Seminar 3 - SQL

Data Storage Paradigms, IV1351

December 2024

Project members:
[Adrian Boström, adrbos@kth.se]
[Viktor Sandström, visand@kth.se]
[Lucas Rimfrost, rimfrost@kth.se]

1 Introduction

Now given that we have a logical model and a physical model from previous assignments, the natural next step is to have data in it and use it. We are tasked with query creation. Only having a structure, with no data or way to use it is pointless. The people contributing to this task were: Adrian Boström, Lukas Rimfrost and Viktor Sandström.

2 Literature Study

The standard query language (SQL) is a query language for large data models. Perfect for writing reading and searching for data. To solve the tasks from this assignment PostgreSQL was used. Taking information from the provided sources such as course literature, lectures and the provided video and document. The SQL language is a query language that abstracts the data from the user such that they do not have to implement how said data is acquired, rather to what data it should provide. This is one great reason as to why SQL has become so popular and efficient.

3 Method

All our code for generating the physical server and to input the data can be found on *this GitHub*. The DBMS for this database is PostgreSQL and the initial challenge was to take the database structure we created from previous tasks and upgrade it to allow more specific queries that depended on relations previously foreseen. This was done by comparing our database with the tasks at hand, creating a theoretical pseudo query with decencies. For example there was task 4 of which needed to count how many spots where left for each lesson, then we needed to know how many students attended each lesson so a entity student_attendance was created.

After each dependency necessary was available, there needed to be more data. This data was primary generated with the generatedata.com webpage. So for a certain entity for example person:

```
CREATE TABLE public.person (
    id serial PRIMARY KEY,
    person_number varchar(12) NOT NULL,
    first_name varchar(50) NOT NULL,
    last_name varchar(50) NOT NULL,
    phone varchar(20) NOT NULL,
    email varchar(50) NOT NULL
);
```

The public.person was specified as a table with attributes person_number, first_name, last_name, phone and email. Since id is a serial primary key it is decided on generation and should not be decided otherwise. For names there where an option of having automatically generated names and possibility to specify only first name or last name. person_number required an number combination of which should be of the right format. But there where no good solution to this with automation so a randomly generated number was inserted instead. Phone was a bit more controlled since Swedish phone numbers are region dependent a Stockholm based phone number could be generated with 072 as the first three numbers. The webpage had an excellent feature to generate believable emails. As this database is still in development data could be unrealistic for now, but it should be taken into consideration that no faulty information is stored and breaks the system. Assigning foreign keys was by generating a random number from the range of primary keys.

Now for query generation; a concept should start simple. From a task it was usually about returning a list of entries of a certain type or in a certain table. To acquire each element from a a table an example query could look like this: *SELECT * FROM student.* This could then be built upon to perhaps only select students that have siblings. Then the *WHERE* statement should be used, it can filter out elements based on conditions. For example *WHERE student.sibling_id IS NOT NULL.* Steps like this can be combined to create more complex queries.

4 Queries

Now given a new set of data with hundreds of students

4.1 Number of lessons per month

Our first task is to make a query which returns the number of lessons in a given month and year, it will also subdivide the lessons into their types, namely: individual lesson, group lesson and ensembles. This will use the data from lesson and each individual relation for lesson type.

First we made a temporary table for each lesson table which counts per month how many of them it has. They all looked like this:

```
CREATE TEMP TABLE individual_count_temp AS
SELECT
    DATE_TRUNC('month', date) AS month,
    COUNT(*) AS individual_count
FROM
    public.lesson
RIGHT JOIN
    public.individual_lesson
ON
    public.lesson.lesson_id = public.individual_lesson.lesson_id
GROUP BY
    month
ORDER BY
    month;
```

After having a similar table for the other lesson types then we just need to join them and sort by year. That is done like this for 2024:

```
SELECT
    COALESCE(
        individual_count_temp.month,
        ensemble_count_temp.month,
        group_count_temp.month
    ) AS month,
    individual_count_temp.individual_count,
    ensemble_count_temp.ensemble_count,
    group_count_temp.group_count
FROM
    individual_count_temp
FULL OUTER JOIN
    ensemble_count_temp
ON
    individual_count_temp.month = ensemble_count_temp.month
FULL OUTER JOIN
    group_count_temp
ON
    COALESCE(
        individual_count_temp.month,
        ensemble_count_temp.month
    ) = group_count_temp.month
```

```
WHERE
    EXTRACT(
        YEAR FROM COALESCE(
            individual_count_temp.month,
            ensemble_count_temp.month,
            group_count_temp.month
        )
    ) = 2024;
```

Doing this to our data set we get 11 results, which are a mix of all lesson types, also including months which have no lessons, and all combinations of missing lesson types.

4.2 number of siblings

The next query is one which relates to siblings and their ids, we need to return the number of siblings which share the same sibling id, and count how many sibling groups their are of different sizes. This is how its done:

```
SELECT sibling_count, COUNT(*) AS num_siblings
FROM (
    SELECT sibling_id, COUNT(*) AS sibling_count
    FROM student
    GROUP BY sibling_id
) AS student_rentals
GROUP BY sibling_count
ORDER BY sibling_count DESC;
```

We have a subquery in which we count how many siblings their are which share the same sibling id, this then returns the wanted output.

4.3 teachers number of lessons

The next query we need to make is one which returns which teachers have more lessons than a specified number, we do this by creating a temporary table like in the first query.

```
    CREATE TEMP TABLE teaching_count_temp AS
SELECT
    DATE_TRUNC('month', date) AS month,
    person_id AS teacher_id,
    COUNT(*) AS teacher_count
FROM
    public.lesson
GROUP BY
    month,
    teacher_id
ORDER BY
    month;
```

Now with this information we can make the final query:

```
SELECT
    teacher_id,
    (SELECT
        CONCAT(first_name, ' ', last_name)
     FROM
        person
     WHERE
        person.id = teacher_id
    ) AS teacher_name,
    teacher_count as No_Of_Lessons
FROM
    teaching_count_temp
WHERE
    EXTRACT(MONTH FROM month) = EXTRACT(MONTH FROM CURRENT_DATE)
    AND EXTRACT(YEAR FROM month) = EXTRACT(YEAR FROM CURRENT_DATE)
    AND teacher_count > 0;
```

This returns a table with 2 teachers, we dont have a lot of lessons in the database so they both just have 1 lesson.

4.4 ensembles next week

This query will return next weeks ensembles using SQLs CURRENT_DATE. This is how we make that query:

```
SELECT
    TO_CHAR(lesson.date, 'Dy') AS Day,
    public.ensemble.genre,
    (
        public.ensemble.max_num_students -
        (
            SELECT
                COUNT(*)
            FROM
                public.student_attending
            WHERE
                student_attending.lesson_id = 39
        )
    ) AS spots_left
FROM
    public.ensemble
LEFT JOIN
    public.lesson
```

```
ON
    public.ensemble.lesson_id = public.lesson.lesson_id
WHERE
    EXTRACT(WEEK FROM lesson.date) = EXTRACT(WEEK FROM CURRENT_DATE) + 1;
```

This returns one ensemble next week which has genre ”classical” on thursday with the right amount of spots left.

5 Discussion

This task was completed only after a bit of tweaks to the database, it was not completely ready for complex queries; the upgrades provided where increase in character allocation for address city and street from 30 to 100 characters. There where also a primary key that was missing for certain queries in the student entity. A final adjustment for query four was to add the student_attending.

Each query are in a way dependent on subqueries since they mostly build their own table that then are used in the original query. These queries are suboptimal since they every positive hit from the original query requires its own new query that resembles an exponential increase in delay and computation time. This should be avoided at all costs but are often impossible to avoid. In complex queries it does require comparisons, sorting and or computation. There are multiple smart tricks that reduce computing time but eventually it will become really slow with high number of elements.

There are no clear to long queries or complicated queries, there have been attempts at optimization but there could be points of optimization like removing the temp table from query 1 and 3. There was planning put into the queries that sought to simplify said queries but it does not directly mean that they are in their simple and shortest form. This could be lack of developer experience and measurements during the implementation. UNION was not a problem in our queries.

We used analyzed query 2 which resulted in a few interesting characteristics of the search, the search itself took less than 0.2ms. The search is divided into two, planning time and execution time. The query uses quicksort to sort the result and uses very minimal amount of memory in the heap which is about 25kB.