# Seminar 3

Axel Månson Lokrantz

2022-12-05

## 1. Introduction

The goal for this seminar was to create online analytic processing OLAP, queries and views for the SoundGood Music School database. A query is a request for data from a database. It can answer questions, combine data into tables, perform calculations, update, change and add data. In at least one of the queries it is mandatory to use `EXPLAIN ANALYZE` which analyze the efficiency of a query. Prior to solving the task, the physical and logical model was adjusted according to the feedback we received.

The queries were written together with Daniel Dahlberg in pgAdmin Postgres tools 4 at the royal institute of technology in Stockholm.

## 2. Literature Study

To prepare for this seminar I watched Paris Carbone's video lecture on the SQL language, read Leif Lindbäcks pdf 'Tips and Tricks for Task 3' and read the corresponding chapters in the 7th edition of Fundamentals of Database Systems by Ramez Elmasri.

## 3. Method

The database was written in DBMS PostgresSQL, and the queries were developed using GUI tool pgAdmin. The content for the database was generated using an online data generator https://generatedata.com/.

To ensure accuracy of our queries we first had to thoroughly go through all the content of our database. This required some manual labour. For example making sure there were enough persons to create the desired amount of instructors and students. The output from all our queries were then cross-checked with the current state of the database to verify they matched. All queries are encapsulated in views for easier readability.

# 4. Results

## 4.1 Query 1

*Query 1.* Show the number of lessons given per month during a specified year. Order by lesson type, i.e (individual, group, ensemble)

```
CREATE VIEW month_lessons AS
    SELECT EXTRACT(YEAR FROM start_time) AS year,
    EXTRACT(MONTH FROM start_time) AS month,
    SUM(CASE WHEN max_num_students = 1 THEN 1 ELSE 0 END)
    AS individual_lesson,
    SUM(CASE WHEN min_num_students > 1
    AND lesson.id != ensemble.lesson_id THEN 1 ELSE 0 END)
    AS group_lesson,
    SUM(CASE WHEN lesson.id = ensemble.lesson_id THEN 1 ELSE 0 END)
    AS ensemble_lesson,
    COUNT (*) AS number_of_lessons_month
    FROM lesson, ensemble
    GROUP BY EXTRACT(YEAR FROM start_time),
    EXTRACT(MONTH FROM start_time);

-- example:
EXPLAIN ANALYZE(SELECT * FROM month_lessons WHERE year = 2022);
```

| year | month | individual_lsn | group_lsn | ensemble_lsn | total_lsn |
|---|---|---|---|---|---|
| 2022 | 12 | 3 | 1 | 4 | 8 |
| 2022 | 8 | 0 | 1 | 0 | 1 |

Table 1: Result from query 1.

To verify the validity of the query we made the lessons take place during two separate months (December and August) and at least one lesson which was scheduled for a different year. This way, we know for certain that the `WHERE year = 2022` and `month` column were properly implemented. The result of the query can be seen in table 1.

## 4.2 Query 2

*Query 2.* Show how many students there are with no sibling, with one sibling, with two siblings etc.

```
CREATE VIEW sibling_count AS
    WITH num_siblings_foreach_student AS (
    SELECT student.person_id,
    COALESCE(COUNT(sibling.student_id), 0) AS siblings
    FROM student
    LEFT JOIN sibling
    ON student.person_id = sibling.student_id
    GROUP BY student.person_id)

        SELECT COUNT(*) AS number_of_students, siblings
        FROM num_siblings_foreach_student
        GROUP BY siblings
        ORDER BY siblings;

-- example:
SELECT * FROM sibling_count;
```

| number_of_students | siblings |
|---|---|
| 18 | 0 |
| 2 | 1 |
| 3 | 2 |

Table 2: Result from query 2.

To solve the second query we created a separate table with columns for all students and their siblings. Obviously, not every student has a sibling therefor we will get a lot of rows with null values which can be problematic. A workaround we discovered was to use `COALESCE` which will count all our null values as zeroes instead. The student and sibling table were then merged together using the `LEFT JOIN` syntax and then counted through `SELECT COUNT(*)`. Small adjustments to our database also had to be made since it lacked students with more than one sibling in its current state.

## 4.3 Query 3

*Query 3.* List all instructors who has given more than a specific number of lessons during the current month.

```
CREATE VIEW instructor_lessons_per_month AS
    SELECT instructor_id, COUNT(*) AS lessons
    FROM lesson WHERE now() > start_time AND
    EXTRACT(MONTH FROM now()) = EXTRACT(MONTH FROM start_time)
```

```
    AND EXTRACT(YEAR FROM now()) = EXTRACT(YEAR FROM start_time)
    AND min_num_students <= (SELECT COUNT(*)
    FROM booking WHERE lesson_id = lesson.id)
    GROUP BY instructor_id
    ORDER BY COUNT(*) DESC;

SELECT instructor_id, lessons
FROM instructor_lessons_per_month
WHERE lessons > 1;
```

| instructor_id | lessons |
|---|---|
| 2 | 2 |

Table 3: Result from query 3.

We first glanced at our database to see if the data was sufficient to perform the query. The bare minimum of data required was at least two instructors that teaches a different amount of lessons during the current month. In this query we also had to take into consideration that a lesson will only occur if the amount of bookings exceed the minimum number of students threshold. The sql `now()` function helped us target the data for the current month and year.

## 4.3 Query 4

*Query 4.* List all ensembles held during the next week, sorted by music genre and weekday. For each ensemble tell whether it's full booked, has 1-2 seats left or has more seats left.

```
CREATE MATERIALIZED VIEW ensemble_lessons_next_week AS
    WITH number_of_students_per_lesson
    AS (
    SELECT booking.lesson_id, COUNT(*)
    AS number_of_students
    FROM booking, lesson, ensemble
    WHERE lesson.id = booking.lesson_id AND
    ensemble.lesson_id = lesson.id
    GROUP BY booking.lesson_id)

    SELECT to_char(start_time, 'Day') AS day, genre, start_time,
    CASE WHEN number_of_students = max_num_students
    THEN 'No seats left' WHEN number_of_students = max_num_students - 1
    THEN 'One seat left'
```

```
    WHEN number_of_students = max_num_students - 2
    THEN 'Two seats left' ELSE 'Many seats left'
    END AS availability

FROM lesson, ensemble
INNER JOIN number_of_students_per_lesson
ON ensemble.lesson_id = number_of_students_per_lesson.lesson_id
WHERE lesson.id = ensemble.lesson_id AND
EXTRACT(DOW FROM start_time) != '0' AND
EXTRACT(DOW FROM start_time) != '6' AND
date_trunc('week',now()) + interval '1 week' = date_trunc('week',start_time);

-- example
EXPLAIN ANALYZE(SELECT * FROM ensemble_lessons_next_week);
```

| day | genre | start_time | availability |
|---|---|---|---|
| Wednesday | Jazz | 2022-12-14 12:00:00 | One seat left |
| Thursday | Classic | 2022-12-15 14:00:00 | No seats left |
| Tuesday | Jazz | 2022-12-13 12:00:00 | Two seats left |

Table 4: Result from query 4.

In the specification for query 4 it was stated that it will be performed programmatically, and that the results will be displayed on Soundgood's web page. For this query we therefor decided to create a materialized view, instead of a normal one. Materialized views are generally faster and consume fewer resources than queries that retrieve data from base tables.

From the three tables `booking`, `lesson` and `ensemble` we derived all ensemble lessons and their student count. To write the logic for the availability column we used the `CASE` syntax, this helped us target each specific case for how many seats there were left for an ensemble. Since we were only interested in the the ensembles during weekdays, we had to remove all ensembles which were scheduled on a weekend. This was done by extracting the day from `start_time` and then exclude Saturdays and Sundays.

## 4.4 Higher grade task

*Query 5.* Create a historical database from the existing database, and also create SQL statements for copying data from your present database to the historical database. The historical database is only required to contain lessons, prices and students.

```
CREATE DATABASE soundgood_historical WITH TEMPLATE soundgood;
WITH info_for_extracting_lesson_price AS(
    SELECT DISTINCT lesson.id AS lesson_id, end_time,
    student_id, current_skill_level, CASE
        WHEN max_num_students = 1 THEN 'Individual'
        WHEN lesson.id != ensemble.lesson_id THEN 'Group'
        ELSE 'Ensemble'
    END AS lesson_type
    FROM lesson, booking, instrument, skill_level, ensemble
    WHERE lesson.id = booking.lesson_id AND
            booking.instrument_id = instrument.id AND
            skill_level.instrument_id = instrument.id AND
            end_time <= now()
    ORDER BY lesson_id
)

SELECT lesson_id, student_id, amount
INTO denormalized_lesson
FROM info_for_extracting_lesson_price AS info,
pricing_and_salary_scheme AS scheme
WHERE   scheme.lesson_type = info.lesson_type AND
        skill_level = current_skill_level AND
        scheme.id < 10;

DROP TABLE  classroom, person, person_info,
phone_number, pricing_and_salary_scheme,
sibling_discount, soundgood_music_staff, student,
application, contact_person, instructor, instrument,
instrument_for_rent, lesson, person_phone_number,
rental_period, sibling, skill_level,
available_time_slot, booking, ensemble;
```

| lesson_id | student_id | amount |
|---|---|---|
| 1 | 8 | 75 |
| 2 | 9 | 40 |
| 2 | 10 | 40 |
| 4 | 16 | 50 |
| 5 | 17 | 50 |

Table 5: Result from query 5.

One of the requirements for the higher grade task was to create SQL statements for copying data from our present database to the historical

database. This is done on the very first line of the query where we simply use our database `soundgood` as a template for the historic one through the input `CREATE DATABASE soundgood_historical WITH TEMPLATE soundgood`. In `info_for_extracting_lesson_price` we gather all the necessary data needed to derive the cost of a lesson for each student. In the last `SELECT` statement we create a new table where we match the lesson type and skill level to get the appropriate amount from our `pricing_and_salary_scheme`. As a final step we drop all tables except the one we just created, since they no longer serve any purpose.

## GitHub

Link to project repository.

# 5. Discussion

## 5.1 Query Plan

Part of the mandatory assignment was to analyze at least one of the queries using the command `EXPLAIN ANALYZE`, we chose to analyze query 3 which can be seen below.

```
1  QUERY PLAN
2  Sort  (cost=16.97..16.98 rows=1 width=12) (actual time=0.041..0.042 rows=1 loops=1)
3    Sort Key: (count(*)) DESC
4    Sort Method: quicksort  Memory: 25kB
5    ->  GroupAggregate  (cost=16.94..16.96 rows=1 width=12) (actual time=0.037..0.038 rows=1 loops=1)
6          Group Key: lesson.instructor_id
7          Filter: (count(*) > 1)
8          Rows Removed by Filter: 2
9          ->  Sort  (cost=16.94..16.94 rows=1 width=4) (actual time=0.034..0.035 rows=4 loops=1)
10               Sort Key: lesson.instructor_id
11               Sort Method: quicksort  Memory: 25kB
12               ->  Seq Scan on lesson  (cost=0.00..16.93 rows=1 width=4) (actual time=0.018..0.031 rows=4 loops=1)
13                     Filter: ((now() > start_time) AND (EXTRACT(month FROM now()) = EXTRACT(month FROM start_time))
14                      AND (EXTRACT(year FROM now()) = EXTRACT(year FROM start_time)) AND (min_num_students <= (SubPlan 1)))
15                     Rows Removed by Filter: 8
16                     SubPlan 1
17                       ->  Aggregate  (cost=1.28..1.29 rows=1 width=8) (actual time=0.004..0.004 rows=1 loops=4)
18                             ->  Seq Scan on booking  (cost=0.00..1.27 rows=2 width=0) (actual time=0.001..0.002 rows=1 loops=4)
19                                   Filter: (lesson_id = lesson.id)
20                                   Rows Removed by Filter: 21
21 Planning Time: 0.153 ms
22 Execution Time: 0.072 ms
```

Figure 1: Query plan for query 3.

If two arrows have the same indention it means they are sibling nodes. All child nodes have parent nodes which are located directly one indent to the left and up from its child. In this query plan there are no sibling nodes, however there are several pairs of parent and child nodes. The query plan is

read from the right most arrow, in this plan on line 18 where we perform a sequential scan on `booking`. Sequential scan means that the program reads all the rows from the table, in order. Next we move upwards and one indent to the left to line 17, which is the parent node of line 18. Aggregate represent `COUNT(*)` from our query. On the next line 12 we perform yet another sequential scan this time on `lesson` with a filter of logical operation which can be found on line 9 through 12 in query 3. After that we move to line 9, 5 and lastly 2 which is called the root node. All nodes has a certain, `cost`, a number of `rows` and a `width`. The `cost` is a measurement of execution time where the first value represent the estimated time before output starts and the second value is the time when the node has completed. The `rows` stand for the number of rows emitted by the node. Lastly, the `width` is the bytes occupied by each output row. It is important to note that the cost of an operation displayed in a child node is carried over to its parent node, meaning the parent owns the total cost of an operation. The operation with the highest cost is the sequential scan executed on line 12, i.e. 16.93 out of the total cost of 16.98 as seen by the root node on line 2. The cost is most likely due to many filters being applied at the same time. However, the cost cannot be reduced, since all filters are necessary in order for the data to be correct.

## 5.1 Denormalization

Controlling redundancy of data has thus far been a big part of this course. Unwanted, or redundant data can lead to duplication of effort, wasted storage space and the risk inconsistency when when working with multiple files representing the same data. The process is called normalization and aims to eliminate said problems. However, sometimes it necessary to use **controlled redundancy**, to improve the performance of the database as normalization tends to divide our data into multiple files, which slows down the read performance. The higher grade task is a strong contender for denormalization, since historic databases seldom update and therefor write performance becomes trivial which also vastly decrease the risk of data anomalies. Instead of storing the prices of a lesson in a separate table we created a new table with all the lessons, their corresponding price and attending students. This will, of course, lead to duplicate data since the same prices will appear for all lessons of the same type i.e. individual, ensemble, and group. To even further denomarlize the data, all lesson types were merged into one column with only their id to distinguish them apart. Generally, normalization is used in OLTP systems (online transactional processing) to update, insert and delete faster. While denormalization is used in OLAP (online analytical processing) systems, which emphasize on tasks such as search and analytics to optimize the read performance. Data integrity is better maintained

in a normalized system whereas it becomes much harder to maintain in a denormalized system. As previously mentioned, denormalization also consumes more disk space as it stores duplicate data while normalization on the contrary frees disk space at the cost of read performance.