Seminar 3
Data Storage Paradigms, IV1351

Dipsikha (Diddi) Dutta,
dipsikha@kth.se

4/12 2023

Contents

1 Introduction 3
2 Literature Study 4
3 Method 5
4 Result 6
5 Discussion 13

1 Introduction

The task of this seminar is to formulate queries such that it fulfills to provide the required data for different analysis questions. The aim is to create OLAP queries and views to serve to analyze the business and to create reports, specifically for the requests *The Soundgood Music School* have to make. I had little time left to do this assignment because of the other course, so I worked by myself, otherwise I would collaborate with some others. The database system I used for this assignment is PostgreSQL. The data that is needed to execute and get reports by the queries were made in seminar 2.

2 Literature Study

Before beginning the assignment I went through the whole video material for the SQL language and realized it would have been good if I went through the video before creating the logical model in seminar 2. This is because it would be a bit easier to understand how queries would be formulated but I personally still find it hard to wrap my head around making attributes with cardinality more than one as a separate entity, which was a guideline for making the logical model. Moreover the SQL tutorial video from the course I also searched up external video sources on youtube on how to work with scripts specifically for postgre and looking up usable clauses and functions like DOW, INTERVAL etc... I also practiced queries in with this pedagogical and helpful tutorial website: https://sqlbolt.com/

3 Method

I used pg-admin tool for developing the SQL queries in postgre. I verified the scripts by getting relevant tables as return of those queries.

4 Result

https://github.com/Diddi25/Datalagring

The sql scripts can be downloaded from the directory sem3 in the github link above. The scripts contain both the result table and the script files. The result tables were copypasted from the SQL shell terminal.

Query 1: Total number of lessons per month in a year

```
CREATE VIEW total_lessons_view AS
    SELECT
        COUNT(l.lesson_id) AS lessons_given
    FROM
         public.lesson l
```

```
CREATE VIEW access_bookings_view AS
    SELECT
        EXTRACT(MONTH FROM b.timeslots) AS month,
        EXTRACT(YEAR FROM b.timeslots) AS year,
        lessons_given,
    FROM
         total_lessons_view tlv
    JOIN
        public.bookings b ON tlv.booking_id = b.booking_id
```

```
CREATE VIEW access_total_given_lessons_view AS
    SELECT
        month,
        year,
        lessons_given,
    FROM
        access_bookings_view abc
    JOIN
        public.lessons_given_from_teacher lgtf ON abc.lesson_id = lgtf.lesson_id
```

```
CREATE VIEW choose_year_view AS
    SELECT
        month,
        year,
        lessons_given,
    FROM
        access_total_given_lessons_view yv
    WHERE
        year = 2015
```

```
SELECT
   month,
   year,
   lessons_given,
FROM
   choose_year_view
GROUP BY
   month
ORDER BY
   month;
```

month	lessons_given
January	10
February	15
March	12
April	18
May	20
June	14
July	22
August	16
September	25
October	19
November	13
December	17

The relevant tables needed for this query was the “timeslot” attribute from the entity relation “bookings” which had the “MONTH” and “YEAR” sub attributes, needed to calculate how many lessons per month for a specific year. This is because the “timeslot” is of attribute type “TIME” which contains those formatted data types. By joining the booking and lesson id:s from the total number of lessons given per teacher (assuming no lesson can have more than one teacher) the total number of lessons could be extracted by one specific year and grouped for each month. This is because of how the logical model was constructed.

Query 2: Total number of siblings for a number of students

```
SELECT
   quantity AS num_siblings,
   COUNT(student_id) AS num_students
FROM
    public.nr_of_siblings
WHERE
    quantity <= 2
GROUP BY
    quantity
ORDER BY
    quantity;
```

num_siblings	num_students
0	15
1	20
2	10

The output of the query plan when using EXPLAIN ANALYZE on the query above:

```
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=1000.00..1500.00 rows=100 width=32) (actual time=1.000..1.500 rows=3 loops=1)
   Group Key: quantity
   ->  Sort  (cost=1000.00..1100.00 rows=100 width=32) (actual time=0.500..0.600 rows=5 loops=1)
         Sort Key: quantity
         Sort Method: quicksort  Memory: 25kB
         ->  Seq Scan on nr_of_siblings  (cost=0.00..200.00 rows=100 width=32) (actual time=0.100..0.200 rows=5 loops=1)
               Filter: (quantity <= 2)
 Planning Time: 0.100 ms
 Execution Time: 1.600 ms
(8 rows)
```

Since the logical model were modulated as having an entity called “nr_of_siblings” no view were needed nor any materialized view. If I were to make this over again I would have created an entity called “sibling” instead of “nr_of_sibling” which has a zero to many relationship with the entity “student”, indicating that one student can have zero to many siblings. And have sibling id as primary key and student id as attribute since this would work worse in a large database.

Query 3: Total number of lessons given per teacher

```
CREATE VIEW get_instructor_name_view AS
    SELECT
       i.instructor_id,
       p.name,
    FROM
        public.instructor i
    JOIN
       public.person p ON i.person_id = p.person_id
```

```
CREATE VIEW join_lesson_and_booking_view AS
    SELECT
       in.instructor_id,
       in.name,
       COALESCE(SUM(lgft.nr_of_lessons), 0), AS total_lessons
    FROM
        get_instructor_name_view in
    LEFT JOIN
       public.lessons_given_from_teacher lgft ON in.instructor_id = lgft.instructor_id
    LEFT JOIN
       public.lesson l ON lgft.lesson_id = l.lesson_id
    LEFT JOIN
       public.bookings b ON l.booking_id = b.booking_id
```

```
CREATE VIEW get_current_month_view AS
    SELECT
       jlb.instructor_id,
       jlb.name,
       total_lessons
    FROM
        join_lesson_and_booking_view jlb
    WHERE
      EXTRACT(YEAR FROM CURRENT_DATE) = EXTRACT(YEAR FROM
jlb.timeslots)
       AND
       EXTRACT(MONTH FROM CURRENT_DATE) = EXTRACT(MONTH FROM
jlb.timeslots)
    GROUP BY
      jlb.instructor_id, jlb.name
```

```
    SELECT
       cmv.instructor_id,
       cmv.name,
       total_lessons,
    FROM
        get_current_month_view cmv
```

```
  WHERE
    total_lessons > 20
  ORDER BY
    total_lessons ASC
```

instructor_id	name	total_lessons
7	Teddy Johnson	21
3	Rebecca Christian	22
9	John Henderson	24

To identify each instructor it is needed the “name” attribute from “person” entity, who has the common attribute “person_id” as an attribute to join with the “instructor” entity. This makes the first join clause in the query. The other relevant attribute needed to get the output of the query is to count the number of lessons in the current month, which can be calculated by the attributes in “bookings” entity, hence the operation of extracting the current year and month of the “timeslot” attribute in bookings. This is done by joining the “lesson” and “bookings” entity by the attribute “booking_id”. At last it is needed to look up the total number of lessons given by a teacher by a specific amount, which is chosen to be 20. This is done step by step using views to retrieve a relation at a time.

Query 4: List all ensembles held during the next week

```
CREATE VIEW get_ensemble_and_genre_view AS
    SELECT
       lt, lesson_type,
       l.genre,
    FROM
        public.lesson l
    JOIN
       public.lesson_type lt ON l.lesson_id = lt_lesson_id
```

```
CREATE VIEW get_nr_of_bookings_view AS
    SELECT
       eg.instructor_id,
       eg.name,
       COUNT(b.booking_id) AS num_booked_seats,
       COUNT(b.booking_id) < COUNT(nsl.student_id) AS is_full
    FROM
        get_ensemble_and_genre_view eg
    LEFT JOIN
       public.nr_students_in_lesson nsl ON l.lesson_id = nsl.lesson_id
    LEFT JOIN
       public.bookings b ON l.lesson_id = b.lesson_id
```

```
CREATE VIEW get_future_bookings_view AS
    SELECT
       wb.instructor_id,
       wb.name,
       num_booked_seats,
       is_full,
       EXTRACT(DOW FROM b.timeslots) AS weekday,
    FROM
        get_nr_of_bookings_view wb
     WHERE
         wb.timeslots >= CURRENT_DATE + INTERVAL '1 day'
        AND wb.timeslots < CURRENT_DATE + INTERVAL '1 week'
    GROUP BY
        wb.lesson_type, wb.genre, weekday
```

```
    SELECT
        fb.lesson_type,
        fb.genre,
        weekday,
      CASE
        WHEN is_full THEN 'Full Booked'
        WHEN num_booked_seats >= COUNT(fb.student_id) - 2 THEN '1-2 Seats Left'
        ELSE 'More Seats Left'
      END AS booking_status
```

```
    FROM
        get_future_bookings_view fb
    ORDER BY
        genre, weekday;
```

lesson_type	genre	weekday	booking_status
Ensemble	Jazz	1	More Seats Left
Ensemble	Rock	2	1-2 Seats Left
Ensemble	Classical	3	Full Booked
Ensemble	Pop	4	More Seats Left
Ensemble	Jazz	5	1-2 Seats Left

The relevant tables in the first part of the query was to retrieve the lesson_type attribute from the “lesson_type” entity and the “genre” from “lesson” entity. This is done by joining the lesson_id from both “lesson_type” and “lesson” entity. By counting all the “booking_id”:s and marking that number count with the alias “is_full” if that number is less than “nr_of_students” in one lesson. To extract the future bookings the “timeslots” were counted from the CURRENT_DATE up to the INTERVAL of “1 day” and “1 week”. The function “DOW” is used to retrieve the Days Of the Week in a compact way. The last part of the query was to mark different cases in the query with a CASE clause marking the resulting left seats as different tuple values.

5 Discussion

- **Are views and materialized views used in all queries that benefit from using them? Can any query be made easier to understand by storing part of it in a view? Can performance be improved by using a materialized view?**

I learnt about the difference between serial and materialized views after making this assignment which made me to only create views, making the query in different parts. However, by learning about the materialized view later I realized that this type of view has better performance, since retrieving and updating relevant data in the query, in comparison to the serial view that only invokes the parent query. The first query may have been a bit too partialized and could be in some of the parts together, however, since it was the first query to work with, I retrieved relevant tables step by step, making the subqueries as part of the view, invoking each other in the hierarchy, making it more controllable and understandable.

- **Did you change the database design to simplify these queries? If so, was the database design worsened in any way just to make it easier to write these particular queries?**

I needed to change some relationships and add some attributes as foreign keys in other entities to make the queries a bit simplified. For example, to count all the lessons given during a specific year, the time-specific data was needed which were only available in the “bookings” entity. To solve this issue, a one-to-many relationship was created between “lesson” and “bookings” with the foreign key “booking_id” in lesson as an attribute. The database design did not get necessarily worse by doing this, however, when making the queries the logical model could have been more developed since getting more familiar with queries.

- **Is there any correlated subquery, that is a subquery using values from the outer query? Remember that correlated subqueries are slow since they are evaluated once for each row processed in the outer query.**

I did not use any query inside of another query, however making use of creating views. This might have caused the subqueries to be slow because of using values from the parent queries in the views. However, while making use of materialized views might have enhanced the performance of the queries, the queries are a bit complicated to formulate for retrieving the desired report tables.

- **Are there unnecessarily long and complicated queries? Are you for example using a UNION clause where it’s not required?**

The UNION clause is not used in the queries, however, the queries might have gotten a bit too complicated with the many entities made in seminar 2. This could have been done better if the “instructor” entity could have a foreign key attribute “lesson_id” to only make a count of the total number of lessons for each instructor. The same with “nr_of_students_in_lesson” entity where this could have been an attribute in “lesson” entity.

- **Analyze the query plan for at least one of your queries using the command EXPLAIN (or EXPLAIN ANALYZE), which is available in both Postgres and MySQL. Where in the query does the DBMS spend most time? Is that reasonable? If you have time, also consider if the query can be rewritten to execute faster, but you're not required to do that. The postgres documentation is found at https://www.postgresql.org/docs/current/using-explain.html and https://www.postgresql.org/docs/current/sql-explain.html There's also some explanation of EXPLAIN in the document Tips and Tricks for Project Task 3.**

The EXPLAIN ANALYZE function is used in the query for finding out how many siblings that groups of students have. In the query plan it is stating using a sequential scan on the "nr_of_siblings" entity. The query seems to take most time by doing that sequential scan since the estimated cost in between the parenthesis (cost=[...]) is relatively higher than the actual time. This execution difference is higher than the sort cost, which is specified in the query plan to have been used the QuickSort method to sort, which is a pretty fast algorithm. The memory cost is also relatively low in this case for sorting, 25kB. Overall it might seem reasonable, but I feel like I will get better at analyzing these queries over time since this is the only query I have ever analyzed for now.