Project Report

Data Storage Paradigms, IV1351

December 2nd
2024

Project members:
Simon Lieb Fredriksson, simonlf@ug.kth.se
Sebastian Taavo Ek, sebte@ug.kth.se
Adalet Adiljan, adalat@ug.kth.se

Declaration:

By submitting this assignment, it is hereby declared that all group members listed above have contributed to the solution. It is also declared that all project members fully understand all parts of the final solution and can explain it upon request.

It is furthermore declared that the solution below is a contribution by the project members only and specifically that no part of the solution has been copied from any other source (except for lecture slides at the course IV1351), no part of the solution has been provided by someone not listed as a project member above, and no part of the solution has been generated by a system.

1 Introduction

This report treats the third task of the course project and involves the writing of specific SQL queries following the instructions on the canvas project page.

Once all the queries had been written, the next step was to add the EXPLAIN ANALYZE tag to one of the scripts and investigate the results, documenting them in this report.

Unlike in previous reports, our group has no aspirations of completing the higher-grade part of this exercise seeing as SQL is quite new to all of us and the basic task was difficult enough. However, we might try to pursue the higher-grade points for seminar 5 if time permits.

2 Literature Study

Unlike the previous two seminar tasks, this task did not come with the guidance of examinator Leif Lindbäck's YouTube videos, and instead we relied almost entirely on the SQL lecture given by Paris Carbone as well as the lengthy document of tips and tricks supplied on the project page. We'll discuss our findings from both here.

2.1 Paris Carbone's lecture on SQL

There were two video lectures given on the topic of SQL - one in which the lecturer investigated the connection between SQL and relational algebra more closely, and one in which the focus was more purely on SQL and how to use the language. For our assignment, the brunt of the information we needed to get started came from the latter of the two.

Carbone begins the lecture by explaining that SQL is a declarative language used for data definition and manipulation - and that it is largely based on the relational data model as well as set theory. The core semantics which connect SQL to the previous project task and the entity-relation model is that we call our entities/tables "relations", our attributes "columns", and our tuples "rows". This way we can easily visualize how to transition our language from the previously created physical-logical model into SQL terms. In addition to these tables, an SQL database can also contain views, which are virtual tables that are either simply aliases for pre-written queries, or materialized views which are base tables in and of themselves and update dynamically as the relevant relations in the query that generated it update as well.

We move on to discover that there are various types of queries; retrieval queries, data manipulation queries as well as aggregation queries. Paris shows clearly how to use powerful keywords such as INTERSECT, UNION and EXCEPT (assuming that we are operating on sets, which we do if we give the retrieval query the DISTINCT keyword) as well as how to structure the conditions for our retrievals in the "WHERE" section of the query.

We're also shown ways to accomplish schema modifications by employing the ALTER keyword - which we used when creating the database in order to assign primary keys or define foreign keys in the table - and lastly we're taught group aggregation keywords such as COUNT, SUM, MIN, MAX, AVG.

2.2 The tips and tricks leaflet

The leaflet contained much of the same information as the lecture given by Paris Carbone, but illustrated more concretely how a very large and verbose query could be constructed part-by-part. This was very useful seeing as many of the SQL queries we ended up constructing for the tasks of this seminar were substantially larger than we first anticipated and needed much trial and error to produce.

Some key bits of information, however, that seemed useful before starting the task was that we should avoid overusing OR conditions in JOIN clauses since they could po-

tentially lead to slow nested loops - and that Views are best used whenever we need to simplify complex queries. Just like the lectures, the leaflet also emphasized that set operators such as UNION are incredibly useful for combining result sets and appending rows rather than merging columns. Effective use of JOIN sub-queries were also encouraged.

Suffice to say, there was an abundance of preparational knowledge present in both the leaflet and the lectures - and though this was incredibly useful, it was also somewhat overwhelming for students who had little previous practise writing queries in SQL.

3 Method

As previously mentioned, one key difference in this task compared to the previous two is that we had to rely solely on the tips and tricks leaflet as well as a pair of video lectures. This made the task a little more difficult since we had no real "example" of what the end result should look like, and left room for misunderstandings that no doubt contributed a lot to any eventual mistakes in our end product.

3.1 Updating our old Database

To start off we first updated both our diagram and our SQL database, which we managed using the DBMS PostgreSQL and its terminal "psql", after the feedback we got from seminar 2. The first change involved how we defined foreign keys in the physical-logical model. Before the feedback, we had specified all foreign keys in the Astah diagram to be of the data type "INT GENERATED ALWAYS AS IDENTITY" - this was an automatic property assignment done by Astah when assigning the column to be a foreign key from another relation, since that was the custom datatype of that primary key in the original table. In our actual SQL-code, we had already reassigned this to be a regular INT - but we updated the text in the diagram nonetheless.

After this, we fixed an issue with how we tracked the instrument rentals. Leif correctly pointed out in the feedback to our seminar 2 report that the rental time was tracked directly in each tuple of the instrument table, even though this would inevitably mean that the information history of rentals for a particular instrument would be deleted and re-assigned values each time a new rental started. The fix we made to this was to create a new table called "student_instrument_rental" which serves as a cross-reference table between student tuples and instrument tuples, and stores all the rental information neatly in a way that persists between rentals.

3.2 Creating the Queries

Once the database had been updated to reflect these changes, as well as re-populated with a modest amount of dummy data, the next step was to focus on creating the queries for analyzing the contents of the database in accordance to each task description. We wrote our queries in Visual Studio Code, then imported them to the database using the \i command in psql.

1. **Task 1 - Lessons per month**
   Here, the retrieval query was supposed to return the total number of lessons given per month, including the breakdown for solo, group, and ensemble lessons.

2. **Task 2 - Students with siblings**
   Next, the query had to yield a count of how many students have 0, 1, 2, or more siblings, without using a sibling count column in the student table - since that would have been impractical to maintain.

3. **Task 3 - Instructors with many lessons this month**
   The third query had to return a list of instructors who have given more than a certain number of lessons in the current month, sorted by lesson count. We set the threshold for the search manually in the query to "2".

4. **Task 4 - Ensembles next week**
   Lastly, the final query had to fetch all ensembles for the next week, sorted by genre and weekday, as well as display the number of available seats in the class.

3.3 Testing the Queries

To verify that the queries were working as they should, we ran them on our database and compared the outputs to the actual contents of each table, and manually verified the veracity of the output. To run the queries, divided up into .sql script files, we used the \i command in our PostreSQL terminal followed by the directory of each script. For example to run task 1 you would do \i *"Path to task1.sql"*;

After checking wether or not the output matched what we could find in our dummy data, we could then adjust the queries according to what was missing or incorrect - or simply start over if the query attempt was a lost cause. Iteratively, we managed to produce a script for each task, and we believe that they are sufficient for their purposes. This was quite challenging, and required much googling and discussion to get right - and it was only a slight consolation to hear Paris Carbone mention in his lecture that even skilled professionals have to search extensively and use online tools to construct SQL queries on the daily. Although we must probably concede that those queries are substantially more complex than the ones produced in this report.

4 Result

The Git Repository: https://github.com/SimonLieb/IV1351-Soundgood

The Github repository contains the files corresponding to each sub-task for this week's assignment. They are named as follows:

1. First: task1.sql
2. Second: task2.sql
3. Third: task3.sql

4. Fourth: task4.sql

4.1 Task 1

The query for task 1 retrieves the breakdown of the number of lessons by type (solo, group and ensemble) for each month and year. For example, the total number of ensemble lessons is calculated by summarizing the number of rows found in the ensemble table that occur on any given month. 'EXTRACT' is used to fetch the month and year from the TIMESTAMP WITH TIME ZONE data type, and 'COUNT(*) FILTER' is used to separately calculate student counts and categorize them as solo or group lessons: single student in the "min" and "max" student columns cause for the tuple to be considered a solo lesson, while more than one student in either of those two columns causes for the tuple to be considered a group lesson.

```sql
WITH LessonCounts AS (
    SELECT
        EXTRACT(YEAR FROM start_time) AS Year,
        EXTRACT(MONTH FROM start_time) AS Month,
        COUNT(*) FILTER (WHERE max_num_of_students = 1 AND min_num_of_students = 1) AS SoloLessons,
        COUNT(*) FILTER (WHERE max_num_of_students > 1 OR min_num_of_students > 1) AS GroupLessons
    FROM lesson
    GROUP BY EXTRACT(YEAR FROM start_time), EXTRACT(MONTH FROM start_time)
),
EnsembleCounts AS (
    SELECT
        EXTRACT(YEAR FROM start_time) AS Year,
        EXTRACT(MONTH FROM start_time) AS Month,
        COUNT(*) AS EnsembleLessons
    FROM ensemble
    GROUP BY EXTRACT(YEAR FROM start_time), EXTRACT(MONTH FROM start_time)
)
SELECT
    l.Year AS Year,
    l.Month AS Month,
    (l.SoloLessons + l.GroupLessons + e.EnsembleLessons) AS TotalLessons,
    l.SoloLessons AS SoloLessons,
    l.GroupLessons AS GroupLessons,
    e.EnsembleLessons AS EnsembleLessons
FROM LessonCounts l
INNER JOIN EnsembleCounts e
    ON l.Year = e.Year AND l.Month = e.Month
ORDER BY Year, Month;
```

Data Output | Messages | Notifications

	year numeric	month numeric	totallessons bigint	sololessons bigint	grouplessons bigint	ensemblelessons bigint
1	2024	10	7	2	2	3
2	2024	11	5	1	2	2
3	2024	12	20	4	6	10

Figure 1: Query and output for task 1.

4.1.1 Explain Analyze

We chose to implement the EXPLAIN ANALYZE command with this query, and our analysis of its outputs is included in the Discussion chapter.

Query Query History

```
1  EXPLAIN ANALYZE
2  WITH LessonCounts AS (
3      SELECT
4          EXTRACT(YEAR FROM start_time) AS Year,
5          EXTRACT(MONTH FROM start_time) AS Month,
6          COUNT(*) FILTER (WHERE max_num_of_students = 1 AND min_num_of_students = 1) AS SoloLessons,
7          COUNT(*) FILTER (WHERE max_num_of_students > 1 OR min_num_of_students > 1) AS GroupLessons
8      FROM lesson
9      GROUP BY EXTRACT(YEAR FROM start_time), EXTRACT(MONTH FROM start_time)
10 ),
11 EnsembleCounts AS (
12     SELECT
13         EXTRACT(YEAR FROM start_time) AS Year,
14         EXTRACT(MONTH FROM start_time) AS Month,
15         COUNT(*) AS EnsembleLessons
16     FROM ensemble
17     GROUP BY EXTRACT(YEAR FROM start_time), EXTRACT(MONTH FROM start_time)
18 )
19 SELECT
20     l.Year AS Year,
21     l.Month AS Month,
22     (l.SoloLessons + l.GroupLessons + e.EnsembleLessons) AS TotalLessons,
```

Data Output Messages Notifications

	QUERY PLAN text
1	Sort (cost=49.34..49.34 rows=1 width=96) (actual time=1.638..1.641 rows=3 loops=1)
2	Sort Key: (EXTRACT(year FROM lesson.start_time)), (EXTRACT(month FROM lesson.start_time))
3	Sort Method: quicksort Memory: 25kB
4	-> Hash Join (cost=43.27..49.33 rows=1 width=96) (actual time=1.409..1.414 rows=3 loops=1)
5	Hash Cond: (((EXTRACT(year FROM ensemble.start_time)) = (EXTRACT(year FROM lesson.start_time))) AND ((EXTRACT(month FROM ensemble.start_time)) = (EXTRACT(month FROM lesson.start_ti…
6	-> HashAggregate (cost=16.07..19.07 rows=200 width=72) (actual time=1.135..1.137 rows=3 loops=1)
7	Group Key: EXTRACT(year FROM ensemble.start_time), EXTRACT(month FROM ensemble.start_time)
8	Batches: 1 Memory Usage: 40kB
9	-> Seq Scan on ensemble (cost=0.00..14.05 rows=270 width=64) (actual time=1.108..1.120 rows=15 loops=1)
10	-> Hash (cost=24.20..24.20 rows=200 width=80) (actual time=0.240..0.241 rows=3 loops=1)
11	Buckets: 1024 Batches: 1 Memory Usage: 9kB
12	-> HashAggregate (cost=21.20..24.20 rows=200 width=80) (actual time=0.234..0.236 rows=3 loops=1)
13	Group Key: EXTRACT(year FROM lesson.start_time), EXTRACT(month FROM lesson.start_time)
14	Batches: 1 Memory Usage: 40kB
15	-> Seq Scan on lesson (cost=0.00..14.80 rows=320 width=72) (actual time=0.167..0.218 rows=17 loops=1)
16	Planning Time: 2.598 ms
17	Execution Time: 1.756 ms

Figure 2: Results of the execution time for task 1.

4.2 Task 2

This part of the task involved retrieving the number of students who have no siblings, one sibling, two, and so on. In the LEFT JOIN first sub-query parenthesis we create a temporary table called ”sibling count” containing each student id in the first column and the number of siblings they have in the second. We then join that table on the student table (fetched in the first SELECT query). The UNION ALL operation between two select statements on ”student id 1” and ”student id 2” should ensure that relationships from both directions in the sibling table are considered, and we end the query by aggregating the data using GROUP BY.

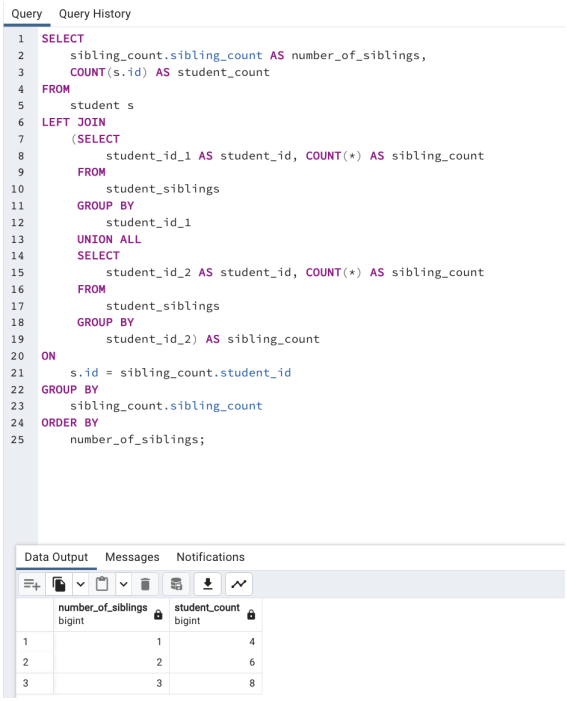

Figure 3: Resulting query for task 2.

4.3 Task 3

The purpose of the third query is to list each instructor's total number of given lessons during the current month - should the total exceed an arbitrary threshold. We accomplish this by using the LEFT JOIN keyword on SELECTs of all tuples from the "ensemble" and "lesson" relations which meet the criterion of occurring on the current month as well as having the instructor's id in its instructor foreign key column. Lastly, we order the data in a descending fashion according to the "total lesson count" alias which we define in the first SELECT statement.

Query Query History

```
SELECT
    i.id AS instructor_id,
    i.name AS instructor_name,
    l.lesson_count + e.ensemble_count AS total_lesson_count
FROM
    instructor i
LEFT JOIN
    (SELECT
         instructor_id,
         COUNT(*) AS lesson_count
     FROM
         lesson
     WHERE
         DATE_TRUNC('month', start_time) = DATE_TRUNC('month', CURRENT_DATE)
     GROUP BY
         instructor_id) l ON i.id = l.instructor_id
LEFT JOIN
    (SELECT
         instructor_id,
         COUNT(*) AS ensemble_count
     FROM
         ensemble
     WHERE
         DATE_TRUNC('month', start_time) = DATE_TRUNC('month', CURRENT_DATE)
     GROUP BY
         instructor_id) e ON i.id = e.instructor_id
WHERE
    l.lesson_count + e.ensemble_count > 2  -- threshold is 2 here, we change as needed
ORDER BY
    total_lesson_count DESC;
```

Data Output Messages Notifications

	instructor_id integer	instructor_name character varying (500)	total_lesson_count bigint
1	1	John Doe	4
2	2	Jane Smith	4
3	3	Alice Johnson	3
4	4	Bob Brown	3
5	5	Charlie Black	3

Figure 4: Resulting query for task 3.

4.4 Task 4

Finally, the last query required us to fetch information about all lessons in the upcoming week and list, per day of the week, what genre of lesson is given for that day - as well as the number of empty seats.

For this task we thought it would be pertinent to use a ”View”, since the task says its something that will be called programmatically and quite often. By converting the query into a view we then make it easier for the developer/user to fetch it and understand what the query does in the code, since we can put a clear name on the view to describe its function.

The query selects three columns for the output table; day of the week, genre of the music being produced by the ensemble, and the number of free seats. We then process information in the ”ensemble” relation as well as the ”student ensemble” relation, and find out how many free seats are available for any given lesson by subtracting the number of students assigned to a lesson in the lookup table ”student ensemble” from the ”max number of students” column data in the ”ensemble” tuple.

Importantly, the query also has to filter the results by time - which we accomplish by using the WHERE condition. We let the condition be ”start time ¿ current date” as well as ”start time ¡ current date + INTERVAL ’1 week’”. Using the ”INTERVAL”

keyword was very new to us, but it seems to appropriately calculate a time between now and a week from now. This way we make sure that the timestamp of each considered lesson falls within the desired interval.

Query Query History

```
1
2  CREATE OR REPLACE VIEW next_week_ensembles AS
3  SELECT
4      TO_CHAR(start_time, 'Day') AS day_of_week,
5      genre,
6      (max_num_of_students - COUNT(student_ensemble.student_id)) AS free_seats
7  FROM
8      ensemble
9  LEFT JOIN
10     student_ensemble ON ensemble.id = student_ensemble.ensemble_id
11 WHERE
12     start_time >= CURRENT_DATE AND start_time < CURRENT_DATE + INTERVAL '1 week'
13 GROUP BY
14     day_of_week, genre, max_num_of_students
15 ORDER BY
16     EXTRACT(DOW FROM MIN(start_time)), genre;
17
18 SELECT * FROM next_week_ensembles;
```

Data Output Messages Notifications

	day_of_week text	genre character varying (50)	free_seats bigint
1	Monday	Classical	2
2	Monday	Rock	4
3	Tuesday	Jazz	5
4	Tuesday	Rock	5
5	Wednesday	Classical	3
6	Wednesday	Hip-hop	3
7	Thursday	Jazz	6
8	Thursday	K-Pop	4
9	Friday	Hip-hop	4
10	Friday	K-Pop	5

Figure 5: Resulting query for task 4.

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?
- Answer here.

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?
- No. We changed the database structure from seminar 2 in accordance with Leif Lindbäck's feedback to our report, and only once those changes had been made did we continue with the SQL-work. So while the database did see some changes, those changes did not occur in order to simplify the writing of our queries. In hindsight, it could have been beneficial to split up the "lesson" relation into "solo lesson" and "group lesson", however, to avoid having to write WHERE statements that evaluate the lesson type each time it comes into question.

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
- We are not entirely clear on the concept, but we believe that there are no correlated subqueries in any of our task solutions. In task 1, both CTEs "LessonCounts" and "EnsembleCounts" should be independent of the outer query. In task 2, the subqueries inside the LEFT JOIN don't look to be correlated either - instead, they should be individual aggregations grouped by column data from the "student siblings" table. In task 3, "lesson count" and "ensemble count" also look to be independent and not using any values from an outer query. Task 4 seems to be clean as well.

We could have entirely misunderstood the meaning, but as far as we understand we think our code has managed to avoid this pitfall.

Are there unnecessarily long and complicated queries? Are you for example using a UNION clause where it's not required?
- We think that there is one case in task 2 where we might have used "UNION ALL" where its not strictly necessary. The UNION ALL operation ensures that both directions of the siblingship between students are considered in the table "student siblings", but if the dummy data had already accounted for that and captured each siblingship as a pair of tuples then we could have probably avoided using a UNION operation and instead simply selected, counted and grouped by. For example, if student 1 is a sibling of student 2, we might have already inserted a tuple automatically where student 2 is a sibling of student 1.

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.
- As seen in the image of the EXPLAIN ANALYZE output in figure 2 of the results

section, the execution time was 1.7 ms and the planning time was less than double at 2.6 ms. Its hard to tell whether or not this is good performance or if the task could have been sped up, and it is not clear to us either how much a larger dataset in the concerned relations would effect the speed of the operation. The brunt of the time seems to be spent on the "Hash Join" part of the execution, which we believe to be the step in the query where the two aggregated data sets "Lesson counts" and "Ensemble counts" are joined.

Furthermore, a decent amount of time is spent sequentially scanning, and a brief bit of additional research shows us that sequential scanning is a quick and straight forward operation that works well for small tables, but could be less efficient for larger tables. [1] If the time complexity here is O(n), then this would mean that as the lesson and ensemble relations grow - this section of the query execution would almost certainly grow too.

6 Comments About the Course

This section is optional, but please at least write approximately how much time you spent on the assignment - We are unsure how much time we spent on the assignment, but it should be around the twenty hour mark collectively. The assignment was definitely less time consuming than the previous one where we produced the physical-logical model, but it was also significantly more time consuming since we were all quite new to SQL and there was a lot of syntax and key words that we had to not just memorize but actually understand.

7 References

References

[1] PGMustard. (2024). *Sequential Scan Explanation*. Available at: `https://www.pgmustard.com/docs/explain/seq-scan`. Accessed: Dec. 2, 2024.