Seminar 3

Data Storage Paradigms, IV1351

Saina Shamshirdar

2023-12-04

1 Introduction

The purpose in this task is to create queries for different output results from the database created in the previous tasks. A number of four different queries must be written which should generate relevant result showed in tables. Also, the sufficiency of one of the queries should be analyzed with EXPLAIN ANALYZE. The sufficiency of the data inserted into the database should also be checked, so that the queries can operate as expected. *In this task, I worked with my group mates Ruth Jenbre Shewa and Lassya Desu.*

2 Literature Study

I have gained my knowledge about SQL from the lectures, the TIPS AND TRICKS file provided in the course and an online website that teaches SQL step by step and provides examples. It also allows one to try the queries by oneself to become more comfortable with how SQL works.

3 Method

The DBMS used to write the queries was Postgress. In order to verify that the queries worked as intended, the result table generated from running the queries were studied to check whether it provided the information required in the requirements.

Help was received at the course tutorials where course assistants verified in some cases that the answer was relevant and enough.

4 Result

Link to the git repository: https://github.com/ruthshewa/SoundGoodMusicSchool

The first query should extract how many lessons were taken each month and how many of each type of lessons were also taken in that month. To access the month of the date that a lesson was held, the extract function was used on the date_of_taken_lesson

attribute of the lesson entity(see figur 1). All the lessons in a month are counted and called as "total". In order to count the number of each specific lesson type, different cases were used. For example, if the lesson_type was "individual", then it was counted as one. All the individual lessons were then summed an put in a separate column for individual lessons. To count how many lessons were taken in a specific month the result was grouped by month. Figure 2 shows the result in a table. For example, in April, there were a total number of 7 lessons taken where 4 of them were ensemble and 3 of them group lessons.

```sql
SELECT
  TO_CHAR(lesson.date_of_taken_lesson, 'MONTH') AS month_,
  COUNT(*) AS total,
  SUM(CASE WHEN lesson_type = 'Individual' THEN 1 ELSE 0 END) AS individual_lessons,
  SUM(CASE WHEN lesson_type = 'Group' THEN 1 ELSE 0 END) AS group_lessons,
  SUM(CASE WHEN lesson_type = 'Ensemble' THEN 1 ELSE 0 END) AS ensemble_lessons
FROM
  lesson
GROUP BY
  month_
ORDER BY
  month_;
```

Figure 1: Query 1.

	month_ text	total bigint	individual_lessons bigint	group_lessons bigint	ensemble_lessons bigint
1	APRIL	7	0	3	4
2	AUGUST	6	1	3	2
3	DECEMBER	5	2	3	0
4	FEBRUARY	5	2	1	2
5	JANUARY	4	3	0	1
6	JULY	6	2	3	1

Figure 2: A part of the result table for query 1.

The second query should show how many students there are with one, two or no siblings. According to figure 1 two columns will appear in the result, number of siblings as ”num_siblings” and number of students as ”num_students” because they are given in the first SELECT clause. The number of students is determined by a COUNT function in the SELECT clause. To determine the number of siblings of each student, a nested query was written in the FROM clause. In this query, the number of siblings is counted from the student_sibling entity by counting the sibling ids. It is then called num_siblings. A left join is done on the student and student_siblings entities where a student.id from the student table should match a student_id in the student_sibling table. The left join is done, since the number of students with no siblings, meaning no id match, is also required. These are then grouped by student_id. Outside the nested query, the condition for the number of possible siblings is decided in the WHERE clause. It should be less than or equal to two and be grouped by the number of siblings, since three rows are required. The total students with zero, one and two siblings are then recorded.

```sql
SELECT
    num_siblings,
    COUNT(student_id) AS num_students
FROM (
    SELECT
        student.id AS student_id,
        COUNT(student_sibling.sibling_id) AS num_siblings
    FROM
        student
    LEFT JOIN
        student_sibling ON student.id = student_sibling.student_id
    GROUP BY
        student.id
) AS student_siblings_count
WHERE
    num_siblings <= 2
GROUP BY
    num_siblings
ORDER BY
    num_siblings;

```

Figure 3: Query2.

	num_siblings bigint	num_students bigint
1	0	35
2	1	14
3	2	1

Figure 4: Result table for query2.

The third query is supposed to illustrate the name and the id of each instructor who has held a lesson during the current month. The total number of given lessons should also be visible for each instructor. Since three columns should appear in the result table, the SELECT clause in the first line in figure 3 includes the name and the id from the person respective the instructor entity. The number of lesson ids from the lesson entity is counted by a COUNT function. After that, ids from the person entity and the instructor entity are joined to discover which ids belong to an instructor. When that is determined, it is joined with the lesson ids to figure out which lesson has been held by which instructor. The current month is obtained by using the EXTRACT function and extracting the month of the current date from the attribute "date_of_taken_lesson" which is of the type "TIMESTAMP". Then the final result is simply grouped by name, id and the current month. It is also ordered by instructor name. The table in figure 4 shows five instructors that have held one lesson each in this month.

```
1  SELECT person.name, instructor.id, COUNT(lesson.id) AS NOlessons
2  FROM person
3  JOIN instructor ON person.id = instructor.id
4  JOIN lesson ON instructor.id = lesson.instructor_id
5  WHERE EXTRACT(MONTH FROM date_of_taken_lesson)=EXTRACT(MONTH FROM CURRENT_DATE)
6  GROUP BY person.name, instructor.id, EXTRACT(MONTH FROM date_of_taken_lesson)
7  ORDER BY person.name
8
```

Figure 5: Query 3.

	name character varying (50)	id integer	nolessons bigint
1	Alec Sweeney	11	1
2	Cheyenne Hill	4	1
3	Kadeem Ingram	12	1
4	Kimberley Bailey	3	1
5	Patricia Malone	13	1

Figure 6: Result table for query3.

To handle this query, a subquery, from line 10 to 23 in figure 5, was created to extract data about the lesson genre, the day the lesson was taken, the number of students involved in the lesson and the maximum number of students allowed in the lesson. Since the only lesson type that is of interest is ensemble, the WHERE clause applies this condition. The subquery finally groups the information by genre, day and the maximum number of spots allowed in a lesson.The subquery is now treated like an individual entity which is used in the FROM clause in the outer query. To differ the number of seats left in every lesson a case function is applied. If the number of students is the maximum number minus one, then there is one seat left. The same way, there is two seats left if the maximum number minus two is the number of students. Otherwise, there are many seats left. Figure 6 shows the resulted table with three columns as required. There are two ensemble lessons next week on Saturday and Friday which both have many seats left. The respective genres are given too which are Punk rock and gospel band.

```
1  SELECT
2    genre,
3    weekday,
4    CASE
5      WHEN NOstudents = max_number_of_spots - 1 THEN 'One seat left'
6      WHEN NOstudents = max_number_of_spots - 2 THEN 'Two seats left'
7      ELSE 'Many seats left'
8    END AS numberoffreeseats
9  FROM (
10   SELECT
11     lesson.genre,
12     TO_CHAR(lesson.date_of_taken_lesson, 'DAY') AS weekday,
13     COUNT(student_lesson.student_id) AS NOstudents,
14     lesson.max_number_of_spots
15   FROM
16     lesson
17   JOIN
18     student_lesson ON lesson.id = student_lesson.lesson_id
19   WHERE
20     lesson.lesson_type = 'Ensemble'
21   GROUP BY
22     lesson.genre, weekday, lesson.max_number_of_spots
23 ) AS subquery
```

Figure 7: Query 4.

	genre character varying (50)	weekday text	numberoffreeseats text
1	punk rock	SATURDAY	Many seats left
2	gospel band	FRIDAY	Many seats left

Figure 8: Result table for query4.

5 Discussion

The query plan for the fourth query is going to be analyzed here with the EXPLAIN ANALYZE. Starting from the bottom of the plan in figure 9, the query planner will do a sequential scan of the "lesson" table, indicated by the text "Seq Scan on lesson" on line 10. It is estimated that 25 rows will be retrieved from the table. The filter on line 9 is the condition for the scanning where only the rows with ensemble as the lesson type should be scanned. The hash operation on line seven involves hashing the hashing the results of the previous "Seq scan on lesson". A sequential scan is done on the student_lesson table which costs 1.51 and the estimated rows to scan is 51. Hash join on line 4 operates on the hashed results and the student_lesson table. The hash condition is the join condition where lesson id from student table and student_lesson table are joined. Hash aggregate groups the results based on the specified keys and performs an aggregate function, which in this case is COUNT. Estimated cost for this operation is 7.37. The group keys are lesson.genre, to_char(lesson.date_of_taken_lesson, 'DAY'::text) and lesson.max_number_of_spots. Subquery Scan on subquery represents the whole subquery which is estimated to cost 7.53 to scan 16 rows. It includes a HashAggregate operation that groups the results from the previous Hash Join and a Subquery Scan to represent the subquery.

"Planning Time: 0.338 ms" This is the time taken by the PostgreSQL query planner to generate the optimal query plan. During the planning phase, PostgreSQL analyzes the structure of the query, considers available indexes, and decides on the most efficient way to execute the query. The time is measured in milliseconds.

"Execution Time: 0.230 ms"

This is the actual time taken to execute the query. It includes the time spent fetching and processing the data. The execution time is the duration the query takes to run and produce the final result. Again, the time is measured in milliseconds. A planning time of 0.338 ms is relatively low and suggests that the query planner was able to quickly determine an efficient plan.

The goal is typically to have both Planning Time and Execution Time as low as possible, as this indicates efficient query planning and execution. These timings can be valuable for performance tuning and understanding how much time is spent at each stage of the query process. The values can vary based on factors such as the complexity of the query, the size of the dataset, and the available system resources. An execution time of 0.230 ms is also quite low, indicating that the query executed quickly. Based on the timings, the query performs efficiently.

The database design didn't need to change at all to simplify any query. The queries were written easily and output results were retrieved conveniently.

	QUERY PLAN text
1	Subquery Scan on subquery (cost=7.17..7.53 rows=16 width=70)
2	-> HashAggregate (cost=7.17..7.37 rows=16 width=50)
3	Group Key: lesson.genre, to_char(lesson.date_of_taken_lesson, 'DAY'::text), lesson.max_number_of_sp…
4	-> Hash Join (cost=5.31..7.01 rows=16 width=46)
5	Hash Cond: (student_lesson.lesson_id = lesson.id)
6	-> Seq Scan on student_lesson (cost=0.00..1.51 rows=51 width=8)
7	-> Hash (cost=5.00..5.00 rows=25 width=22)
8	-> Seq Scan on lesson (cost=0.00..5.00 rows=25 width=22)
9	Filter: ((lesson_type)::text = 'Ensemble'::text)

Figure 9: Query plan for query4.

6 Comments About the Course

It was very unclear what and how much was required considering the EXPLAIN ANALYZE. There should be examples of an explain analyze on a query so that one can base their work on.