Task 3

Datalagring, IV1351

Adrian Jonsson Sjödin
adriansj@kth.se

January 10, 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 DbGate and the CLI. The queries was tested manually by retrieving the data in question from the right tables and see if the query result matched was as expected.

3 Result

The SQL scripts can be found here: GitHub.

```sql
DROP VIEW IF EXISTS  lessons_per_month_year;
CREATE VIEW lessons_per_month_year AS
SELECT
  YEAR(time_start) AS year,
  DATE_FORMAT(time_start, '%M') AS month,
  COUNT(*) AS 'total_nr_of_lessons',
  SUM(
    CASE
      WHEN lesson_type = 'individual' THEN 1
      ELSE 0
    END
  ) AS individual,
  SUM(
    CASE
      WHEN lesson_type = 'group' THEN 1
      ELSE 0
    END
  ) AS 'group',
  SUM(
    CASE
      WHEN lesson_type = 'ensemble' THEN 1
      ELSE 0
    END
  ) AS ensemble
FROM
  music_lessons
GROUP BY
  YEAR(time_start),
  DATE_FORMAT(time_start, '%M');
```

year	month	total_nr_of_lessons	individual	group	ensemble
2023	January	15	6	4	5
2023	February	4	1	1	2
2023	March	2	1	1	0

Figure 3.1: Total number of lessons per year and month 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
      students b
    WHERE
      b.parent_id = a.parent_id
  ) AS 'has_x_sibling(s)'
FROM
  students a
ORDER BY
  id;
```

```
+------------+------------------+
| student_id | has_x_sibling(s) |
+------------+------------------+
|          1 |                1 |
|          2 |                0 |
|          3 |                2 |
|          4 |                2 |
|          5 |                0 |
|          6 |                2 |
|          7 |                1 |
|          8 |                0 |
|          9 |                0 |
+------------+------------------+
```

Figure 3.2: The number of siblings for each student

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. It is the easiest query out of the four and uses aggregate functions in combination with flow control functions.

The query seen in fig. 3.2 is the one that displays how many siblings each student have that is also enrolled at the school. This was the query that took me the longest to write and has a correlated sub-query in the **SELECT** clause.

The third query is seen in fig. 3.3 and is the one that list all instructors that have more than a specific number of lessons during the current month. This query makes use of **INNER JOIN** to create a temporary table containing the data we wanted.

Lastly we have the query seen in fig. 3.4 which is the one that lists all available ensemble spots for the coming week. It utilizes a combination of flow control functions to display available spots, and **INNER JOIN** to get the data from the two tables where the keys are matching.

```sql
DROP VIEW IF EXISTS  workload;
CREATE VIEW workload AS
SELECT
    *
FROM(
    SELECT
      DATE_FORMAT(music_lessons.time_start, '%M') AS month,
      instructors.employment_id,
      persons.first_name,
      persons.last_name,
      Count(*) as total_number_of_lessons
    FROM
      music_lessons
      INNER JOIN instructors ON instructors.id = music_lessons.instructor_id
      INNER JOIN persons ON instructors.person_id = persons.id
    WHERE
      MONTH(music_lessons.time_start) = MONTH(NOW())
    GROUP BY
      month,
      employment_id,
      persons.first_name,
      persons.last_name
    ORDER BY
      Count(*) DESC
  ) AS instructors;


-- SELECT * FROM workload
-- WHERE
--   total_number_of_lessons > 6;
```

```
+---------+---------------+------------+-----------+------------------------+
| month   | employment_id | first_name | last_name | total_number_of_lessons |
+---------+---------------+------------+-----------+------------------------+
| January |             1 | Angelica   | Aguirre   |                      8 |
| January |             2 | Angelica   | Smith     |                      7 |
+---------+---------------+------------+-----------+------------------------+
```

Figure 3.3: 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
    ensemble.genre AS genre,
    WEEK(music_lessons.time_start) AS week,
    DATE_FORMAT(music_lessons.time_start, '%a') AS weekday,
    CASE
        WHEN music_lessons.numb_of_participants = ensemble.maximum_number_of_students THEN 'Full'
        WHEN (ensemble.maximum_number_of_students - music_lessons.numb_of_participants) = 1 THEN '1 spot left'
        WHEN (ensemble.maximum_number_of_students - music_lessons.numb_of_participants) = 2 THEN '2 spots left'
        ELSE '3 or more spots left'
    END AS spots_left
FROM ensemble
    INNER JOIN music_lessons ON music_lessons.id = ensemble.lesson_id
WHERE WEEK(music_lessons.time_start) = WEEK(NOW())+1 -- Ensembles are scheduled in week 4 so set = 4 for testing
GROUP BY ensemble.genre, music_lessons.numb_of_participants, music_lessons.time_start, ensemble.maximum_number_of_students
ORDER BY weekday DESC;
```

genre	week	weekday	spots_left
classical	4	Wed	3 or more spots left
classical	4	Thu	1 spot left
jazz	4	Sun	2 spots left
jazz	4	Mon	Full
jazz	4	Fri	3 or more spots left

Figure 3.4: List all ensembles held during the next week and how many spots are left, sorted by music genre and weekday

```
-> Table scan on <temporary>  (actual time=3.022..3.024 rows=3 loops=1)
  -> Aggregate using temporary table  (actual time=3.019..3.019 rows=3 loops=1)
      -> Table scan on music_lessons  (cost=2.35 rows=21) (actual time=0.171..0.202 rows=21 loops=1)
```

```
-> Table scan on lessons_per_month_year  (cost=2.50..2.50 rows=0) (actual time=0.293..0.295 rows=3 loops=1)
  -> Materialize  (cost=0.00..0.00 rows=0) (actual time=0.292..0.292 rows=3 loops=1)
      -> Table scan on <temporary>  (actual time=0.265..0.267 rows=3 loops=1)
          -> Aggregate using temporary table  (actual time=0.263..0.263 rows=3 loops=1)
              -> Table scan on music_lessons  (cost=2.35 rows=21) (actual time=0.053..0.071 rows=21 loops=1)
```

Figure 3.5: On top is the query from fig. 3.1 without storing it in a view while the bottom is the same query when stored in a view

```
| -> 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. 3.2 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. 3.1 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 seem to support it. If I could have used them then I would most likely have used them on query 3.1 and 3.2. The first since I'm assuming the external application will create all lessons for multiple weeks in advanced and thus not needing to update it very 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.

The same reasoning is behind why I would make query 3.2 into a materialized view. There will be a significant larger number of times I will read the table I get from that query as opposed to the number of times the materialized view would have to be updated.

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 3.4 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 make a temporary change by using DbGate to change the number of participants that had signed up for ensemble lessons larger. This so that I could test that query 3.4 worked as intended. It was the easy solution for testing purpose since it didn't require me to manually create more rows in the **bookings** table and match them with the corresponding music lesson (something that would be handled by the application that the administrative staff uses).

As mentioned in the Result section, query 3.2 uses a correlated sub-query. It is possible I could get around this with more time and use some kind of **JOIN** operator instead, but currently I couldn't find a better solution.

We see in fig. 3.5 that it took on average 0.202 ms to read all 21 rows in the **music_lesson** table and that the aggregation took 3.019ms for when the query was not stored as a view. And the outermost table scan took 3.024. Meaning the majority of the time was spent on the aggregation and the outermost table scan. Giving us a total of about 6.23 ms. Looking at the result for the view we see that the reading of rows in the **music_lesson** table was a bit faster and here taking 0.071 ms and the aggregation took

only 0.263 ms. The outer table scan was also faster, clocking in at 0.267 ms. However we have two more costs in the view at 0.292 ms and 0.295 ms, giving us a total of about 1.19 ms. This means that the view in this case is more than five times faster.

We don't get same result for the view and non view correlated sub-queries seen in fig 3.6 however. There the view takes approximately 0.985 ms and the non view takes approximately 0.302 ms, making the view more than three times slower in this case.