Project Report
Task 3
Data Storage Paradigms, IV1351

Wilhelm Nordgren
cwfno@kth.se

January 2023

Contents

1 Introduction

1.1 Task description

In the third task it's time to start using the database created in the previous tasks. This is done by creating queries that extract information from the database. There are four main queries which should be created.

1. Query 1
   (a) Requirement: The first query should show the numbers of lessons giver per month during a specified year. The lessons should be shown grouped by type of lessons eg. group and individual lessons. The query should also show the total number of lessons added together, preferably shown on the same row for the particular month.
   (b) Test: Because I've myself inserted the data into the database I could manually check that the correct figures are obtained when using the query.
2. Query 2
   (a) Requirement: Query 2 should show how many students there are with no, one or two siblings.
   (b) Test: Since I only have 10 students in the database where five of them has zero siblings, two has one sibling and three has two siblings it's easy to manually check that the query works.
3. Query 3
   (a) Requirement: The third query should list all instructors who has giver more than a specific number of lessons during the current month.
   (b) Test: To verify that the correct count of lessons are achieved I manually counted the lessons that the instructors had during the current month.
4. Query 4
   (a) Requirement: The last query should list ensembles held during the next week, sorted by music genre and weekday. It should also say whether the ensemble is fully booked or how many seats it has left.
   (b) Test: Manually checking again. In general the database has just a few ensembles each week which makes it quite simple to check.

2 Literature Study

To prepare for this task I've been reading the tips and trickstask3.pdf document and read in the postgres documentation to learn about how to write the queries.

3 Method

I've used the pgAdmin 4 DBMS for this task. To verify that the queries perform as intended I've tried to manually check with the database. The database is big but still possible to check on some items.

1. Query 1

   (a) Method: To solve this task I approached the problem in an agile way creating the required data incrementally with views. With this approach I could create views for each lesson type showing the number of lessons given per month. This was done for ensembles, group- and individual- lessons. With a forth view I summed them all together for a total number of lessons given per month. Lastly I could put it all together in a view which completes the query. In that query the user can specify what year data is requested from.

2. Query 2

   (a) Method: This is a quite straight forward query to make. Since every sibling is shown by a relation between the student and sibling table I just join these two tables and count the number of relations each student has there. In a separate view I then count how many students there are with each number of relations and shows this in a table.

3. Query 3

   (a) Method: To get this query working I first did a view where I list all the lessons that the instructors have during the full year. For each lessons that an instructor has the instructor id and month gets an entry in the list. With this list I could then use another view to count how many lessons the instructors have during a specified month.

4. Query 4

   (a) Method: First I needed to extract the ensemble id's and other data for the ensembles held next week. With those id's I could find relations to student bookings that's in the database. By counting all booking relations that an ensemble has and compare it to the maximum number of students for that ensemble I can extract the number of seats available.

4 Result

`https://github.com/WilleNord/KTH_work.git` In the github link there's a dump of my database and queries. Below I will walk through the results and code for each query.

1. Query 1

   (a) Result: For each lesson type the count of lessons per month was created with a view as in Figure 1. Every lessons is related to a time slot with a timestamp variable. With a LEFT JOIN all ensembles with related time stamps are join and with the date_trunc function the month is extracted. A count of number of lessons per month is then performed and these numbers are grouped and ordered by month. This is done for the three lessons types.

Figure 1: Ensembles per month

```
SELECT date_trunc('month'::text, ts.datetime) AS month,
    count(gl.id) AS "Group lessons/month"
   FROM group_lesson gl
     LEFT JOIN time_slot ts ON ts.id = gl.time_slot_id
  GROUP BY (date_trunc('month'::text, ts.datetime))
  ORDER BY (date_trunc('month'::text, ts.datetime));
```

Data Output | Messages | Notifications

	month timestamp without time zone	Group lessons/month bigint
1	2022-12-01 00:00:00	6
2	2023-01-01 00:00:00	23
3	2023-02-01 00:00:00	22
4	2023-03-01 00:00:00	25
5	2023-04-01 00:00:00	15
6	2023-05-01 00:00:00	19
7	2023-06-01 00:00:00	20
8	2023-07-01 00:00:00	24
9	2023-08-01 00:00:00	15
10	2023-09-01 00:00:00	28
11	2023-10-01 00:00:00	27
12	2023-11-01 00:00:00	17
13	2023-12-01 00:00:00	22

To get the total number of lessons per month a total count was required. This was done by first joining all the lessons counts into a single view and then using a view to sum all the lesson types for each month. To cope with null values for months not containing a certain type of lesson the COALESCE function was used. In figure 2 the total numbers of lessons per month code is shown.

Figure 2: Total number of lessons per month

```
SELECT all_lessons.month,
    SUM(COALESCE(all_lessons."Ensembles/month",0)
        + COALESCE(all_lessons."Group lessons/month",0)
        + COALESCE(all_lessons."Individual lessons/month",0))
        AS total
  FROM all_lessons
  GROUP BY all_lessons.month
  ORDER BY all_lessons.month;
```

Data Output | Messages | Notifications

	month timestamp without time zone	total numeric
1	2022-12-01 00:00:00	8
2	2023-01-01 00:00:00	46
3	2023-02-01 00:00:00	40
4	2023-03-01 00:00:00	52
5	2023-04-01 00:00:00	31
6	2023-05-01 00:00:00	30
7	2023-06-01 00:00:00	37
8	2023-07-01 00:00:00	40
9	2023-08-01 00:00:00	32
10	2023-09-01 00:00:00	41
11	2023-10-01 00:00:00	45
12	2023-11-01 00:00:00	28
13	2023-12-01 00:00:00	35

In a last view the data is joined together and shown in a single table. In the code the user can specify for which year the user wants data from. Figure 3 shows the result.

Figure 3: Number of lessons/month for a year

```
SELECT total_month.month,
   e_month."Ensembles/month",
   gl_month."Group lessons/month",
   il_month."Individual lessons/month",
   total_month.total
  FROM total_month
    FULL JOIN il_month ON il_month.month = total_month.month
    FULL JOIN gl_month ON gl_month.month = total_month.month
    FULL JOIN e_month ON e_month.month = total_month.month
 WHERE EXTRACT(year FROM total_month.month) = 2023::numeric;
```

Data Output Messages Notifications

	month timestamp without time zone	Ensembles/month bigint	Group lessons/month bigint	Individual lessons/month bigint	total numeric
1	2023-01-01 00:00:00	10	23	13	46
2	2023-02-01 00:00:00	7	22	11	40
3	2023-03-01 00:00:00	11	25	16	52
4	2023-04-01 00:00:00	10	15	6	31
5	2023-05-01 00:00:00	4	19	7	30
6	2023-06-01 00:00:00	9	20	8	37
7	2023-07-01 00:00:00	12	24	4	40
8	2023-08-01 00:00:00	8	15	9	32
9	2023-09-01 00:00:00	8	28	5	41
10	2023-10-01 00:00:00	8	27	10	45
11	2023-11-01 00:00:00	6	17	5	28
12	2023-12-01 00:00:00	7	22	6	35

2. Query 2

   (a) Result: The first view that counts how many sibling relations a student has is shown in figure 4. The count of these relations are then counted and show in a separate view which completes the query. That's shown in figure 5.

Figure 4: Number of sibling relations

```
SELECT count(sibling.sibling_id) AS siblings
  FROM student
    LEFT JOIN sibling ON sibling.sibling_id = student.person_id
 GROUP BY student.person_id;
```

Data Output Messages Notifications

	siblings bigint
1	1
2	2
3	0
4	0
5	1
6	0
7	0
8	0
9	2
10	2

Figure 5:

```
1  SELECT number_of_siblings.siblings AS "Number of siblings",
2     count(number_of_siblings.siblings) AS count
3    FROM number_of_siblings
4   GROUP BY number_of_siblings.siblings
5   ORDER BY number_of_siblings.siblings;
```

Data Output Messages Notifications

	Number of siblings bigint	count bigint
1	0	5
2	1	2
3	2	3

3. Query 3

   (a) Result: In the first view all the lessons that the instructors have is extracted. Because it's just the count of lessons per month that is requested only the instructor id and the month is detached. This is done by first doing a LEFT JOIN time slot person_id and the instructor.person_id. With that all the time slots that corresponds to an instructor is extracted. The lessons are related to the time slots with id's and these can be extracted with FULL JOIN's. The code for this is shown in Figure 6.

Figure 6:

```
1  SELECT instructor.person_id,
2     EXTRACT(month FROM ts.datetime) AS month
3    FROM instructor
4      LEFT JOIN time_slot ts ON ts.person_id = instructor.person_id
5      FULL JOIN individual_lesson il ON il.time_slot_id = ts.id
6      FULL JOIN group_lesson gl ON gl.time_slot_id = ts.id
7      FULL JOIN ensemble e ON e.time_slot_id = ts.id;
```

The second step is to count the number of lessons each instructor has for the current month and sorting for the specified number of lessons criteria. By using the EXTRACT and CURRENT_DATE functions the current month is extracted. With the HAVING clause the criteria can be added. This is shown in figure 7.

Figure 7:

```
1  SELECT instructor_lesson_month.person_id, count(instructor_lesson_month.month)
2  AS lessons
3  FROM instructor_lesson_month
4  WHERE instructor_lesson_month.month = EXTRACT(month FROM CURRENT_DATE)
5  GROUP BY instructor_lesson_month.person_id
6  HAVING count(instructor_lesson_month.month) > 5
7  ORDER BY lessons;
```

Data Output | Messages | Notifications

	person_id integer	lessons bigint
1	1	7
2	5	9
3	3	10
4	4	11
5	2	13

4. Query 4

   (a) Result: I first joined together the time slot and ensemble table on their respective time_slot id's. I could then sort for ensembles planned for next week with the EXTRACT and CURRENT_TIMESTAMP functions. I also extracted the weekday, genre, max and min students for each ensemble. This is shown in figure 8.

Figure 8:

```
1  SELECT ensemble.id,
2     ensemble.genre,
3     EXTRACT(dow FROM ts.datetime) AS weekday,
4     ensemble.min_students,
5     ensemble.max_students
6    FROM ensemble
7      JOIN time_slot ts ON ensemble.time_slot_id = ts.id
8   WHERE EXTRACT(week FROM ts.datetime) =
9   (( SELECT EXTRACT(week FROM CURRENT_TIMESTAMP) + 1::numeric));
```

Data Output | Messages | Notifications

	id [PK] integer	genre character var	weekday numeric	min_students integer	max_students integer
1	50	Jazz	4	4	8
2	54	Jazz	5	4	8

Then I needed to count how many booking relations each ensemble had. This corresponds to how many seats that are already booked in that particular lesson. This is done with the view shown in figure 9.

Figure 9:

```
1  SELECT nwe.id,
2      count(booking_ensemble.ensemble_id)
3      FROM booking_ensemble
4      JOIN next_week_ensembles nwe
5      ON nwe.id = booking_ensemble.ensemble_id
6      GROUP BY nwe.id
```

Data Output | Messages | Notifications

	id integer	count bigint
1	50	7
2	54	1

Lastly I use a CASE-statement in a separate view to show if an ensemble is fully booking or how many seats there are left. By comparing the seat count for each ensemble id to the maximum number of students for that ensemble the number of available seats can be shown to the user. This is shown in figure 10.

Figure 10:

```
1  SELECT  next_week_ensembles.id,
2          next_week_ensembles.genre,
3          next_week_ensembles.weekday,
4  CASE
5      WHEN
6          (SELECT seat_count.count
7           FROM seat_count
8           WHERE seat_count.id = next_week_ensembles.id)
9           < next_week_ensembles.max_students
10         THEN next_week_ensembles.max_students - (SELECT seat_count.count
11          FROM seat_count
12          WHERE seat_count.id = next_week_ensembles.id)
13         ELSE 0
14         END "seats available"
15 FROM next_week_ensembles
16 ORDER BY next_week_ensembles.genre, next_week_ensembles.weekday
```

Data Output | Messages | Notifications

	id integer	genre character varying (100)	weekday numeric	seats available bigint
1	50	Jazz	4	1
2	54	Jazz	5	7

5 Discussion

5.1 EXPLAIN

Looking at the query plan from figure 11 we can analyze some of the output rows. Since the DBMS pgadmin doesn't indent the rows it makes the interpretation a little bit harder but looking at the cost variable it's possible to figure out in which order things are executed. The query starts at row

22 with a sequential scan on the time slots, this takes 15.68 time units. The query then hashes and then does a hash right join with the result from the sequential scan on group_lesson made on row 20. The scan on row 20 has no delay in starting it's execution but the hash right join on row 18 has to wait until the first scan from row 22 is ready. The query plan then continues to do the same with ensembles and individual lessons.

It can also be seen that the query plan uses an index scan to left join the time slot person_id with the person_id. Since index scan is most favorable when only a small portion of a table is used it's probably the best choice here. The single most costly operation done is found on row 9 which is a nested loop join, the cost in time units is 17,7. Even though nested loop's usually are most time consuming it's not that much more than the first scan which costed 15,68. The last sort and group operations are very fast and just adds up to fractions of time units to complete.

Figure 11:

	QUERY PLAN
2	Sort (cost=42.61..42.61 rows=1 width=12)
3	Sort Key: (count(EXTRACT(month FROM ts.datetime)))
4	-> GroupAggregate (cost=42.53..42.60 rows=1 width=12)
5	Group Key: instructor.person_id
6	Filter: (count(EXTRACT(month FROM ts.datetime)) > 5)
7	-> Sort (cost=42.53..42.54 rows=3 width=12)
8	Sort Key: instructor.person_id
9	-> Nested Loop (cost=24.81..42.51 rows=3 width=12)
10	-> Hash Right Join (cost=24.64..26.48 rows=3 width=12)
11	Hash Cond: (il.time_slot_id = ts.id)
12	-> Seq Scan on individual_lesson il (cost=0.00..1.60 rows=60 width=4)
13	-> Hash (cost=24.61..24.61 rows=3 width=16)
14	-> Hash Right Join (cost=22.88..24.61 rows=3 width=16)
15	Hash Cond: (e.time_slot_id = ts.id)
16	-> Seq Scan on ensemble e (cost=0.00..1.52 rows=52 width=4)
17	-> Hash (cost=22.84..22.84 rows=3 width=16)
18	-> Hash Right Join (cost=15.72..22.84 rows=3 width=16)
19	Hash Cond: (gl.time_slot_id = ts.id)
20	-> Seq Scan on group_lesson gl (cost=0.00..6.26 rows=326 width=4)
21	-> Hash (cost=15.68..15.68 rows=3 width=16)
22	-> Seq Scan on time_slot ts (cost=0.00..15.68 rows=3 width=16)
23	Filter: (EXTRACT(month FROM datetime) = EXTRACT(month FROM CURRENT_DATE))
24	-> Memoize (cost=0.17..6.85 rows=1 width=4)
25	Cache Key: ts.person_id
26	Cache Mode: logical
27	-> Index Only Scan using instructor_pkey on instructor (cost=0.15..6.84 rows=1 width=4)
28	Index Cond: (person_id = ts.person_id)

5.2 Queries

The queries are probably not the fastest way to fetch the requested data. For example in the first query I managed to extract all the lesson types per month but couldn't find a neat way to add them together for the total-column. That resulted in that I had to create a separate view to sum them together and then join that with the other results. When analyzing with EXPLAIN that extra view costs around 95 time units while the full query costs around 172. That means that more than half the query time costs is related to that totaling column.