two things, how many siblings each student has, and the amount of students that belong to each sibling count category.
To achieve this functionality, it uses a combination of subquery, a LEFT JOIN and a GROUP BY. The subquery creates a list of all sibling relationships present in the sibling table, making sure to treat them as bidirectional. We then join the student table to ensure those without siblings are included, for each student id in the student table the LEFT JOIN checks if there is a match in the combined subquery otherwise the count will be 0. The GROUP BY steps aggregate the results. See table 3.

No of Siblings	Students
0	11
1	10
2	6

Table 3: Output from the SiblingCount query

NextWeekEnsembles is a view that provides a list of the ensembles scheduled for the next 7 days. We convert start time of the lesson to a more readable day of the week, then we ensure that each row in the result is tied to a specific genre.

The logic for calculating the number of free seats available is done by taking the number of students already registered and subtracting them from the specified max allowed number of students. The WHERE clause ensures the query adapts to current date, the CASE statement simplifies availability in to more descriptive categories. See table 4.

Day	Genre	No of Free Seats
Monday	Jazz	Many Seats
Wednesday	Blues	Many Seats
Wednesday	Pop	No Seats
Friday	Pop	Many Seats
Friday	Classical	Many Seats
Saturday	Jazz	1-2 Seats

Table 4: Output from the NextWeekEnsembles query

5 Discussion

When it comes to the choice of VIEW and MATERIALIZED VIEW, we decided not to implement the materialized views because of their stale nature and the fact that our data is expected to change very frequently. We did consider the query for the ensembles to be MATERIALIZED because of the fact that this can potentially be run anywhere from once a week to hundreds of times per day. This is entirely dependent on how many students there are in the Soundgood musical school and the browsing pattern of the students. This could also be implemented in the website in such a way that it is displayed on the front page, in which case the load on the server will be considerably lowered when it loads in every time they open the Soundgood website. This comes at a cost of write performance and storage size, however the write is to a single row while the read is done to the entire table as well as the query complexity.

For this task there were a lot of things that had to be changed in our database. However that was because the previous design had flaws that made it unable to function. After fixing the database the only thing that was changed to allow for easier queries would be the separation of first_name and last_name from the previous column full_name. This does not impact the complexity nor quality of the database and there is a possibility for a negligible impact on size since both new columns are varchar, it is however a change that allows for easier querying in general.

None of the queries contain correlated subqueries which is good since we avoid the associated performance penalty, instead we rely on joins, aggregations and simple subqueries which are executed once, such as those present in the SiblingCount view. These subqueries do not depend on the outer query and is evaluated once and joined with the student table.

In the view SiblingCount we use UNION ALL to combine students from the sibling table to treat the relationships as bidirectional, in this case the UNION is warranted since we need to ensure bidirectionality; however, one could imagine that the siblings table ensured this functionality itself which would then render the UNION ALL unnecessary.

By running EXPLAIN ANALYZE on the SiblingCount view we can analyze the query plan to gain some insight in to how PostgreSQL executes the query. The Seq Scan on student table processes very quickly, the Append step combines two Seq Scan operations on the siblings table and are also very fast. We can see that the DBMS spends the most time in the HashAggregate and Hash Right Join steps. HashAggregate aggregates the sibling counts and Hash Right Join, joins the student table with the sibling relationships. Aggregation and joining are inherently more complex operations and given such a small dataset which PostgreSQL handles efficiently, this is a reasonable outcome.