Task 3, version 2

Datalagring, IV1351

Adrian Jonsson Sjödin
adriansj@kth.se

April 9, 2023

Contents

1 Introduction

The purpose of task 3 was to write OLAP queries towards the database created in task 2, and confirm that everything works as intended. The creation and reasoning behind the OLAP queries is what will be covered in this report.

2 Method

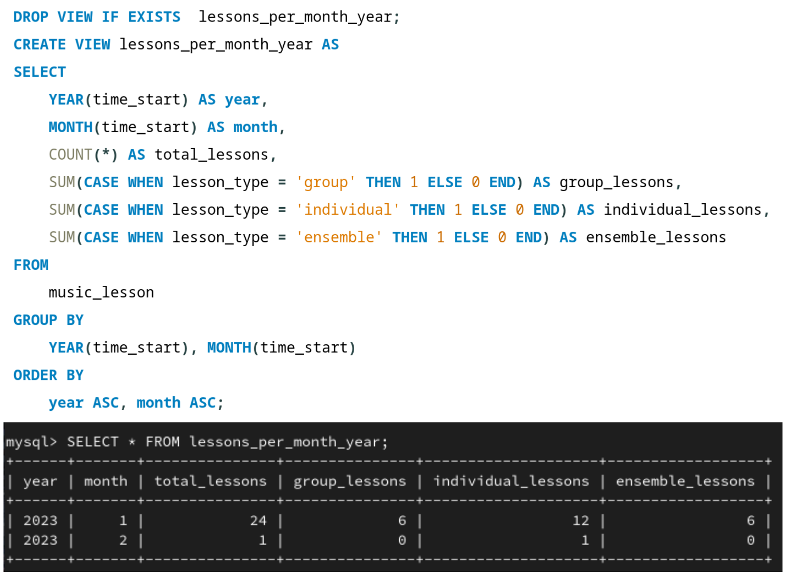
The DBMS used for the Sound Good Music School database is MySQL 8 and the queries was developed using a combination of MySQL Workbench and the CLI.

The process behind creating the queries was to first breaking them up into smaller parts and test that we got the expected output from them when they where run in the CLI, and then keep building upon them until we achieve the sought after query. For this we used MySQL Workbench since it is easier to write the code and modify it there than directly in the CLI.

The queries was then tested manually by retrieving the data in question from the right tables and manually look at the tables and see if the query result matched what we got when we looked directly at the tables and noted with pen and paper the data.

3 Result

The SQL scripts can be found here: GitHub.



```
DROP VIEW IF EXISTS  lessons_per_month_year;
CREATE VIEW lessons_per_month_year AS
SELECT
    YEAR(time_start) AS year,
    MONTH(time_start) AS month,
    COUNT(*) AS total_lessons,
    SUM(CASE WHEN lesson_type = 'group' THEN 1 ELSE 0 END) AS group_lessons,
    SUM(CASE WHEN lesson_type = 'individual' THEN 1 ELSE 0 END) AS individual_lessons,
    SUM(CASE WHEN lesson_type = 'ensemble' THEN 1 ELSE 0 END) AS ensemble_lessons
FROM
    music_lesson
GROUP BY
    YEAR(time_start), MONTH(time_start)
ORDER BY
    year ASC, month ASC;
```

```
mysql> SELECT * FROM lessons_per_month_year;
+------+-------+---------------+---------------+--------------------+------------------+
| year | month | total_lessons | group_lessons | individual_lessons | ensemble_lessons |
+------+-------+---------------+---------------+--------------------+------------------+
| 2023 |     1 |            24 |             6 |                 12 |                6 |
| 2023 |     2 |             1 |             0 |                  1 |                0 |
+------+-------+---------------+---------------+--------------------+------------------+
```

Figure 3.1: Total number of lessons per year and month for each lesson type

The query seen in fig. 3.1 is the one that gives how many lectures we have per month and year and how many of each type. The leftmost column in it tells us how many available lessons the school currently have by looking in the "music_lesson" table. The other three columns shows us how many of each type we have. It is the easiest query out of the four and uses aggregate functions in combination with flow control functions.

In fig. 3.2 we have a similar query but in this one the three columns that correspond to the different lesson types instead tells us how many of that type have been created. They will only be created when a new booking is created for a student to a lesson and thus tells us how many of the different lessons have at least one student signed up to it. This query uses aggregate functions and JOIN operators.

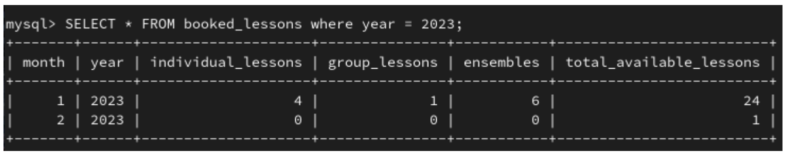
The query seen in fig. 3.3 is the one that displays how many siblings each student have that is also enrolled at the school. This query uses a correlated sub-query to achieve the sought after result.

The query seen in fig. 3.4 is the one that list all instructors that have more than a specific number of lessons during the current month. This and the next query were the ones who gave me the most trouble. This because the query makes use of sub-queries, JOIN clauses, aggregate functions, and conditional filtering, making it quite complex for me. Note that the column ”total_number_of_lessons” is counting just the lessons that have students assign to them. I did it this way because it felt the most relevant to know because if there’s no students taking a class it stands to reason that the instructor won’t be holding that class. If one want to get just the total number of lessons an instructor is assigned to then one can easily get that from the ”music_lesson” table.

Lastly we have the query seen in fig. 3.5 which is the one that lists all available ensemble spots for the coming week. Like the previous query it utilizes a combination JOIN clauses, aggregate functions, flow control functions, conditional filtering with the WHERE clause, and GROUP BY and ORDER BY clauses for grouping and sorting the results.

In the last figure 3.6 we see the result of running **EXPLAIN ANALYZE** on the query 3.3.

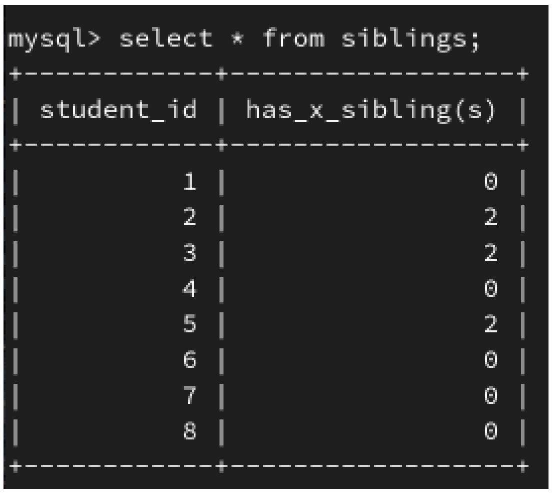
```
-- Run with: SELECT * FROM booked_lessons WHERE year = 2023;
DROP VIEW IF EXISTS  booked_lessons;
CREATE VIEW booked_lessons AS
SELECT
    MONTH(ml.time_start) AS month,
    YEAR(ml.time_start) AS year,
    COUNT(il.music_lesson_id) AS individual_lessons,
    COUNT(gl.music_lesson_id) AS group_lessons,
    COUNT(e.music_lesson_id) AS ensembles,
    COUNT(*) AS total_available_lessons
FROM
    music_lesson ml
LEFT JOIN
    ensemble e ON ml.id = e.music_lesson_id
LEFT JOIN
    group_lesson gl ON ml.id = gl.music_lesson_id
LEFT JOIN
    individual_lesson il ON ml.id = il.music_lesson_id
    GROUP BY
    MONTH(ml.time_start),
    YEAR(ml.time_start);
```



```
mysql> SELECT * FROM booked_lessons where year = 2023;
+-------+------+--------------------+---------------+-----------+-------------------------+
| month | year | individual_lessons | group_lessons | ensembles | total_available_lessons |
+-------+------+--------------------+---------------+-----------+-------------------------+
|     1 | 2023 |                  4 |             1 |         6 |                      24 |
|     2 | 2023 |                  0 |             0 |         0 |                       1 |
+-------+------+--------------------+---------------+-----------+-------------------------+
```

Figure 3.2: Total number of available lesson per year and month and how many lessons have students that have booked them for each lesson type

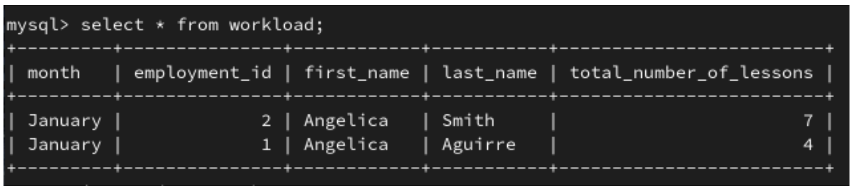
```sql
DROP VIEW IF EXISTS  siblings;
CREATE VIEW siblings AS
SELECT
  id AS 'student_id',
  --   parent_id,
  (
    SELECT
      COUNT(*) -1
    FROM
      student b
    WHERE
      b.parent_id = a.parent_id
  ) AS 'has_x_sibling(s)'
FROM
  student a
ORDER BY
  id;
```



```
mysql> select * from siblings;
+------------+------------------+
| student_id | has_x_sibling(s) |
+------------+------------------+
|          1 |                0 |
|          2 |                2 |
|          3 |                2 |
|          4 |                0 |
|          5 |                2 |
|          6 |                0 |
|          7 |                0 |
|          8 |                0 |
+------------+------------------+
```

Figure 3.3: The number of siblings for each student

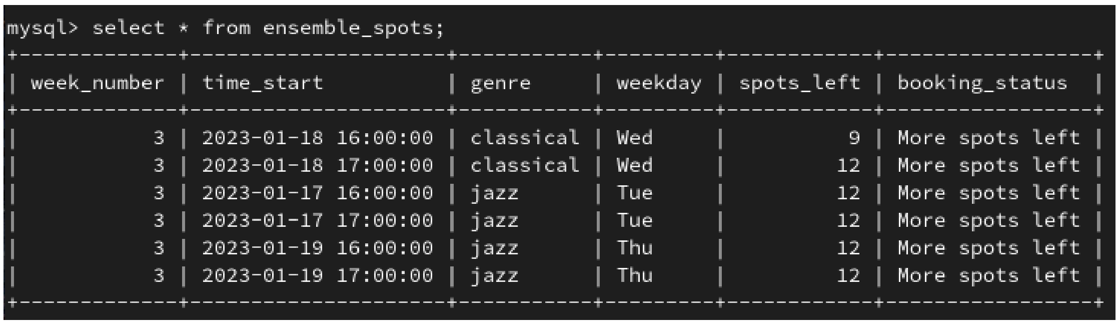
```sql
DROP VIEW IF EXISTS  workload;
CREATE VIEW workload AS
SELECT *
FROM (
  SELECT
    DATE_FORMAT(music_lesson.time_start, '%M') AS month,
    instructor.employment_id,
    person.first_name,
    person.last_name,
    COUNT(*) AS total_number_of_lessons
  FROM
    music_lesson
    INNER JOIN instructor ON instructor.id = music_lesson.instructor_id
    INNER JOIN person ON instructor.person_id = person.id
  WHERE
    (music_lesson.id IN (SELECT music_lesson_id FROM group_lesson) OR
    music_lesson.id IN (SELECT music_lesson_id FROM individual_lesson) OR
    music_lesson.id IN (SELECT music_lesson_id FROM ensemble)) AND
    -- Replace 1 with MONTH(NOW()) if you want to see lessons for the current month. However the database only contains data for January
    MONTH(music_lesson.time_start) = 1 AND
    YEAR(music_lesson.time_start) = YEAR(NOW())
  GROUP BY
    month,
    instructor.employment_id,
    person.first_name,
    person.last_name
) AS instructors
WHERE total_number_of_lessons > 1 -- CHANGE THIS TO CHANGE WHERE THE CUT OFF NUMBER IS
ORDER BY total_number_of_lessons DESC;
```



```
mysql> select * from workload;
+---------+---------------+------------+-----------+-------------------------+
| month   | employment_id | first_name | last_name | total_number_of_lessons |
+---------+---------------+------------+-----------+-------------------------+
| January |             2 | Angelica   | Smith     |                       7 |
| January |             1 | Angelica   | Aguirre   |                       4 |
+---------+---------------+------------+-----------+-------------------------+
```

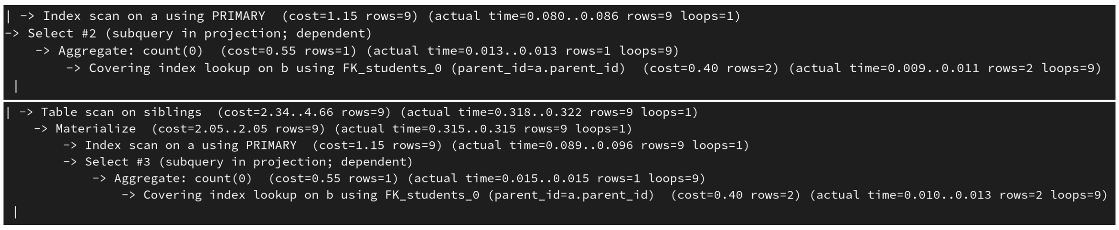
Figure 3.4: List of all instructors that have more than a specific nr of lesson during the current month

```sql
DROP VIEW IF EXISTS  ensemble_spots;
CREATE VIEW ensemble_spots AS
SELECT
    WEEK(ml.time_start, 1) AS week_number,
    ml.time_start,
    g.genre,
    DATE_FORMAT(ml.time_start, '%a') AS weekday,
    e.maximum_number_of_students - COUNT(b.id) AS spots_left,
    CASE
        WHEN e.maximum_number_of_students - COUNT(b.id) = 0 THEN 'Fully booked'
        WHEN e.maximum_number_of_students - COUNT(b.id) <= 2 THEN '1-2 spots left'
        ELSE 'More spots left'
    END AS booking_status
FROM
    music_lesson ml
JOIN
    ensemble e ON ml.id = e.music_lesson_id
JOIN
    genre g ON e.genre_id = g.id
LEFT JOIN
    booking b ON ml.id = b.music_lesson_id
WHERE -- CHANGE LINE BELOWW FOR IT TO ALWAYS CHECK THE NEXT WEEK BASED ON THE CURRENT WEEK. HARDCODED DATE SINCE I DON'T HAVE DATA EXCEPT FOR WEEK 3
WEEK(ml.time_start) = 2 + 1 -- CHANGE 2 TO WEEK(NOW()). Only have data for week 3
GROUP BY
    ml.id
ORDER BY
    g.genre ASC, weekday DESC;
```



```
mysql> select * from ensemble_spots;
+-------------+---------------------+-----------+---------+------------+-----------------+
| week_number | time_start          | genre     | weekday | spots_left | booking_status  |
+-------------+---------------------+-----------+---------+------------+-----------------+
|           3 | 2023-01-18 16:00:00 | classical | Wed     |          9 | More spots left |
|           3 | 2023-01-18 17:00:00 | classical | Wed     |         12 | More spots left |
|           3 | 2023-01-17 16:00:00 | jazz      | Tue     |         12 | More spots left |
|           3 | 2023-01-17 17:00:00 | jazz      | Tue     |         12 | More spots left |
|           3 | 2023-01-19 16:00:00 | jazz      | Thu     |         12 | More spots left |
|           3 | 2023-01-19 17:00:00 | jazz      | Thu     |         12 | More spots left |
+-------------+---------------------+-----------+---------+------------+-----------------+
```

Figure 3.5: List all ensembles held during the next week and how many spots are left, sorted by music genre and weekday



```
| -> Index scan on a using PRIMARY  (cost=1.15 rows=9) (actual time=0.080..0.086 rows=9 loops=1)
-> Select #2 (subquery in projection; dependent)
    -> Aggregate: count(0)  (cost=0.55 rows=1) (actual time=0.013..0.013 rows=1 loops=9)
        -> Covering index lookup on b using FK_students_0 (parent_id=a.parent_id)  (cost=0.40 rows=2) (actual time=0.009..0.011 rows=2 loops=9)
 |
```

```
| -> Table scan on siblings  (cost=2.34..4.66 rows=9) (actual time=0.318..0.322 rows=9 loops=1)
    -> Materialize  (cost=2.05..2.05 rows=9) (actual time=0.315..0.315 rows=9 loops=1)
        -> Index scan on a using PRIMARY  (cost=1.15 rows=9) (actual time=0.089..0.096 rows=9 loops=1)
        -> Select #3 (subquery in projection; dependent)
            -> Aggregate: count(0)  (cost=0.55 rows=1) (actual time=0.015..0.015 rows=1 loops=9)
                -> Covering index lookup on b using FK_students_0 (parent_id=a.parent_id)  (cost=0.40 rows=2) (actual time=0.010..0.013 rows=2 loops=9)
 |
```

Figure 3.6: On top is the query from fig. 3.3 without storing it in a view while the bottom is the same query when stored in a view

4 Discussion

As can be seen from the queries in the Result section, all queries are stored as views. This is to make it easier to run them at a later date. All queries except the query seen in fig. 3.2 are stored in full. This was done because only that query had the requirement of being able to vary what was shown based of a requirement. The rest of them didn't have that condition and thus it would be easier to store the whole query as a view.

The reason why I didn't use materialized view is simply because MySQL 8 (or earlier versions) doesn't support it. If I could have used them then I would most likely have used them on query 3.1, 3.3 and 3.4.

The first since the query is expected to be performed a few times per week and involves aggregation. Furthermore the rows in the table "music_lesson" will most likely be created in batches as the schedule is set and thus the view shouldn't need to be updated that often. This would allow for a faster reading of data whenever it is needed, with the drawback of having to update the table whenever new lessons are added. But most likely that wouldn't be too often making it worthwhile turning it into a materialized view.

While the query seen in fig. 3.3 is simple I would still turn it into a materialized view since there will most likely be a significant larger number of times that I will read the table, as opposed to updating it. After all it will only be required to be updated whenever a new student have been added to check if said student has a sibling already enrolled in the school.

Lastly the query seen in fig. 3.4 would probably benefit from being a materialized view as the query is expected to be executed daily and involves multiple joins and aggregation functions.

For the rest of the queries I think a view is the better choice since the columns containing the data we need to retrieve most likely will be updated often. For example the result from query 3.5 will change whenever another students booking for an ensemble lessons is accepted, or they cancel their booking. And since the result will be displayed on the web page it needs to be up to date.

The database didn't need any permanent changes to make the queries. I did however have to insert some extra data to test some of the queries since I've only inputted lesson for a few days in the database.

We see in 3.6 a slight difference between the two results. When run on the saved view, MySQL first materialize the view, which means it creates a temporary table containing the result of the view, and then scans that temporary table. On the other hand when **EXPLAIN ANALYZE** is run on the original query MySQL directly executes the query without materializing it first. The actual execution time and other metrics however are very similar between the two results, with only small minor differences. The most timeconsuming part of the query is as expected the correlated sub-query. This sub-query is executed once for each row returned by the outer query, as is indicated by the loops=9 in both cases.

One way of optimizing the query could be by trying to use a derived table with a **JOIN** instead of the correlated sub-query. That way one could avoid executing the sub-query multiple times which may lead to better performance. However I didn't manage to implement it different earlier and because of time constraints it will have to stay as a correlated sub-query unfortunately.