Project Report - Task 3

Data Storage Paradigms, IV1351

Reza Hosseini rezahos@kth.se

December 2022

1 Introduction

The goal in this task is to create OLAP, Online Analytical Processing, queries and views. Such queries serve to analyze the business and to create reports. We will also analyze the efficiency of the queries using EXPLAIN ANALYZE. Even though that's not standard SQL, it's available in both PostgreSQL and MySQL. One thing to consider is to have enough and sufficient data to test all the queries and make sure that we get reasonable resposnses otherwise we need to insert more data.

2 Literature Study

To understand the task we first, watched the course lecture about SQL given by Paris Carbone: `IV1351 - 2021 - 3.SQL`
We did also take a quick look at the chapters 6 and 7 from the main book of the course
For the higher grade part, we studied about `denormalisation` on Google and how you can implement it.
The document `tips-and-tricks-task3.pdf` was also continuously used throughout the task.

3 Method

In this task we used PostgreSQL as our Database Management System. We used Visual Studio Code and the PostgreSQL extension to develop the queries and we used pgAdmin4 to run/test different queries. Both VSC (with PostgreSQL extension) and pgAdmind 4 was used to verify the results of the queries.
To test and verify the queries, we ran them all in pgAdmin 4 and made sure we got results back without any errors.

Verifying the results was done in three steps:
We ran other simpler but multiple queries to see if we got the similar results. For instance to verify whether we get all ensembles that are supposed to be held next week, we ran a simple query that shows us all the ensembles ordered by time then we manually looked for the ones that have its time during next week and compared them with the results of our first query.
Another step to verify the results was to just look at the inserted data and see if we have all the information covered. For instance to see if we have all sibling data, we just looked at the inserted data and verified the returned results.
Last step was to simply count number of results or rows we get back after running the queries and see if it's expected or not. For instance running the query that shows number of lessons during every month should have 12 rows, representing each month, and the "total" column should be the sum of three other columns "individual lesson" + "group lesson" + "ensemble"

4 Result

The file with OLAP queries as well as the SQL scripts for both generating the database and inserting data into it, is found in the following GitHub repository: `https://github.com/Rezaavoor/Soundgood-Music/tree/master/Task3`

Number of lessons per month

Below is shown the result of the query:

month	individual	group	ensemble	total
January	2	1	0	3
February	5	0	1	6
March	1	1	0	2
April	2	0	0	2
May	0	0	1	1
June	1	1	1	3
July	1	0	0	1
August	1	0	0	1
September	0	0	0	0
October	2	0	1	3
November	1	0	1	2
December	0	1	0	1

Table 1: Lessons given per month in 2023, seperated by lesson type

The query is made of three views and one temporary table.
Each view consists of the month and number of lessons during that month for each of the lessons types: individual, group or ensembles. We use the command `extract` to get the month out of the column "time". it returns a value between 1 to 12 which says the month of the year. To make sure we do not mix months of different years, we also check that we are in the year 2023 by extracting the year from time: `...where extract( year from el.time) = 2023`.
The temporary table only contains the name of each month. So the ID of each row in the table is the numeric value of that month and the column "month" shows the name of that month. This is more of an optional table to get the actual names of the months. You can skip this table and go only with numeric representation if you wish.
After creating all the views, we can do a `full outer join` between them to make sure we don't lose any data but we do not wanna have `null` values in our table so we use the command `coalesce(number, 0)` instead which replaces the value with 0 if it's null. And finally we add them all together to fill the "total" column

Siblings

Below is shown the result of the query:

num-of-siblings	count
0	36
1	10
2	2
3	2

Table 2: Number of students with different amount of siblings

The result shows that there are 36 students without any siblings, 10 with only one sibling, 2 with two siblings and 2 with three siblings.
The query is a union between two "select" commands:
The first one shows the count of students in the table student-sibling. And the second one just counts the number of students that are not existed in the table student-sibling which are the students who have no sibling, so the column "num-of-sibling" is 0 in this table.

Instructors and taken lessons

Below is shown the result of the query:

id	name	num-of-lessons
1	Cruz	6
5	Wylie	2
8	Megan	3

Table 3: Number of lessons of instructors with more than 1 lesson this month

This query shows, there are 3 instructors this month that have more than 1 lesson.
The query is large `union all` between all the 3 types of lessons based on instructor-id. Then only columns that its time is during this month, is taken and finally all columns are grouped and counted by instructor-id.

Ensembles held during next week

Below is shown the result of the query:

id	genre	weekday	max	min	enrolled	status
8	Gospel Band	5	6	2	5	1-2 seats left
6	Punk Rock	2	15	5	5	more seats left
9	Punk Rock	6	6	2	5	1-2 seats left
7	Rock	4	10	3	10	full booked

Table 4: Ensembles held during next week

The query is a "join" between ensemble and student-ensemble to be able to count the number of enrollments. To get the weekday we use the command `isodow from time`. After the join, we group and count the results by ensemble-id and finally order it by genre and weekday. In order to have the "status" column, we used the command `case...when...end` and compared our "enrolled" values to "max" values.

5 Discussion

- Usage of UNION:
  We used UNION when trying to list all instructors who has given more than a specified number of lessons. In this case we think we had to use UNION in order to get all the lesson because we don't know if there is any other way of counting the number of lessons an instructor is holding. We also used UNION when trying to merge the students who have no siblings with the ones who have. Since the table student-sibling only contains the data of students who have sibling, we think the only solution was to have a UNION and get other students as well.

- Usage of Views:
  We created three views in the query about number of lessons given per month. The reason was that we not only needed the count of every type of lessons for each month, but also needed them to be separated. So we could not just join all the three lesson types and do one big operation with them. Therefore we used views and performed similar operations on every lesson type and then joined them all together. Since each view is a separate column in the final result, we think the usage of views was necessary in this case.

- EXPLAIN ANALYZE: We analyzed the query that shows instructors who have more than one lesson during the current month:

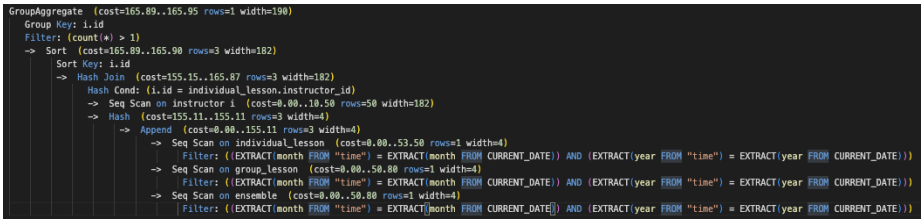

Figure 1: Result of analyzing the query

The results show that:
We do sequential scanning through all the three tables `individual-lesson, group-lesson and ensemble` with the same filter that only extracts the ones that have the current month and the current year as their time. This is the select operation and the where clause that comes after. Estimated start-up cost is 0 for all the there operations but estimated total-cost is 53, 50 and 50 respectively. The number of rows output by these operations is in total 3 (1 for each operation).
Next comes an append operation that basically puts together all the retrieved data from the previous three operations which then follows by a hashed scan which normally is faster.

Another sequential scan on the instructor table is done because we are getting ready to do a join between them. The estimated start-up cost on the hashing takes 155.11 while it is 0 on the instructor scanning part.
Next up is the join operation with the condition of
`i.id = individual-lesson.instructor-id`.
Now that all data is retrieved, we do a sorting based on `i.id` which has 165.89 estimated start-up cost time and 165-90 estimated total-cost.
And finally another filter is done with the condition of `count(*) > 1` and at last we get a group aggregation.

- Historical data and denormalization:

  All the SQL statements for copying data from main DB, creating new tables in the historical DB and inserting data into them is shown in the same GitHub repository mentioned above in file called `historical-db.sql`.

  The task for marketing purposes is that the school wants to be able to see which lessons each student has taken, and at which cost, since they first joined. This means we have to create a historical database where we store data for ever and it needs to be separated from our current database to not make it too large and slow.
  The solution we came up with was to just have two tables in the historical data `student and lesson` tables. The student table looks almost identical to our current student table, however the lesson table is changed. Now we have merged all the three types of lessons into one general and we only keep the data of price, time, attended students and type of lesson which is shared across all the three different types of lessons.
  What we are doing is `denormalization` which may lead to some duplicated data but we want to avoid complexity as much as possible.

  In order to create the database we first need to transfer data from our main DB and that is done by using en extension called `dblink`. With the help of dblink we create view for each necessary table that its data needs to be transferred.
  We later on use those different views to insert appropriate data into our two single tables `student and lesson`.
  One thing to note is that there are many-to-many relationships between students and group lessons and ensembles. Since we want to avoid complexity and need of "join" as much as possible, we decided to create a new row for each student that has taken a group-lesson or an ensemble. This leads to duplication of data but brings us simplicity.

As briefly talked above, there are some pros and cons with denormalizaion when it comes to storing historical data:

- faster reads:
  A denormalaized data base usually gives you faster read queries since you do fewer joins and other complex SQL statements. For instance, we can now list all the individual-lessons and its prices without needing to do any joins since all the data is stored in the "lesson" table. Another example is that by only doing one single join between student table and lesson table, you can have access to all the data in the historical DB such as which lessons a student has taken and at what price.
- simpler queries:
  Since we deal with fewer tables, we have then simpler queries and getting the desired data is more straightforward. It is also less likely to get bugs in queries. For instance in our main database, in order to get the type of a lesson we have to do two joins. First join with lesson-detail and then join with lesson-type but in the denormalized database we do not need any joins at all.
- more duplicated data:
  One downside to denormalization is that your database has to store a lot of duplicated data. If the database is a large one then this may lead to some serious problems as the database keeps growing. In our case this problem shows itself the most when we need to store the many-to-many relationships. If for instance 20 students have attended a group-lesson, that means we have 20 almost identical rows of information in the lesson table in which only the student-id is different.
- expensive with updates and data insertion
  Denormalizaion usually requires many sequential scans and reformation our current tables which can cost a lot. If updating the historical Db in the company happens regularly and often then this problem needs to be considered. In our database, we don't really have a problem with the student table but we have to do multiple sequential scans for every cross-table as well as all three types of lessons in order to insert data. We use also views for every table to transfer data which also costs a lot.