

Seminar 3

Completed Version

Data Storage Paradigms, IV1351

Ayda Ghalkhanbaz
aydag@kth.se

January 7, 2024

Contents

| | |
|----------------------------|----------|
| Introduction | 3 |
| Literature Study | 3 |
| Method | 4 |
| Result | 5 |
| Discussion | 7 |
| Attachments | 9 |

Introduction

The primary aim of this task is to write OLAP queries in order to retrieve the desired data from the database which was created in the previous task. Additionally, the efficiency of one of these queries has been comprehensively analysed which is discussed in later sections. This report mainly covers how and why such OLAP queries have been written.

Literature Study

To gain enough knowledge for completing this task, I watched the lecture on “SQL- The Structured Query Language”. Additionally I did numerous searches in different external sources; such as the Postgresql documentation which was provided for this task.

Method

The DBMS utilised for this task is *Postgresql*, and the queries were created using *pgAdmin4*.

Developing each query included several steps. Initially, each query was divided into smaller parts, with each part tested individually to observe the outcomes. Subsequently, these parts were then merged and adjusted to achieve the desired result.

Since there is a reasonable amount of data on the database, it was easy to retrieve all data from each table and check manually whether the results of the queries match the actual data.

Result

The SQL scripts for both the mandatory and the higher grade part of this task can be found on [GitHub](#). **The higher grade part of this task has been completed which is about creating a historical database and copying data from school's database. The historical database has three tables: "student", "lesson" and "booking" with several columns . The data from school's database was then transfered using *dblink* extension.**

Number of lessons given per month

The query for this bullet point, as demonstrated in listing [1](#), starts with listing the year and month of the lessons. The third column counts the total number of given lessons and the remaining columns indicate the total number of each type of lesson during a specific month.

For summing up the number of specific lessons, subqueries have been used. Each type of lesson, which has its own table, contains the general lesson ID. If the lesson ID is found in any of these tables, it means that there is a lesson of that type, so the number of that specific type of lesson should be increased by one. This filtering is achieved by utilising the **WHEN** clause. The output of running this query is illustrated in figure [1](#).

Number of siblings

This query, see listing [2](#), also employs subqueries to calculate how many siblings a student has by matching the student IDs in the student table with the student IDs in the sibling table. Then the result of this subquery, which is shown in figure [2](#), is grouped by the number of siblings to summarise the total number of students with 0, 1 or 2 siblings.

Hardworking Instructors

This query had a simpler structure than the others. As it is demonstrated in listing [3](#), the query employs **INNER JOIN** to find the persons who are instructors, and also the lessons that are given by those instructors. As the output of this query shows in figure [3](#), there are 3 instructors with the most given lessons during this month.

Figure [4](#) illustrates the query plan as a result of executing the **EXPLAIN** clause, which will be discussed in the next chapter.

Ensembles during next week

The script of this query is illustrated in listing [4](#).

This part of the task was the most challenging and time consuming. This query employs several **CASE** clauses and also subqueries. The **CASE** clauses are used to specify the weekday that corresponds to the number of the day in a week. Another **CASE** clause was used to handle different scenarios of empty spots for each ensemble. Additionally, several **JOIN** clauses were employed to find the ensembles and their genres.

Since there was only one ensemble per week in the created database, two different outputs are shown in figure [5](#) and [6](#). Figure [5](#) demonstrates the ensembles during the next week, and figure [6](#) shows the ensembles during 3 weeks.

Discussion

As we can observe in the scripts, the first three queries are stored as a non-materialised view while the last query is stored as a materialised view. The reason behind choosing the non-materialised view for the first three tasks is that the data may change frequently, and more importantly, these queries will be executed multiple times per week.

Another reason can be that the three first queries don't include complicated calculations. A non-materialised view is suitable in cases like this; that is the data may change frequently so we need to have up-to-date data, it isn't necessary to precompute complicated calculations and the queries have simpler structure so that they could be executed fast.

When it comes to the last query, we can see that it requires calculations and the structure is way more complicated than the others. Moreover, it wasn't clearly specified how frequently the data is retrieved. By assuming that it is okay to update the data periodically and the delay due to updating the data is tolerable, a materialised view can be suitable considering the mentioned conditions.

The query plan for the third query of this task, as mentioned earlier, is illustrated in figure 4. As we can observe, the hash join operation on lines 5 and 7 and the sequential scan on line 9 are the most expensive operations for the third query. The hash joins on line 5 and 7 are performed between instructor and person, and instructor and lesson tables, respectively.

The interesting point about the cost of seq scan on line 9 is that this is where we scan the lesson's start time and filter them to find the lessons during the current month. The seq scan operation has a cost between 0.00...48.10.

Since the hash joins and seq scan are the most expensive operations, it is smarter to apply optimizations on these parts. For instance, the size of tables matters when it comes to hash joins. More columns in a table may lead to costly hash joins. This could be improved in this case, by reducing the size of the person table and storing the address data for each person in its own table. For the seq scan, one optimization could be rephrasing the filter conditions, which in this case I've tried to keep as simple as possible.

Moreover, a historical database containing the information about the lessons, their types and the students who booked them, was created as specified in the higher grade part of the task (which, as mentioned earlier, can be found on [GitHub](#)).

Historical databases include denormalizing the design of the database for instances by having duplicated data. There are both several advantages and disadvantages of denormalizing the database. One can mention improved query performance, simplified query structure and reduced JOIN operations as advantages of denormalization. Fewer tables lead to reduced need of using JOIN operations. This also results in a simpler

way to write, read and understand queries. While the data redundancy and higher requirements on increased storage may be drawbacks of the denormalization.

In conclusion, denormalization is an efficient way for storing the older data that might be retrieved for analytical purposes while it has several disadvantages such as increased risk of anomalies and data maintenance complexity.

Attachments

```

1 CREATE VIEW number_of_lessons AS
2 SELECT -- getting the year and month from the date of
      lessons
3     EXTRACT(YEAR FROM start_time) AS year,
4     CASE
5     WHEN EXTRACT(MONTH FROM start_time) = 11 THEN 'NOVEMBER'
6     WHEN EXTRACT(MONTH FROM start_time) = 12 THEN 'DECEMBER'
7     WHEN EXTRACT(MONTH FROM start_time) = 1 THEN 'JANUARY'
8     END AS month,
9
10    COUNT(id) AS total,
11    SUM(CASE WHEN id IN (SELECT lesson_id FROM
12        individual_lesson) THEN 1 ELSE 0 END) AS individual,
12    SUM(CASE WHEN id IN (SELECT lesson_id FROM group_lesson)
13        THEN 1 ELSE 0 END) AS group,
13    SUM(CASE WHEN id IN (SELECT lesson_id FROM ensembles) THEN
14        1 ELSE 0 END) AS ensemble
14
15 FROM lesson
16 -- can be removed to get the lessons during next year too
17 WHERE EXTRACT (YEAR FROM start_time) = EXTRACT (YEAR FROM
18     CURRENT_DATE)
18 GROUP BY month,year
19 ORDER BY year,month DESC;

```

Listing 1: The script for the first query







| | year numeric  | month text  | total bigint  | individual bigint  | group bigint  | ensemble bigint  |
|---|---|---|---|--|---|--|
| 1 | 2023 | NOVEMBER | 7 | 5 | 2 | 0 |
| 2 | 2023 | DECEMBER | 6 | 0 | 3 | 3 |

Figure 1: The output of the first query

```

1 CREATE VIEW number_of_siblings AS
2 -- no_of_siblings depends on the subquery. selects two
   columns for the result
3 SELECT no_of_siblings, COUNT(id) AS no_of_students
4 FROM (
5     SELECT student.id, COUNT(sibling.sibling_student_id) AS
       no_of_siblings FROM student
6     LEFT JOIN sibling ON student.id = sibling.student_id GROUP
       BY student.id)
7     AS no_of_siblings
8 -- grouping by the number of siblings..0, 1, 2
9 GROUP BY no_of_siblings
10 ORDER BY no_of_siblings;

```

Listing 2: The script for the second query

| | no_of_siblings bigint | no_of_students bigint |
|---|--------------------------|--------------------------|
| 1 | 0 | 55 |
| 2 | 1 | 42 |
| 3 | 2 | 3 |

Figure 2: The output of the second query

```

1 CREATE VIEW instructor_given_lessons AS
2 SELECT i.id AS instructor_id, p.first_name , p.last_name ,
3        COUNT(l.id) AS no_of_lessons
4 FROM lesson AS l
5 INNER JOIN instructor AS i ON l.instructor_id = i.id
6 INNER JOIN person AS p ON i.person_id = p.id
7
8 WHERE (SELECT EXTRACT(MONTH FROM l.start_time) = EXTRACT(
9        MONTH FROM CURRENT_DATE))
9 GROUP BY i.id, p.first_name , p.last_name
10 ORDER BY COUNT(l.id) DESC;

```

Listing 3: The script for the third query

| | instructor_id integer | first_name character varying (200) | last_name character varying (200) | no_of_lessons bigint |
|---|---------------------------------|--|---|--------------------------------|
| 1 | 4 | Henry | Larsson | 3 |
| 2 | 1 | Lilian | Henriksson | 2 |
| 3 | 6 | Erik | Lundgren | 1 |

Figure 3: The output of the third query

| | QUERY PLAN text |
|----|--|
| 1 | Sort (cost=119.45..121.04 rows=635 width=27) |
| 2 | Sort Key: (count(l.id)) DESC |
| 3 | -> HashAggregate (cost=83.54..89.89 rows=635 width=27) |
| 4 | Group Key: i.id, p.first_name, p.last_name |
| 5 | -> Hash Join (cost=25.70..77.19 rows=635 width=23) |
| 6 | Hash Cond: (i.person_id = p.id) |
| 7 | -> Hash Join (cost=17.20..66.99 rows=635 width=12) |
| 8 | Hash Cond: (l.instructor_id = i.id) |
| 9 | -> Seq Scan on lesson l (cost=0.00..48.10 rows=635 width=8) |
| 10 | Filter: (SubPlan 1) |
| 11 | SubPlan 1 |
| 12 | -> Result (cost=0.00..0.02 rows=1 width=1) |
| 13 | -> Hash (cost=13.20..13.20 rows=320 width=8) |
| 14 | -> Seq Scan on instructor i (cost=0.00..13.20 rows=320 width=... |
| 15 | -> Hash (cost=6.00..6.00 rows=200 width=19) |
| 16 | -> Seq Scan on person p (cost=0.00..6.00 rows=200 width=19) |

Figure 4: The query plan for the third query

```

1 CREATE MATERIALIZED VIEW number_of_seats AS
2 SELECT
3     CASE
4         WHEN EXTRACT(DOW FROM l.start_time) = 1 THEN 'Monday'
5         WHEN EXTRACT(DOW FROM l.start_time) = 2 THEN 'Tuesday'
6         WHEN EXTRACT(DOW FROM l.start_time) = 3 THEN 'Wednesday'
7         WHEN EXTRACT(DOW FROM l.start_time) = 4 THEN 'Thursday'
8         WHEN EXTRACT(DOW FROM l.start_time) = 5 THEN 'Friday'
9         WHEN EXTRACT(DOW FROM l.start_time) = 6 THEN 'Saturday'
10        WHEN EXTRACT(DOW FROM l.start_time) = 7 THEN 'Sunday'
11    END AS weekday,
12    DATE(l.start_time),
13    g.genre_name AS genre,
14
15    CASE
16        WHEN e.max_students - COUNT(student_id) FILTER (WHERE
17            student_id IN (SELECT student_id FROM lesson_booking))
18            >2 THEN 'Many seats'
19        WHEN e.max_students - COUNT(student_id) FILTER (WHERE
20            student_id IN (SELECT student_id FROM lesson_booking))=
21            0 THEN 'No seats'
22        ELSE '1 or 2 seats'
23    END AS free_seats
24 FROM lesson AS l
25 JOIN ensembles AS e ON e.lesson_id = l.id
26 JOIN genre AS g ON e.genre_id = g.genre_id
27 LEFT JOIN lesson_booking AS lb ON lb.lesson_id = l.id
28 -- this gives the result for the next week only
29 -- WHERE(SELECT EXTRACT(WEEK FROM l.start_time) = EXTRACT(
30     WEEK FROM CURRENT_DATE + INTERVAL '1 WEEK'))
31 WHERE l.start_time BETWEEN CURRENT_DATE AND CURRENT_DATE +
32     INTERVAL '3 WEEK'
33 GROUP BY l.start_time, g.genre_id, e.max_students
34 ORDER BY weekday, genre;

```

Listing 4: The script for the fourth query





| | weekday  text | date  date | genre  character varying (500) | free_seats  text |
|---|---|--|--|--|
| 1 | Tuesday | 2023-12-12 | Rock | Many seats |

Figure 5: The output of the fourth query: showing the ensembles during the next week

| | weekday text 🔒 | date date 🔒 | genre character varying (500) 🔒 | free_seats text 🔒 |
|---|--------------------------|-----------------------|---|-----------------------------|
| 1 | Monday | 2023-12-04 | Jazz | No seats |
| 2 | Thursday | 2023-12-21 | Blues | Many seats |
| 3 | Tuesday | 2023-12-12 | Rock | Many seats |

Figure 6: The output of the fourth query: showing the ensembles during the 3 weeks