Project Report Task 3

Data Storage Paradigms, IV1351

2024-12-02

Project members:
Jonathan Värild, varild@kth.se
Oscar Caddeo, ocaddeo@kth.se
Elias Holm, eliholm@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

2 Literature Study

Our primary source of information for this assignment was the PostgreSQL documentation available at `https://www.postgresql.org/docs/current/`. This provided valuable information and even examples on how to write different queries. The chapters 6 and 7 in the book also provided valuable information on how to structure the actual queries and how to work with the data. For example, **7.1 More Complex SQL Retrieval Queries** mentions uses of the different functions such as **SUM**, how to work with **GROUP BY**, joining tables, etc. They also provide very good examples with images and diagrams which further help to visualize the concepts.

3 Method

This project task was to create an OLAP consisting of writing 4 different SQL queries to the previously created Database, as well as create a denormalized historical database and a query to fill it with data. The DBMS used for this task was **postgreSQL** and the queries where developed in the IDE visual studio code, with git used for version control.

The method used for verifying that the queries work as intended was a structured testing method. Each query was run against the database with some prepared mock data, and the results were reviewed to make sure they met all the specified requirements from the query descriptions as well as the Soundgood music school. The verification for this included checking and changing the output formatting, handling edge cases and confirming all logic. The expected output was also tested, initially with a small dataset when creating the queries and lastly with bigger datasets to make sure all edge cases and could be detected and tested against the expected output.

When making the denormalized database we started by looking at the information that the database should contain. We quickly realized that there wasn't much data that needed to be stored and that we probably would be enough with creating a single table for said data. Minimizing the number of tables also makes the database more denormalized and ensures that you don't have to include as many **JOIN**:s later when making queries. We tried to consider what the database would be used for to determine how much we should denormalize it whilst still keeping it usable in a real-world scenario. For example, we decided to keep the student first and last name in separate columns. Next, we made a diagram in Astah using our knowledge from previous tasks. We included notes and constraints to our data to make it as clear as possible. Implementing our database in SQL could also easily be made from our knowledge from previous tasks.

4 Result

The GitHub repository with Astah files, SQL files, etc., can be found at `https://github.com/Cadde0/IV1351-Project`.

4.1 Query 1: Show the number of lessons given per month during a specified year

See code: `https://github.com/Cadde0/IV1351-Project/blob/da393a327bdfa9095d946e80d1bcf94df4ccb866/queries.sql#L1`.

The first query we created was for retrieving information about how many activities there are of each type, each month.

The query works using a **SELECT** operation on the **activity** table which is **LEFT JOIN**:ed with **lesson_individual**, **lesson_group**, and **ensamble** where their **activity_id** matches. Basically, what we do is that we take all the activities and get the full information of each activity from the respective table depending on the activity type. **LEFT JOIN** allow us to join the **activity** table with all the tables at once (**lesson_individual**, **lesson_group**, and **ensamble**), but each join will only affect

matching rows for each activity. If we were to use normal **JOIN**, we would lose a lot of rows where matches weren't found since each activity only is linked with 1 of the 3 activity tables.

The data is filtered using a **WHERE** clause where we extract the year from the **start_time** column of each activity, and compare it to a number representing the year we want to look at. This ensures only activities from that year are included, as per the assignment requirements. If you want to look at other years, you need to modify the number value for that comparison.

The **GROUP BY** clause ensures that all the activities are grouped by their month. The reason why we group the rows by both the text based month and the numerical month is since we use the numerical month for sorting, and therefore have to include it.

Finally, the **ORDER BY** clause where we sort the rows based on the numerical value of each month.

	month text	total bigint	individual bigint	group bigint	ensamble bigint
1	Jan	13	4	8	1
2	Feb	12	6	5	1
3	Mar	10	2	7	1
4	Apr	15	7	5	3
5	May	13	4	6	3
6	Jun	14	1	9	4
7	Jul	12	6	4	2
8	Aug	13	6	5	2
9	Sep	16	9	5	2
10	Oct	11	5	3	3
11	Nov	10	3	4	3
12	Dec	24	6	6	12

Figure 1: The output from performing query 1 on our database with the test data provided in the GitHub repository.

4.2 Query 2: Show how many students there are with no sibling, with one sibling, with two siblings, etc.

See code: `https://github.com/Cadde0/IV1351-Project/blob/da393a327bdfa9095d946e80d1bcf94df4ccb866/queries.sql#L25`.

The second query will list how many students that different number of siblings registered in the school database.

First, we want to get all sibling relations that there is in our database. All siblings are registered in a table called **sibling** which consists of two columns corresponding to two student school IDs that link the two students together. To ensure that there are no

duplicate rows for two students, the two rows are a composite primary key and a trigger enforces that the two student school IDs are ordered with the smallest school ID in the first column and the largest in the second column.

Therefore, we make two **SELECT** operations. First we make one where we select all **student_school_id_first** and count the number of **student_school_id_second**. Since we count the number of **student_school_id_second**, we of course have to use **GROUP BY** on **student_school_id_first**. After that we do exactly the same **SELECT** operation, but we swap **student_school_id_first** and **student_school_id_second**. To merge these operations, we use **UNION ALL** between them. These operations are done in a Common Table Expression (CTE) called **sibling_count** to increase readability. Next, we have another CTE called **total_sibling_count** where we sum up the number of siblings that we counted respectively for each column in the previous CTE.

Finally, we display the information using a **SELECT** operation. Since we also want to know how many students that there are without siblings, we make the **SELECT** operation on the **student** table. We **LEFT JOIN** all students with our CTE **total_sibling_count** on their student IDs which ensures that we only join students that have siblings, but still keep the rows of student that doesn't have siblings. We select the **no_of_siblings_sum** from each student as **"No of Siblings"** and use the function **COALESCE** to ensure that **NULL** values are replaced by **0**, since this are students without siblings that weren't matched up with our **LEFT JOIN** clause. Next, we select **COUNT** with all rows as **"No of Students"** to count the number of students for each number of siblings.

Finally, we use a **GROUP BY** clause to group everything by the number of siblings. We also use a **ORDER BY** clause to order the rows by the number of siblings. The result can be seen in Figure 2 below.

	No of Siblings numeric	No of Students bigint
1	0	17
2	1	10
3	2	3
4	3	4
5	4	20
6	5	6
7	9	10

Figure 2: The output from performing query 2 on our database with the test data provided in the GitHub repository.

4.3 Query 3: List ids and names of all instructors who has given more than a specific number of lessons during the current month.

See code: `https://github.com/Cadde0/IV1351-Project/blob/da393a327bdfa9095d946e80d1bcf94df4ccb866/queries.sql#L61`.

In this query we will select the instructor ID, first name, last name, and the number of lessons that each instructor has given during the current month. In our example, we limited the number of lessons to **0** to show more data, but this number can obviously easily be changed to only show instructors that has given a certain number of lessons.

The query starts with a CTE called **lessons_this_month** which contains a **SELECT** operation on the **activity** table. We select the **instructor_school_id** column and use **COUNT** as **total_lessons** on all rows to count the number of lessons that each instructor is assigned on. Since we use **COUNT** we also have to use a **GROUP BY** clause on the **instructor_school_id** column. To enforce that we only count lessons that were given during the last month we have a **WHERE** clause where we check that the month and year of the **start_time** column on each activity is the same as our current month and year.

Now we only have the instructor ID and the number of lessons that each instructor ID has given. To connect this information with a real person, we finally make a new **SELECT** operation on **lessons_this_month** and join this data with the **person** table on the instructor ID. This allows us to finally select the first and last name of the instructor. It's here that we have a **WHERE** clause where we can decide the lower limit of lessons to include the instructor. Finally, we have a **ORDER BY** clause that orders everything by **"No of Lessons"**.

	Instructor Id integer	First Name character varying (50)	Last Name character varying (50)	No of Lessons bigint
1	55	Charlotta	Nord	11
2	2	Linn	Nord	3
3	59	Lena	Henriksson	3
4	34	Evert	Linder	2
5	3	Carina	Pettersson	2
6	30	Stig	Andersson	1
7	54	Christian	Oskarsson	1
8	15	Magdalena	Karlsson	1

Figure 3: The output from performing query 3 on our database with the test data provided in the GitHub repository.

4.4 Query 4: List all ensembles held during the next week

See code: `https://github.com/Cadde0/IV1351-Project/blob/da393a327bdfa9095d946e80d1bcf94df4ccb866/queries.sql#L84`.

This query will give us a schedule-like output of all ensembles next week and also include the number of free seats for each ensemble.

We start by defining a CTE called **ensambles_next_week** where we have a **SELECT** operation from activity. This table is in turned **JOIN**:ed with the **ensamble** table where their activity IDs match to only include activities that are ensembles. This also gives us access to all the information about each ensemble. We use a **WHERE** clause where we check so the **start_time** of the ensemble is within the first and last day of next week.

Next, we create another CTE called **bookings_per_ensamble** where we count the number of bookings for each ensemble. We do this using a **SELECT** operation on **ensambles_next_week** which is joined using **LEFT JOIN** on the **booking** table where their activity ID matches. We select the activity ID and use **COUNT** to count the number of rows for this activity.

Finally, we make an **SELECT** operation on **ensambles_next_week** and **LEFT JOIN** this with **bookings_per_ensamble** where their activity IDs match. We select the weekday using the **TO_CHAR** function with the **start_time** column, genre, and use a **CASE** clause to show different texts for the **"No of Free Seats"** column based on the number of seats left. We use the **COALESCE** function to ensure that we have a **0** to compare with for **NULL** values where there are no bookings. Finally, we use a **ORDER BY** clause to order everything first by **Day** and then by **Genre**.

	Day text	Genre character varying (99)	No of Free Seats text
1	Mon	Folkmusik	Many seats
2	Sat	Klassisk	No seats
3	Thu	Jazz	1 or 2 seats

Figure 4: The output from performing query 4 on our database with the test data provided in the GitHub repository.

4.5 Query 5: Query for selecting and inserting historical data in denormalized database

See code: `https://github.com/Cadde0/IV1351-Project/blob/da393a327bdfa9095d946e80d1bcf94df4ccb866/queries.sql#L116` and `https://github.com/Cadde0/IV1351-Project/blob/main/CreateHistorical.sql`.

This last query is part of the higher grade task, where we are assigned to select all activities that people have made bookings for into a denormalized database. We have also created the historical database diagram which can be seen in **Figure 5** below. To make it easier to move the data to the **activity_history** table with a single query we implemented it into the same database. This might be bad practice because you don't necessarily want to fill up your main production database with historical data that is based on data within it. Instead, it would be a great idea to offload this data to a

database that only keeps historical data. However, moving data between databases like this isn't introduced in this course.

The **activity_history** table consists of a composite primary key of **student_school_id** and **activity_id** which ensures that there can only be one row for each activity and student. This effectively eliminates the possibility of duplicate rows for the same booking. Next, we have the **activity_type** column which is implemented as a enum defined in the connected note. Next, we have the **activity_price** which is the final price for the activity, based on the hourly price of the activity and its length. The **activity_price** also have a check to ensure that it isn't negative, since this would be unlogical. After that we have the **instrument** and **genre** columns which are allowed to be **NULL**, but uses checks to ensure that they only are **NULL** when allowed depending on the **activity_type**. Lastly, we have the student's first name, last name, email and the date of the activity. You could denormalize the database even more by combining the first and last name into a single column, but we chose to separate them because we believe that there might be practical reasons to still be able to easily differentiate this later if you need to use this historical data.

Looking back at the query, we start with a CTE called **personal_emails** where we have a **SELECT** operation which selects **DISTINCT school_id**, **email** and a **school_id**. We have a **WHERE** clause which ensures that we only select rows that have personal contact details and an email address. Basically, what we do is that we select all personal email addresses registered, but only one per **school_id** since our denormalized database only allows one email, per booking.

Next, we have a CTE called **history_data** which basically joins and merges all data that we need to later make the **INSERT** operation into the denormalized **activity_history** table. It starts with a **SELECT** operation on the **booking** table, since it is the bookings we are interested in. We join this table with **person** to get access to the person name, etc., behind each booking. We use **LEFT JOIN** on **personal_emails** to join the email addresses of all students that have a personal email registered. The **activity** table is joined to get access to activity data such as the start time, end time, and activity type. The **pricing** table is joined, so we can retrieve pricing data that can be calculated using the start and end time in the **activity** table. We use **LEFT JOIN** on **lesson_individual**, **lesson_group**, and **ensamble** to join the bookings only with its matching activity. Finally, we make a join with **instrument_type** to get the name of the instrument for all the lessons.

All this data is selected pretty easily except from **activity_length_hours** that is calculated from the start and end time of the activity, and **activity_type** that is made into an **activity_type_enum** as needed by the **activity_history** table,

A final **INSERT** operation then uses a **SELECT** operation to get the data from **history_data** and insert it into **activity_history**. The **activity_price** is calculated by multiplying the columns **activity_length_hours** and **hourly_price**.

Running the query multiple times will not be a problem due to the final clause **ON CONFLICT** which ignore rows that already have been inserted. However, you would probably also want to include a **WHERE** clause that ignores activities before a certain date. Nothing about this is however mentioned in the assignment, so this is just some-

thing we considered ourselves. The query will run just fine for moving data without **ON CONFLICT** as long as you don't run it on the same data set twice.

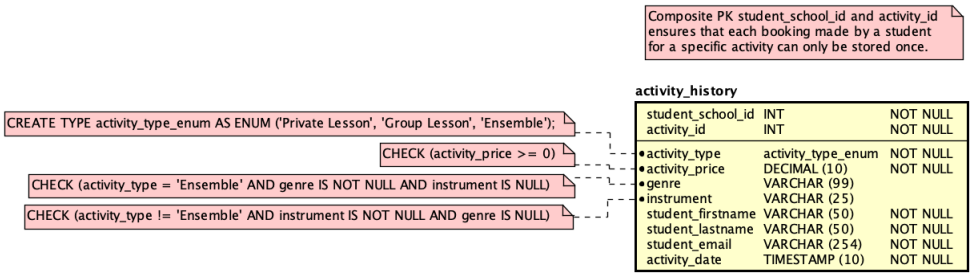

Figure 5: Diagram of our denormalized database for storing historical data.

5 Discussion

5.1 Analyzing Queries

No part of the database was changed or modified from the previous task to simplify these specific queries since the database design should not be changed for the lesser only to simplify a single query. Below follows a short discussion regarding each query:

Query 1 uses grouping and aggregation and is not computationally expensive enough, which can be seen using postgreSQL EXPLAIN, or by simply logically looking at the query, to justify a materialized view, however if the query is done frequently enough, a materialized view could help performance. It also uses no correlated subqueries.

Query 2 uses "sibling_count" and "total_sibling_count" CTEs to break down the query int smaller logical steps for clarity. There exists an argument to have a materialized view since sibling counts are most likely almost static since siblings don't change unless a sibling quits the school or a new one joins. It uses grouping and summing instead of any correlated subqueries. This query uses a UNION ALL, but this is required since a student can appear in both the first and the second column of the table and not using a UNION ALL would not simplify the query but instead lead to incomplete results and losing data requested.

Query 3 uses "lessons_this_month" to calculate total lessons this specific month and then uses that to filter based on the number of lessons. A materialized view would not be appropriate for this query since the number of lesson can change, especially if you are querying for the current month where scheduled lessons are customed to change. In this query all aggregations are handled using GROUP BY and JOIN and therefore avoids any correlated subqueries.

Query 4 has two CTEs, one for next weeks ensembles and one for how many bookings/students booked each ensemble has to improve clarity and make the query more logical. The students booked CTE is used to calculate the number of available places from the maximum number of places at the lesson. A materialized view would not fit

here since it queries the ensembles for next week, meaning the lessons and especially the number of students booked on a lesson are most likely to change. To avoid any correlated subqueries that would reduce performance this query uses a CASE statement and joins.

5.2 Advantages and disadvantages of using denormalization

Denormalized databases are often used in implementations where you want to favor read speeds. Its main advantages include better performance for reading data due to fewer uses of joins and potentially better indexing. It also makes queries a lot easier to understand since you won't have to work with a lot of tables like you do in a normalized database.

One of the obvious disadvantages is bigger database size since you store a lot of the same data multiple times. It might also be hard to maintain a denormalized database if you need to modify the same data at multiple locations. This can become time-consuming for its maintainers and makes it prone to errors. It will also be a lot more performance heavy if you for example need to update the same data at multiple locations. If you are working with huge amounts of data, you also pose the risk of meeting the limitations of your database quicker. This might for example arise as performance degradation, etc.

6 Comments About the Course