Seminar 3 - SQL

Data Storage Paradigms, IV1351

Kevin O'Flaherty, kevinof@kth.se

January 10, 2023

1 Introduction

Building on the logical/physical model from the previous seminar, for this assignment we were expected to write the script for a set of given queries, simulating the Soundgood School's need for reports and analysis. Since the queries aimed to output data given specific kinds of relations, part of the task required us to fill the database with relevant data.

The method chapter covers the tools used to achieve these tasks, and the query outputs are presented in the Result chapter.

Given that we had the tools at hand, we also were required to use the EXPLAIN ANALYZE operation to evaluate at least one of our OLAP queries or views. This is presented in the Discussion chapter.

While working on this task I compared notes with Patricia Lagerhult. Filling the databases with generated data we discovered some issues with our respective table schemas and so we did a fair deal of preparatory work together. While mostly writing our queries independently, we compared them, swapping solutions with each other.

2 Literature Study

In preparation for this task I watched the course lecture "SQL - The Structured Query Language" and read the seminar specific "Tips and Tricks." I did find the lecture kind of opaque, and found a better understanding of SQL by using the Khan Academy resource Intro to SQL. So armed, the corresponding chapters (6 and 7) of the main textbook Fundamentals of Database Systems was easier to take in.

For solutions to specific use cases I encountered a quick google search often returned me to the official PostgreSQL documentation or the PostgreSQL Tutorial .

To fill the created database with data to query I made extensive use of `https://generatedata.com/`.

Finally, I used the Assessment criteria for Seminar 3 to structure the Discussion chapter of this report.

3 Method

To build and manage the database I continued to use PostgreSQL as for the previous task. To write the script I worked in Jetbrains DataGrip, although to be fair I didn't make full use of it's functionality. Except for a few syntactic mishaps indicated by the IDE I might well have used any text editor for writing the script. Initially I wrote simple queries directly to the DB, to get a hang of the output but soon switched to writing the queries in a file and running it from PostgreSQL, due to the fact that they were becoming more complex.

3.1 Building the database

To test the queries I wrote I needed av variety of data for them to return. Due to dependencies in the database design, this meant filling a few tables not directly involved in the given query. I tried to err on the side of caution and inserted data that might not be used, preferring not to have to find out later that some specific table needed to be queried and have to manually add extra data later (for example foreign keys).

While trying to figure out how to implement the queries I had to return to the logical-physical model created in the previous assignment. In doing so, and discussing the model with my colleague students, it became clear that the solution of inheritance between the lesson table and its children ensemble_lesson, individual_lesson and group_lesson was flawed. This led to changing the ensemble-individual-group triad to a ensemble-instrument diad, letting the CASE-clauses based on the maximum allowed students given in an attribute of lesson decide whether an instrument_lesson was "individual" or not.

3.2 The queries themselves

Writing the queries I tried to work backwards from the intended output given the data requested and the data I'd entered into the database. I usually began with a specific case by manually setting a few select tuples from relevant tables, using more general randomly generated data as confidence grew.

Given the query requirements from the assignment and my databasy design I didn't find a lot cause for views minimizing otherwise duplicated code. I did however make use of them to increase the queries' readability and filtering out certain types of irrelevant data (for example timeslots not connected to lessons) in an early stage of the query.

4 Result

All of the scripts, as well as logical-physical model can be found in the repository at `https://github.com/kaytaffer/soundgoodDB`. To see the outputs in the following chapter run dbbuild.sql, insert.sql and queries.sql in order. The inserted data is however time sensitive, as some of the queries rely on CURRENT_DATE and the inserted data is mostly focused around the beginning of year 2023.

For the first query we were asked to output the amount of lessons given per month during a specified year. My result is visible in figure 4.1.

month	total_lessons	ensembles	individual_lessons	group_lessons
4	1	0	0	1
6	4	1	0	3
7	5	2	2	1
8	1	0	0	1
10	3	2	0	1
11	3	1	0	2
12	2	0	1	1

(7 rader)

Figure 4.1: Output Query 1

The script creates two views where the first selects the filled lesson time slots for the year in question. The second view divides data about the maximum number of students allowed for each instrument_lesson, separating them into individual and group lessons respectively. Finally these views are combined into a query presenting the total lessons, and number of each kind of lesson given per month.

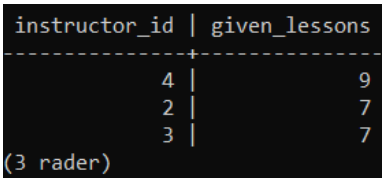
The second query should show how many students had what amount of siblings. The 20 students in my dataset are disributed according to figure 4.2.

siblings	students_with_number_of_siblings
0	10
1	4
2	6

(3 rader)

Figure 4.2: Output Query 2

This script counts the number of students with 0, 1 or 2 (and so on) siblings, using a subquery 'students_sibling_amount' that counts the sibling relations of each student. This is my only case of a subquery that wasn't created in a seperate view. I decided to leave it nested in the FROM-clause because it was relatively simple.



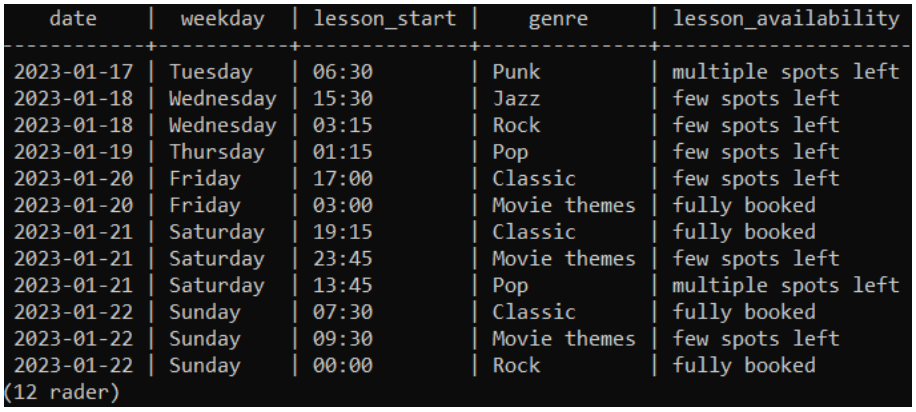
instructor_id	given_lessons
4	9
2	7
3	7

(3 rader)

Figure 4.3: Output Query 3

The third query was about burnout among instructors and counts lessons that have been given by instructors during the current month, presented in descending order by number of lessons, with a specific cutoff point defining which instructors are working to much. In my example I chose the cutoff point at more than 5 lessons, as shown in figure 4.3. As above, a subquery selects the contents of the FROM-clause.

The final query was the most complex, requiring multiple joins. Comparatively, the requested info can be stated simply: Retrieve all ensembles held during the coming week, sorted by genre and weekday. It is somewhat complicated by the free spaces left having to be presented in different way, depending on how many they are. This is solved with a CASE-clause. See figure 4.4 for the output the third week of januari 2023.



date	weekday	lesson_start	genre	lesson_availability
2023-01-17	Tuesday	06:30	Punk	multiple spots left
2023-01-18	Wednesday	15:30	Jazz	few spots left
2023-01-18	Wednesday	03:15	Rock	few spots left
2023-01-19	Thursday	01:15	Pop	few spots left
2023-01-20	Friday	17:00	Classic	few spots left
2023-01-20	Friday	03:00	Movie themes	fully booked
2023-01-21	Saturday	19:15	Classic	fully booked
2023-01-21	Saturday	23:45	Movie themes	few spots left
2023-01-21	Saturday	13:45	Pop	multiple spots left
2023-01-22	Sunday	07:30	Classic	fully booked
2023-01-22	Sunday	09:30	Movie themes	few spots left
2023-01-22	Sunday	00:00	Rock	fully booked

(12 rader)

Figure 4.4: Output Query 4

5 Discussion

As stated in the Result chapter, for the query where the expected output was the number of students with different numbers of siblings enrolled at the school, I used a subquery in a FROM-clause instead of a view. The same was done very similarly in the instructor burnout query. In most of the other similar cases I preferred to write it as a view joined with the main query, but for this case it seemed like a simple enough query to nest within the main one, also given that it isn't a correlated subquery.

For the rest of the views I decided to keep them as not materialized based solely on the repeated calls to "[n]ever optimize for time unless it's proven that a problem exists, and that the optimization solves it" as Leif Lindbäck puts it in the tips and tricks-document for this lecture. I did however consider it for the first query, ordering the given lessons for a specified year. Once a new calendar year starts the data is not expected ever to be overwritten for passed years, and so would be great candidates for a materialized view. The volumes of data today are however miniscule, and don't really merit optimization.

I also reflected over the fact that he "instructor burnout" query would be performed daily with continually updated data and so would be a bad candidate for a materialized view. This is even more true for the final query, intended to be displayed online and to change at any given time a student is signed up to an ensemble lesson. That information would need to be absolutely current to be useful.

I changed the database design in the case of lesson inheritance, which in some ways eased access to specific data (the numbers of students allowed for a given lesson regardless of lesson type) but in others made the queries harder to write. This was specifically the case where I had to derive the data that was asked for, since the lesson type wasn't given in the table name outright. The higher normalization came at a cost of a few extra lines of code.

```
                                   QUERY PLAN
--------------------------------------------------------------------------------------------------------
 Sort  (cost=5.74..5.74 rows=1 width=12) (actual time=2.371..2.374 rows=3 loops=1)
   Sort Key: (count(lesson.instructor_id)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  GroupAggregate  (cost=5.70..5.72 rows=1 width=12) (actual time=2.334..2.349 rows=3 loops=1)
         Group Key: lesson.instructor_id
         Filter: (count(lesson.instructor_id) > 5)
         Rows Removed by Filter: 2
         ->  Sort  (cost=5.70..5.70 rows=1 width=4) (actual time=2.319..2.324 rows=33 loops=1)
               Sort Key: lesson.instructor_id
               Sort Method: quicksort  Memory: 25kB
               ->  Hash Join  (cost=3.41..5.69 rows=1 width=4) (actual time=2.265..2.306 rows=33 loops=1)
                     Hash Cond: (lesson.timeslot_id = timeslot.id)
                     ->  Seq Scan on lesson  (cost=0.00..2.00 rows=100 width=8) (actual time=0.009..0.020 rows=100 loops=1)
                     ->  Hash  (cost=3.40..3.40 rows=1 width=4) (actual time=0.197..0.198 rows=34 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 10kB
                           ->  Seq Scan on timeslot  (cost=0.00..3.40 rows=1 width=4) (actual time=0.115..0.159 rows=34 loops=1)
                                 Filter: (EXTRACT(month FROM lesson_start) = EXTRACT(month FROM CURRENT_DATE))
                                 Rows Removed by Filter: 86
 Planning Time: 0.312 ms
 Execution Time: 2.430 ms
(20 rader)
```

Figure 5.1: EXPLAIN ANALYZE instructor burnout-query

Figure 5.1 shows the output from EXPLAIN ANALYZE run on my implementation of the third query, the one listing instructors working too much. It doesn't have any "humps" and might suffer from it due to the fact that each step relies on previous step. There might not be a lot of concurrent work.

The main work is done below the node indicated by the third indented arrow, where the query joins the two tables *lesson* and *timeslot* to sort out only the relevant data to be able to select it. This is done in a subquery.

```
FROM student_lesson
INNER JOIN ensemble_lesson
    ON student_lesson.instructor_id = ensemble_lesson.instructor_id
    AND student_lesson.timeslot_id = ensemble_lesson.timeslot_id
```

The two arrows in line represent the two operations that can be performed concurrently. The sequential scan on *lesson* can be performed while the corresponding scan and hashing of *timeslot* is performed. Each of these processes start at time cost 0.00, and while one ends at 2.00 the other one ends at 3.40. The hash join then starts where they lead of, at expected time cost 3.41.

The query then proceeds up the tree ending at an expected cost of 5.74. So much for the EXPLAIN-clause. The ANALYZE-clause then clues us in to the measured times and we see a break with expectation at exactly the same level.

While the scan-and-hash of *timeslot* indeed takes more time than the scan of *lesson*, the time where the hash join is started is a couple of milliseconds after either of them, orders of magnitude later. This will of course be different system to system.

Also, given that the majority of the total expected time cost is due to simply reading the data in the first place (something more or less inevitable), and that this data is filtered before being operated on in later steps, I doubt the query could be improved much beyond the level it is at.