SQL

Data Storage Paradigms, IV1351

Group 50

2024-12-02

Project members:
[Allan Al Saleh, Allan2@kth.se]
[Mostafa Faik, mfaik@kth.se]
[Derfesh Mariush, derfesh@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.

Introduction

In this assignment we implemented queries for the database, which was made in the previous assignment. The purpose of the queries is to visualize important information such as the amount of: lessons per month, students who have siblings, lessons teachers have in a certain month, and seats available for ensembles.

Literature Study

To gain the foundational knowledge for this task, we relied on these resources:

- Lecture video(SQL): The video taught us all the basics of SQL, everything from creating tables, implementing columns with constraints, modifying information in a database, creating queries which retrieve and structure information from other tables and lastly implementing views.
- Textbooks: We referred to the book *Fundamentals of Database Systems* (7th edition by Elmasri and Navathe), specifically chapters 6 and 7, to complement the information on SQL which we learned from the previously mentioned lecture.
- Tips and Tricks: We referred to the PDF mostly to bolster our knowledge on SQL basics, creating queries and subqueries for a physical/logical model, to step by step visualize certain aspects in a database and lastly, when to utilize materialized and non-materialized views.

Method

Diagram Editor and Tools Used:

pgAdmin was used for database management, writing SQL queries. Data generation for the inserted script data, was facilitated by the online tool Generate data.

Query Verification:

To ensure the accuracy and reliability of the SQL queries, a systematic verification process was worked as intended. Each query was initially executed on the data which we provided. The data was built to verify that the database constraints and triggers work correctly. This meant that we tried to go over the database constraints with limits out of boundaries and insert values in wrong columns. To further test the database we compared the results from the database with the expected results based on manual calculations benchmarks. This approach ensures that the queries achieved as the results we intended, and matched the analytical objectives of the assignment.

Result

This section presents the queries developed to meet the requirements of the assignment, including detailed explanations for each query. All queries are stored in a Git repository, which also contains the scripts for creating the database and inserting test data.

The solution is stored in the following Git repository:
https://gits-15.sys.kth.se/derfesh/IV1351-HT24-Data-Storage-Paradigms-Soundgood

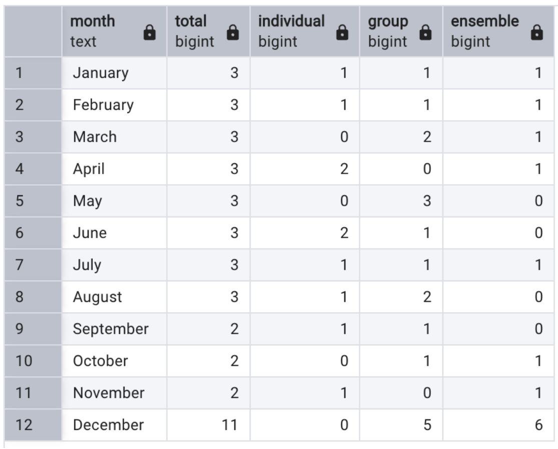
Repository Structure:

The Git repository contains the following files:

- database.sql: Script that creates the database structure, including tables, relationships, and constraints.
- data.sql: Script that inserts sample data into the database for testing purposes.
- queries.sql: Script containing all the queries/views for each task.
- historicalDatabase.sql: script containing the historical database table.
- historicalQuery.sql: the query/SQL statements for copying data from the normalized database to the historical database.

Query 1: Number of Lessons Given Per Month

This query retrieves the total number of lessons, as well as the number of individual lessons, group lessons, and ensemble lessons, for each month of a specified year. The query is designed to run multiple times per week and aggregates lessons by type and month.



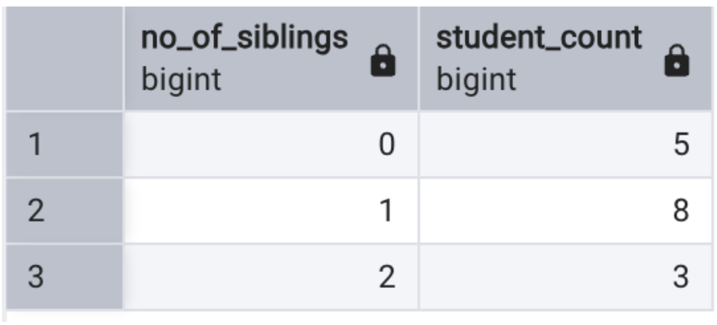
	month text	total bigint	individual bigint	group bigint	ensemble bigint
1	January	3	1	1	1
2	February	3	1	1	1
3	March	3	0	2	1
4	April	3	2	0	1
5	May	3	0	3	0
6	June	3	2	1	0
7	July	3	1	1	1
8	August	3	1	2	0
9	September	2	1	1	0
10	October	2	0	1	1
11	November	2	1	0	1
12	December	11	0	5	6

Figure 1: Query 1

This query extracts the month from each lesson's start date, then aggregates the total lessons, and counts for each lesson type. The results are grouped by month and sorted to show the lessons in chronological order. Currently, the query shows the output for the year 2024. This can simply be changed into the desired year by changing the input year in the *WHERE* condition in the query (e.g., 2023, 2022 and etc). We can also have the query output the current year from the *CURRENT_DATE* by changing the input year in the *WHERE* condition in the query to *EXTRACT(YEAR FROM CURRENT_DATE).*

Query 2: Number of Students with No, One, and Two Siblings

This query counts how many students have zero, one, and two siblings, and outputs the number of students for each category. The database is structured in such a way that siblings are tracked through a separate table, and this query uses a LEFT JOIN to handle the sibling relationship.



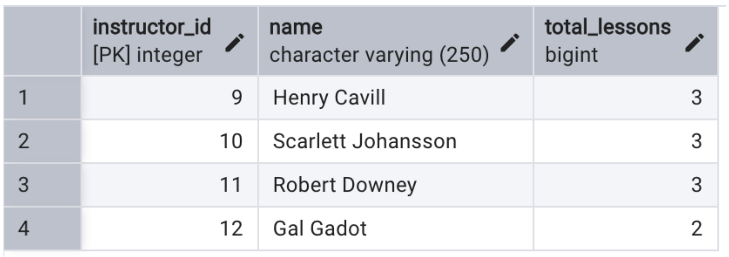
	no_of_siblings bigint	student_count bigint
1	0	5
2	1	8
3	2	3

Figure 2: Query 2

This query uses a subquery to first calculate the number of siblings for each student. The outer query then counts how many students fall into each category of the sibling count (*0, 1, or 2*). This query currently outputs only students that have *0, 1 or 2* siblings. We can also change the query to output the number of students who have more than *2* siblings by simply removing the WHERE condition in the query.

Query 3: Instructors who have given more than a specific number of lessons this month

This query identifies instructors who have given more than a specified number of lessons during the current month. It aggregates lessons regardless of the type and sort the result by the total number of lessons given.



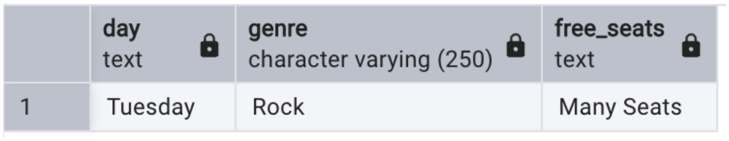
	instructor_id [PK] integer	name character varying (250)	total_lessons bigint
1	9	Henry Cavill	3
2	10	Scarlett Johansson	3
3	11	Robert Downey	3
4	12	Gal Gadot	2

Figure 3: Query 3

The query counts the lessons each instructor has given during the current month. Filter by current year and month and only shows instructors who have taught at least one lesson. The results are sorted by the total number of lessons. Currently, the query shows the output of the year and month of the *CURRENT_DATE* (i.e., 2024 and December). We can make the query output the desired month and year by simply changing the inputs in the *WHERE* condition from *EXTRACT(YEAR FROM CURRENT_DATE)* to the desired year and from *EXTRACT(MONTH FROM CURRENT_DATE)* to the desired month.

Query 4: Ensembles Held During the Next Week

This query retrieves ensemble lessons scheduled for the next week, sorted by music genre and weekday. It also categorizes the seat availability using a CASE statement.



	day text	genre character varying (250)	free_seats text
1	Tuesday	Rock	Many Seats

Figure 4: Query 4

This query checks the number of seats available for ensemble lessons in the next week. It uses a CASE statement to determine seat availability (no seats, 1-2 seats, or many seats). This query can show different outputs if we run it on a different day (i.e., it can show different outputs if we run it tomorrow or the day after), because it shows the number of seats in the ensemble lesson for the next week (7 days from now / 7 days from when we run it).

Discussion

This section evaluates the queries, the use of Views and Materialized Views and discusses the de-normalization strategy applied in the historical database and the advantages and disadvantages of using de-normalization for this particular case. The de-normalized database tracks information such as the type of lesson, genre (for ensembles), instrument (for individual/group lessons), lesson price, and student information, including their name and email.

Use of Views and Materialized Views:

Views and materialized views are useful tools to simplify queries, enhance performance, and provide easier access to aggregated or frequently used data. However, not all of the queries in this assignment would benefit significantly from using them.

An important fact to remember is that Soundgoods database is not very complex, most of the queries could run either as non-materialized views or SQL queries and the system would work fine. Materialized views are mostly for queries that are extremely complicated and require pre-processing.

- Query 1 (Total Lessons per Month): This query is used occasionally every week, which means that it is not an extremely rare query which only needs to be used once, so it does validate it being a view. However, the query is not used for a current date, but is for a specified year, which means that it is better of as an SQL query. As making it a view makes it less flexible.
- Query 2 (Number of Students with Siblings): The same goes for this query, it is used a few times a week. This table won't be updated frequently, which would potentially be an excuse to implement a materialized view for it. However, the query isn't complex enough to make it worth it. It is better for it to be a non-materialized view because we can always reuse the code.
- Query 3 (Instructors with Many Lessons): This query is used daily and requires to update constantly, as a lot of changes can happen on a weekly bases. The code can also be reused, which would make it a good case to implement it as a non-materialized view.
- Query 4 (Ensembles in the Next Week): We used a view for this query as it is updated constantly, by the logic that the amount of free seats of a lesson might see fast changes. The code for this table can be reused, which means that implementing it as a non-materialized view is efficient.

Changes to Database Design:

To simplify these queries, the database schema was designed with efficiency in mind, but no major changes were made specifically to optimize the queries. The lesson and lesson_price tables are designed to handle lesson types, and the sibling table captures the relationship between students and their siblings. The only changes that we made, were based on the feedback that we received for seminar 2.

There were no significant design changes made that would negatively affect the schema for the sake of simplifying queries. The relationships between tables (e.g., lesson to lesson_price, student to sibling) were appropriately normalized, which helps keep the database efficient and avoids data redundancy.

Correlated Subqueries:

None of the queries in this report use correlated subqueries. Correlated subqueries, where a subquery depends on values from the outer query, can be slow because they are executed for each row in the outer query. Fortunately, the queries here were structured to avoid such subqueries, using JOIN operations and GROUP BY clauses to efficiently aggregate and filter the necessary data. For example, Query 2 calculates the number of siblings for each student without needing a correlated subquery.

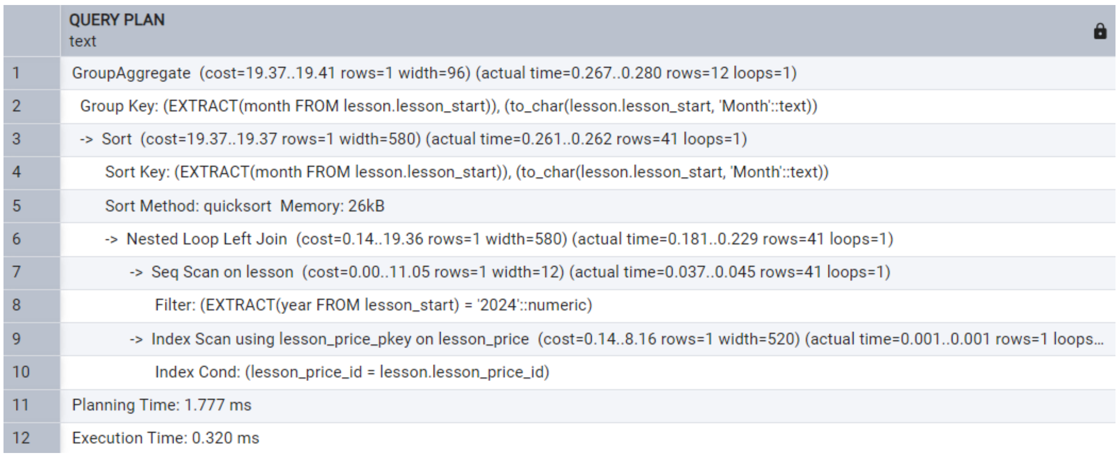
Query Complexity and Use of UNION:

None of the queries use the UNION clause because it was not necessary for the task at hand. The queries use JOIN operations to combine data from different tables, which is more efficient than using UNION.

The queries do not contain unnecessarily long or complicated subqueries, and each one performs a well-defined aggregation or filtering operation. There is no redundancy in the queries, and each is optimized for readability and performance.

Query Plan Analysis with EXPLAIN:

For this sequence, the system decides on this plan:



```
QUERY PLAN
text
GroupAggregate  (cost=19.37..19.41 rows=1 width=96) (actual time=0.267..0.280 rows=12 loops=1)
  Group Key: (EXTRACT(month FROM lesson.lesson_start)), (to_char(lesson.lesson_start, 'Month'::text))
  -> Sort  (cost=19.37..19.37 rows=1 width=580) (actual time=0.261..0.262 rows=41 loops=1)
        Sort Key: (EXTRACT(month FROM lesson.lesson_start)), (to_char(lesson.lesson_start, 'Month'::text))
        Sort Method: quicksort  Memory: 26kB
        -> Nested Loop Left Join  (cost=0.14..19.36 rows=1 width=580) (actual time=0.181..0.229 rows=41 loops=1)
              -> Seq Scan on lesson  (cost=0.00..11.05 rows=1 width=12) (actual time=0.037..0.045 rows=41 loops=1)
                    Filter: (EXTRACT(year FROM lesson_start) = '2024'::numeric)
              -> Index Scan using lesson_price_pkey on lesson_price  (cost=0.14..8.16 rows=1 width=520) (actual time=0.001..0.001 rows=1 loops...
                    Index Cond: (lesson_price_id = lesson.lesson_price_id)
Planning Time: 1.777 ms
Execution Time: 0.320 ms
```

Figure 5: Query Plan

We analyzed this query plan from the bottom node level. The system starts by sequentially scanning the “lesson” table for lessons in the year 2024, from there we can see that all lessons in the database are in the year 2024, as no rows were removed. After that, the system performs an index scan on the “lesson price” table, with the index being the ”lesson price id” and tries to do a left join with the modified “lesson” table. This means that we pair the rows of “lesson” and “lesson price” when they share the same lesson price id, which results in lesson rows with added information on which type of lesson it is, the difficulty of the class, the price etc. The system then quick sorts the data based on the month numerically, meaning that it modifies the list so that lessons that starts in the month of January, then February and etc. We then group the formed rows which are of the same month which leads us to the creation of the final table.

Denormalization and Database Design:

Denormalization is the process of combining or copying data from multiple tables into one, often at the expense of data redundancy. In this assignment, we chose to store all the lesson data, including the price, type, genre, instrument, and student details, directly in the historical_lessons table. This table is intended to track historical data for marketing purposes, where each row represents a specific lesson taken by a student, along with its associated price and other relevant details.

Advantages of Denormalization:

- Improved Query Performance for Reporting: By denormalizing the historical lesson data, we eliminate the need for complex joins across multiple tables when generating reports. Queries for marketing purposes, such as generating lists of lessons taken by a student and their corresponding prices, can be executed more efficiently, as all the required information is stored in one table. This reduces the time complexity of reports.
- Simplified Query Logic: Denormalized tables make querying for historical data simpler and easier to understand. In a normalized database, generating reports could involve complex JOIN operations across several tables. With the denormalized historical table, all the necessary data is available in a single table, which reduces the need for complex query logic and makes it easier to retrieve the data.
- Avoiding the Complexity of Time-Tracking for Prices: By storing the price directly in the historical_lessons table instead of linking it to the lesson_price table, we avoid having to track the time intervals for prices. This makes it easier to access the data, as we no longer need to manage and calculate the effective price during specific periods.

Disadvantages of Denormalization:

- Data Redundancy: Denormalization introduces data redundancy, as the price information, student details, and other fields are repeated across multiple rows. For instance, if multiple students take the same lesson, the lesson price and other lesson-related information will be duplicated for each student. This redundancy increases the size of the database, potentially leading to higher storage requirements.

- Increased Risk of Data Anomalies: Denormalization can lead to data anomalies, especially if data consistency is not carefully managed. For example, if the price for a specific lesson type changes, the updated price must be propagated across all historical records of that lesson type. This introduces the risk of data inconsistencies if the update is not properly applied to all relevant rows.
- Complexity in Data Updates: If there are changes in the lesson data (e.g., price adjustments or changes in the type of lessons), these changes must be manually propagated to the historical database. Unlike in a normalized database, where changes are automatically reflected in related tables, the denormalized data requires careful maintenance and manual updates, increasing the potential for errors.