Seminar 3,
SQL

Data Storage Paradigms, IV1351

Esra Salman

esras@ug.kth.se

9/1 – 2024

Table of Contents

1 Introduction 3

2 Method 4

3 Result 6

4 Discussion 12

1 Introduction

Seminar 3 is the third phase in developing a single database application. The objective of this project is to create a database for the *Soundgood music school co* for information handling and business transaction facilitation, including all data and several specified operations. The foundation of the school is built on selling music lessons to students, which are offered in various categories and levels. The intended learning outcomes are stated as follows, taken from Canvas IV1351:

> *Use relational databases and query languages. Describe and explain basic concepts, principles and theories in the area of data/databases/data storage and in information administration and database design.*

With the aim of developing skills in crafting *Online Analytical Processing* (OLAP) queries and views, this seminar involved participation in SQL lectures. The topics covered included relational calculus, data modification, and nested queries, among others. Additionally, the seminar delved into *Database Storage*, addressing concepts and techniques related to database storage, as well as exploring the media and mechanisms utilized in *Database Management Systems* (DBMSs) for optimization. Furthermore, preparation for the task involved reading Chapter 16 in the provided course literature, “Fundamentals of Database Systems”.

The main objective of this seminar is to create OLAP, queries, and views that aim to analyze the school’s business and help create reports. It is additionally required to analyze one query’s efficiency with a specific method called EXPLAIN ANALYZE, available in the familiar PostgreSQL, and update the script from seminar 2 if needed. The higher-grade task includes a level of denormalization to serve the historical database part and the quires.

This seminar task was made alongside Daniel Ibrahimi and Ermia Ghaffari.

2 Method

To start this seminar, an extensive preparation was made by thoroughly examining ch. 14 & 16 in the course literature, “Fundamentals of Database Systems” as well as a comprehensive review of lectures *Database Storage* and *SQL* alongside *YouTube*-clips explaining *Denormalization* and *Historical data*. As mentioned above, seminar 3 primarily consists of creating OLAP queries and views. In accordance with the requirements, decisions were made in coding in regards to the data provided. The open-source database management system *PostgreSQL* (in the database management tool *pgAdmin*) was used for the given tasks. Initially began by the first task to execute several queries manually.

The first SQL query is expected to retrieve the number of lessons provided annually, month by month with the expected execution of several times during a week’s length. The result is expected to provide an integral for each month, alongside numbers for the type of lessons. The second query is expected to retrieve the number of students with no siblings or more with the same execution expectation with the output displaying an integer for the number of students and an integer for the number of siblings in the same row.

The third query is expected to retrieve list IDs and names of all instructors who have provided more than a given number of lessons during the intended month, to help the school keep track of instructors overworking with the expected daily execution. The final and fourth query is expected to retrieve all ensembles by music genre, weekday, and number of available seats for the upcoming week. To verify that the SQL query works as intended another database was created and a less amount of data inserted to compare data to output. This ensured the main data to be correctly output.

For the higher-grade, the model is assessed for denormalization which is the backward process for normalization, disorganizing and restructuring the data to achieve more redundancy and dependency to improve query performance. This is appropriate for

historical data since no update is occurring. Since the requirements are that the school should be able to access the lessons taken by which student at what cost. In order to build this track record a denormalized historical database is created with SQL statements storing merged lesson group, individual, and ensemble types, with genre of music, instruments, lesson price, name and email of the student.

Finally, when the queries are complete alongside the denormalized historical database they are verified to work as intended by the method mentioned above. Data is generated through a data generator tool (https://generatedata.com/).

3 Result

All queries are published in GitHub:

https://github.com/esrsal/IV1351.git

Queries & Tables:

In fig 3.1 a visual representation of the first query is shown of the task described above, see section 2. Designed to retrieve information on lessons, TO_CHAR function converts a value to a month, then COUNT provides the value of whatever is asked, i.e., individual, or other type of lessons. EXTRACT then, as the name suggests, retrieves a year which is particularly useful when only a portion of the full data is required. Fig. 3.1.2 shows the output of the above query, where month, type of lesson, total, and year are included.

```
SELECT
    TO_CHAR(T.timeslot_date, 'Mon') AS month,
    COUNT(IL.lesson_id) AS individual_lesson,
    COUNT(GL.lesson_id) AS group_lesson,
    COUNT(EL.lesson_id) AS ensemble_lesson,
    COUNT(IL.lesson_id) + COUNT(GL.lesson_id) + COUNT(EL.lesson_id) AS total,
    EXTRACT(YEAR FROM CURRENT_DATE) AS current_year
FROM
    timeslot AS T
JOIN lesson ON T.lesson_id = lesson.lesson_id
LEFT JOIN individual_lesson AS IL ON IL.lesson_id = lesson.lesson_id
LEFT JOIN group_lesson AS GL ON GL.lesson_id = lesson.lesson_id
LEFT JOIN ensemble_lesson AS EL ON EL.lesson_id = lesson.lesson_id
WHERE EXTRACT(YEAR FROM T.timeslot_date) = EXTRACT(YEAR FROM CURRENT_DATE)
GROUP BY
    TO_CHAR(T.timeslot_date, 'Mon')
ORDER BY
    CASE TO_CHAR(T.timeslot_date, 'Mon')
        WHEN 'Jan' THEN 1
        WHEN 'Feb' THEN 2
        WHEN 'Mar' THEN 3
        WHEN 'Apr' THEN 4
        WHEN 'May' THEN 5
        WHEN 'Jun' THEN 6
        WHEN 'Jul' THEN 7
        WHEN 'Aug' THEN 8
        WHEN 'Sep' THEN 9
        WHEN 'Oct' THEN 10
        WHEN 'Nov' THEN 11
        WHEN 'Dec' THEN 12
    END;
```

Figure 3.1: Queries for first task, amount of lessons given monthly during a year.

	month text	individual_lesson bigint	group_lesson bigint	ensemble_lesson bigint	total bigint	current_year numeric
1	Jan	17	25	12	54	2023
2	Feb	3	17	3	23	2023
3	Mar	36	43	33	112	2023
4	Apr	40	30	17	87	2023
5	May	15	51	3	69	2023
6	Jun	19	24	26	69	2023
7	Jul	8	26	3	37	2023
8	Aug	42	29	9	80	2023
9	Sep	22	50	15	87	2023
10	Oct	9	31	6	46	2023
11	Nov	28	7	15	50	2023
12	Dec	19	15	44	78	2023

Figure 3.1.2: Table output for first task, amount of lessons given monthly during a year.

In fig. 3.2 a table of task 2 is shown, as described in section 2, with up to two siblings (more is filtered through HAVING). Here for instance the number of students with one sibling amounts to 142. As seen in the queries in GitHub (linked above), a CTE expression family_counts is used to select the id of the family and iterate number of students with COUNT, this means if three students share the same family_id a new set is formed, which is also unique. The SUM(CASE WHEN sibling_count = …THEN…) sums the students based on the count of siblings. If three students share the same family_id, each has two siblings in the code. Then the result is grouped and ordered.

	no_of_siblings bigint	number_of_students bigint
1	0	154
2	1	142
3	2	87

Figure 3.2: Table output for second task, number of student of given number of siblings.

Fig. 3.3 shows the table including the output of task three, described in section 2. The school's supposed to track the instructors' lesson counts in order to prevent them from overworking. For instance, the table shows an id 17 connected to the name Medge Lynch, who has given 2 lessons. The main query operates on the results from

lesson_count, as seen in GitHub, then selecting id and full name from *instructor* and *person* entities, and lesson from the previous lesson_count, then joins it to finally give the results in the table which are filtered to only contain the previous month.

	instructor_id integer	first_name character varying (500)	last_name character varying (500)	given_lesson_count bigint
1	3	Kiara	Delaney	1
2	17	Medge	Lynch	2
3	35	Claudia	Mcdowell	1
4	42	Maisie	Miller	1
5	45	Kelsey	Simmons	1
6	51	Xenos	Schwartz	1
7	56	Wylie	Franks	1
8	61	Baker	Tucker	1
9	69	Edan	Fitzpatrick	1
10	83	Jane	Mcdonald	1
11	98	Lucius	Gill	1

Figure 3.3: Table output for third task, instructor's id with full name and amount of lessons given.

In fig 3.4 an excerpt of the queries for task 4 is shown. This task is dual, one part to create a view and the other consists of queries. The objective is to show all ensemble lessons that are scheduled for the coming week, alongside the availability seats. As seen in GitHub the query retrieves data from the view, selecting weekday, genre for the ensemble, then sums the available seats from the count of max and min students. The query is simply extracting information the view to give a number on the available seats.

	day_of_week text	genre character varying (500)	number_of_free_seats text	number_of_students bigint
1	Mon	Electronic	many seats	11
2	Mon	Hip	many seats	11
3	Mon	Dance	many seats	11
4	Mon	Reggae	many seats	8
5	Mon	Music	many seats	8
6	Mon	Electronic	many seats	8
7	Mon	Electronic	many seats	8
8	Mon	Hop	many seats	8
9	Mon	Country	many seats	8
10	Mon	Jazz	many seats	8
11	Sun	Music	many seats	12
12	Sun	Country	many seats	12
13	Sun	Music	many seats	12
14	Mon	Dance	many seats	6
15	Mon	(EDM)	many seats	6
16	Mon	Hop	many seats	6

Figure 3.4: Excerpt from table output for fourth task, ensemble lessons with genre, available seats and students.

Historical Database:

For the higher-grade task, a new sequence of SQL commands are incorporated for a new database which contains historical information. In order to establish a foreign server that links to the school's database on localhost at a given port, an extension *postgres_fdw* is implemented.

The process begins by generating a user mapping for the current user, establishing a connection to a foreign server using designated login credentials. Subsequently, a new schema called the 'historical schema' is created. The 'public' schema from the foreign server is then imported into this newly created schema. Within the 'historical schema,' a table named 'recording' is introduced, capturing details such as student name, email, lesson type, genre, instrument used, and price.

The last part of the script involves two SELECT queries, the first one retrieves information from multiple tables, consolidating data related to students, lessons, prices, and additional entities. LEFT JOINs are utilized to accommodate potential null values.

The second query focuses on extracting specific columns from the 'recording' table within the 'historical schema,' providing a streamlined and comprehensive view of student recordings.

```
CREATE DATABASE historical;
CREATE EXTENSION IF NOT EXISTS postgres_fdw;
CREATE SERVER histserver
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (dbname 'SoundGoodMusic', host 'localhost', port '5432');
CREATE USER MAPPING FOR current_user
SERVER histserver
OPTIONS (user 'postgres', password 'xxxx');
CREATE SCHEMA historical_schema;
IMPORT FOREIGN SCHEMA public FROM SERVER histserver
    INTO historical_schema;
CREATE TABLE historical_schema.recording (
    record_id SERIAL PRIMARY KEY,
    student_name VARCHAR(255),
    student_last_name VARCHAR(255),
    student_email VARCHAR(255),
    lesson_type VARCHAR(255),
    genre VARCHAR(255),
    instrument_used VARCHAR(255),
    price NUMERIC
);
SELECT
    person.first_name AS student_name,
    person.last_name AS student_last_name,
    email.email_address AS student_email,
    lesson_type_ENUM.lesson_type AS lesson_type,
    ensemble_lesson.genre AS genre,
    COALESCE(individual_lesson.instrument_used,
        group_lesson.instrument_used) AS instrument_used,
    price_management.price AS price
```

Figure 3.5: Query for higher-grade task, historical database with students' full names, emails, lesson type, price, and genre (for ensemble).

	first_name character varying (500)	last_name character varying (500)	email_address character varying (200)	lesson_type character varying (200)	price money	genre character varying (500)	instrument_us character vary
52	Carolyn	Chen	nunc@hotmail.net	Group	$600.00	[null]	Trumpet
53	Maxwell	Graves	bibendum.sed.est@protonmail.couk	Group	$600.00	[null]	Trumpet
54	Maxwell	Graves	bibendum.sed.est@protonmail.couk	Group	$600.00	[null]	Trumpet
55	Sybill	Bell	elementum@yahoo.com	Group	$600.00	[null]	Trumpet
56	Lara	Simpson	lectus.pede.ultrices@protonmail.org	Group	$600.00	[null]	Trumpet
57	Evangeline	Bentley	mauris@icloud.ca	Group	$600.00	[null]	Trumpet
58	Amy	Perkins	eu.tellus@icloud.ca	Group	$600.00	[null]	Trumpet
59	Carolyn	Harris	consectetuer.euismod@aol.ca	Group	$600.00	[null]	Trumpet
60	Reece	Frost	nullam.ut@icloud.ca	Group	$600.00	[null]	Trumpet
61	Brendan	Kline	ac.sem@outlook.com	Group	$600.00	[null]	Trumpet
62	Louis	Burke	augue.eu@hotmail.ca	Group	$600.00	[null]	Trumpet
63	Kasper	Bean	donec@yahoo.com	Group	$600.00	[null]	Saxophone
64	Carolyn	Chen	nunc@hotmail.net	Group	$600.00	[null]	Saxophone
65	Maxwell	Graves	bibendum.sed.est@protonmail.couk	Group	$600.00	[null]	Saxophone
66	Maxwell	Graves	bibendum.sed.est@protonmail.couk	Group	$600.00	[null]	Saxophone
67	Sybill	Bell	elementum@yahoo.com	Group	$600.00	[null]	Saxophone
68	Lara	Simpson	lectus.pede.ultrices@protonmail.org	Group	$600.00	[null]	Saxophone
69	Evangeline	Bentley	mauris@icloud.ca	Group	$600.00	[null]	Saxophone
70	Amy	Perkins	eu.tellus@icloud.ca	Group	$600.00	[null]	Saxophone
71	Carolyn	Harris	consectetuer.euismod@aol.ca	Group	$600.00	[null]	Saxophone
72	Reece	Frost	nullam.ut@icloud.ca	Group	$600.00	[null]	Saxophone
73	Brendan	Kline	ac.sem@outlook.com	Group	$600.00	[null]	Saxophone

Figure 3.6: Excerpt from table output for higher-grade task, historical database with students' full names, emails, lesson type, price, and genre (for ensemble).

4 Discussion

The queries, views, tables and historical database presented in this seminar are conducted with all requirements taken into consideration. Based on the assessment criteria regarding name conventions, auto-commit, SELECT, listing and renting instruments and terminating rentals, all are met. Expanding on the concise overview of queries provided earlier, three subjects are of interest in this section: *Refactoring and Trade-Off, Views and Readability*, and *Time Analysis*.

In the SQL tasks, when attempting to retrieve historical information the previously used model and design from seminar 2 was not altered (see fig. 4.2). This was due to the intent of optimizing memory usage, particularly post the decisions made during the second phase when balancing the memory and query complexity. This is mainly connected to the decisions regarding normalization and denormalizations, and the pros and cons of weighing on one more than the other. The advantages of denormalization include improvement of query performance due to a smaller number of joins, which also leads to faster execution, there is also an enhancement in less query complexity. However, the disadvantages could also be risk of duplication which both increase memory usage and performance issues. The increased data could lead to challenges regarding anomalies increasing. As explained in section 2, the historical database requires no changes or updates therefore a level of denormalization is required. The final task necessitated employing multiple join methods (see fig. 3.5) as intentionally distributing the data in the database to prevent redundancy. See fig 3.6 for historical data output.

While contemplating options to reducing join methods, specifically individual_lesson, group_lesson, and ensemble_lesson into the lesson entity, we recognized the potential introduction of issues related to handling empty or repeated information within one large group. Another prospective approach for simplifying queries could involve merging the email, person, and student entities. This would facilitate quicker access to email, name, and lesson information without additional entity joins and attribute

selection. These examples underscore the intricate nature of achieving the right balance and making informed decisions when enhancing SQL queries, see fig. 3.5.

In the provided query execution plan analysis (EXPLAIN ANALYZE), the most resource-intensive tasks are the sorting operation and specific join operations. The sorting process, which arranges the result set based on the 'Mon' (month abbreviation) derived from 'timeslot_date,' stands out as particularly costly. This is evident from the actual time ranging between 20.244 to 20.490 seconds and the involvement of 986 rows. Sorting large datasets poses a computational challenge, and in this case, it significantly contributes to the overall query execution time.

The sequential scans on 'ensemble_lesson,' 'group_lesson,' and 'individual_lesson' are relatively efficient as they deal with small row counts (50, 101, and 60 rows, respectively). However, the subsequent hash right join operations with the 'lesson' table incur higher costs. The first hash right join with 'ensemble_lesson' involves 50 rows, the second with 'group_lesson' involves 101 rows, and the third with 'individual_lesson' involves 60 rows. While these join operations contribute to the overall computational load, their costs are partially mitigated by the reduced result sets from previous steps.

The nested loop operation towards the end of the plan is less computationally expensive, handling 219 rows. Its impact is notably influenced by the smaller result set obtained from preceding operations, highlighting the effectiveness of sequential filtering. In conclusion, the primary performance bottlenecks in this query stem from the substantial sorting operation and specific join operations. Despite their resource-intensive nature, these operations are strategically employed in conjunction with smaller result sets to optimize the overall performance of the query.

Finally, this task’s objective was to dually create queries for specified tasks and a new historical database, both are fulfilled.

1	GroupAggregate (cost=30.59..30.63 rows=1 width=96) (actual time=20.244..20.490 rows=12 loops=1)
2	Group Key: (to_char((t.timeslot_date)::timestamp with time zone, 'Mon'::text))
3	-> Sort (cost=30.59..30.59 rows=1 width=44) (actual time=20.207..20.282 rows=986 loops=1)
4	Sort Key: (to_char((t.timeslot_date)::timestamp with time zone, 'Mon'::text))
5	Sort Method: quicksort Memory: 71kB
6	-> Hash Right Join (cost=18.64..30.58 rows=1 width=44) (actual time=19.212..19.822 rows=986 loops=1)
7	Hash Cond: (el.lesson_id = lesson.lesson_id)
8	-> Seq Scan on ensemble_lesson el (cost=0.00..11.40 rows=140 width=4) (actual time=0.010..0.018 rows=50 loops=1)
9	-> Hash (cost=18.62..18.62 rows=1 width=16) (actual time=2.008..2.013 rows=687 loops=1)
10	Buckets: 1024 Batches: 1 Memory Usage: 38kB
11	-> Hash Right Join (cost=16.24..18.62 rows=1 width=16) (actual time=1.483..1.745 rows=687 loops=1)
12	Hash Cond: (gl.lesson_id = lesson.lesson_id)
13	-> Seq Scan on group_lesson gl (cost=0.00..2.00 rows=100 width=4) (actual time=0.016..0.031 rows=101 loops=1)
14	-> Hash (cost=16.23..16.23 rows=1 width=12) (actual time=1.447..1.451 rows=439 loops=1)
15	Buckets: 1024 Batches: 1 Memory Usage: 27kB
16	-> Hash Right Join (cost=14.39..16.23 rows=1 width=12) (actual time=1.113..1.272 rows=439 loops=1)
17	Hash Cond: (il.lesson_id = lesson.lesson_id)
18	-> Seq Scan on individual_lesson il (cost=0.00..1.60 rows=60 width=4) (actual time=0.009..0.019 rows=60 loops=1)
19	-> Hash (cost=14.38..14.38 rows=1 width=8) (actual time=1.082..1.084 rows=219 loops=1)
20	Buckets: 1024 Batches: 1 Memory Usage: 17kB
21	-> Nested Loop (cost=0.15..14.38 rows=1 width=8) (actual time=0.196..0.983 rows=219 loops=1)
22	-> Seq Scan on timeslot t (cost=0.00..6.20 rows=1 width=8) (actual time=0.039..0.280 rows=219 loops=1)
23	Filter: (EXTRACT(year FROM timeslot_date) = EXTRACT(year FROM CURRENT_DATE))
24	Rows Removed by Filter: 21
25	-> Index Only Scan using lesson_pkey on lesson (cost=0.15..8.17 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops…
26	Index Cond: (lesson_id = t.lesson_id)
27	Heap Fetches: 219
Total rows: 29 of 29	Query complete 00:00:00.051

Figure 4.1: EXPLAIN ANALYZE excerpt output from pgAdmin.

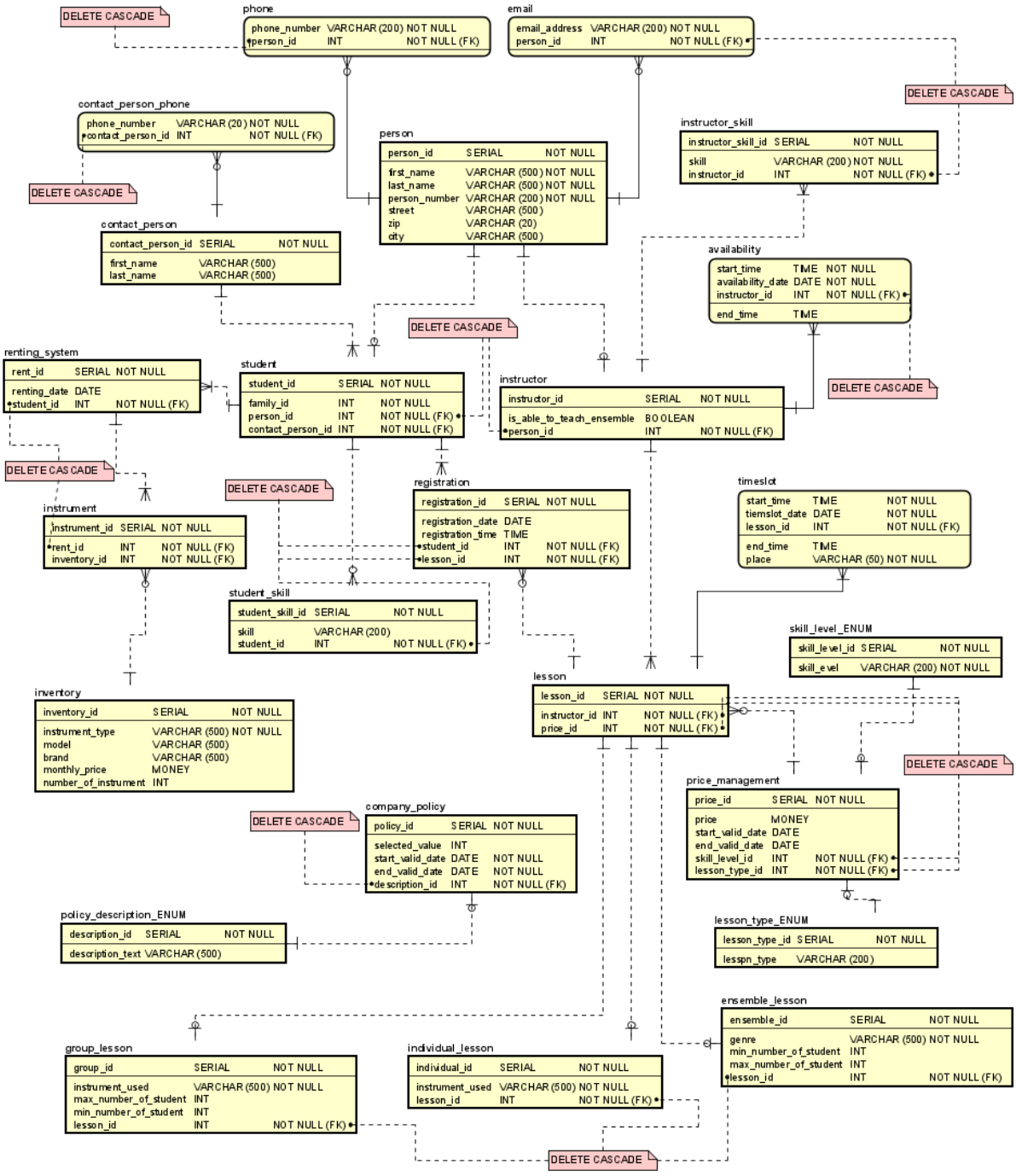

Figure 4.2: Physical & Logical Model from seminar 2.