IV1351
Project report

Kalle Elmdahl - kelmdahl@kth.se
22-01-04

1 Introduction

This report covers a project for the course IV1351. The project was worked on during the whole >2 month long course. The content of the project is based on a music school called Soundgood Music School and its requirements. The requirements are specified on the course page but can be summarized. The task is to create a working database system for the school. The school teaches lessons and rents instruments to its students.

The school also wants a simple application and pre-made database access programs called views on top of the database. This was created for the purpose of the project as SQL views, materialized or not, and a java program with a simple terminal view for renting instruments.

During the project the result was created with discussion from Erik Malm and Hampus Nilsson, however all of our solutions are different.

1.1 Database

The database itself needed to store information about

- Students and instructors
- Lessons
- Rentable instruments
- Student's rentals over time
- An archive of lessons previously taught

Student data included information about the student itself, some form of parent contact and sibling information for a sibling discount. For lessons, time and place was of course stored but also data about which type of lesson, if it was an ensemble lesson, the lessons genre needed to be stored. Rentable instruments needed data about their condition instrument type and some indicator of its availability. Data about how the students rent instruments, meaning time and duration.

The archive was not a necessity but it was a possible extra. The archive contains data about lessons in the past, in an easily accessible manner.

1.2 Views

The first view was needed to show statistics about lessons held during a specific year. It shows how many of each lesson type was held during each month. The second view does the same but shows the average number of lessons of each type per month.

The third view showed overworked instructors by showing how many lessons each instructor has given if they have given over a specified limit of lessons.

View four was needed for the school’s website and show next week’s ensemble lessons and how full they are

1.3 Application

The application was a simple program for renting instruments. The application can list available instruments students can rent. It can start a new rental and terminate it. Information about the rental is stored even after the rental is returned therefore students and administrators can view what rentals have been made.

2 Literature Study

During development in the project, it was important to study and have knowledge about the thing being worked on. The main lectures were followed over the course to get a good understanding of broader, and sometimes more specific concepts. The pre recorded lectures were a great way to get a specific understanding of what to do in tasks 1,2 and 4. Much modeling related information was also taken from the course literature. The course literature described how to organize and structure the models, and the combination of the course literature and the pre-recorded lectures was key to make a good model.

Inheritance was a concept not much described in the course, therefore before doing the higher grade tasks a lot of research was done in the form of reading in the course literature. There one could learn of the many ways to do inheritance in a database and how to model it. This was a key part in understanding the higher grade tasks.

For task 3 the live lecture was great to get a basic understanding. However, to get a deeper understanding, especially about more advanced statements, the course literature was a good reference, together with a few internet searches.

For the last task a deep understanding of ACID transactions was needed. For this, the course literature and the live lecture were great sources of information and one could learn a lot about specifics of the properties in the course literature.

2.1 Summary

Task 1
Before development: Live lectures and course literature.
During development: pre-recorded lectures and checking with the course literature.
Inheritance part: Course literature.

Task 2
Before development: Live lectures and course literature.
During development: pre-recorded lectures and checking with the course literature.
Inheritance part: Course literature.

Task 3
Before development: Live lectures and course literature.
During development: Live lectures, course literature and internet.

Task 4
Before development: Live lectures and course literature
During development: pre-recorded lectures, checking with the course literature and live lecture on transactions

3 Metod

3.1 Task 1

When creating the conceptual model a 5 part process was followed.

3.1.1 Nouns

Coming up with describing nouns of which describe what is being modeled. In this case it is the *Soundgood Music School*. The school has provided a description of the school which includes what services it offers. From there, nouns can be taken out to form as “entity candidates”. These are possible entities for the Conceptual Model. Not all nouns need to be taken out of the text but it is prefered to have too many than too little.

3.1.2 More Entities

Use more of the imagination to come up with more entity candidates. In this part a category list is used as help to come up with them. This is the last part where entities are added so it is important to make sure all possible entities are in the model at this part. The categories are related to things going on in the background. This can be easy to miss in the description so it is important to consider this separately.

3.1.3 Remove

After the first two steps many unnecessary entities are probably in the model. This is mainly because creating a conceptual model is an iterative process and improvements to the understanding what entities should be in the case of just this model are likely to come up with. The possibility of duplicates from part one and two are common where two entities’ purposes are almost the same thing. After this step, all entities should be defined for the model.

3.1.4 Attributes

All entities for the model are now purposeful and have a name. However, it is not quite clear what information the entities themself contain. Now attributes are needed to describe what data each entity could hold. For example a “Student” entity probably needs some sort of name and contact information attribute for the school to inform and keep track of its students. This process can be quite tedious for bigger tasks such as this because it is not always completely clear as to what information is needed so it can sometimes end up being a quite creative process if no customer is available to talk to.

3.1.5 Relations

All the entities should now get connections to other related entities. Some attributes can not be attributed because they do not hold primitive data, in that case a separate entity is needed and a relation between them could be defined. An example of this is the address entity. It could be an attribute but it is probably better as a separate entity so zip code city and the address can be separated.

A relation also holds a name and a cardinality. This is to show how many of each the other one can have. On one side it is either zero or one or one, with one meaning that entity is required for the other entity to exist. On the other side zero or more, one or more or zero or one. To know which one to use one needs to think of how many of it the other relation can possibly have. For example, a relation between “person” and “address” one would probably relate person to address saying zero or one person lives at zero or more addresses but the customer is most likely not interested in more than one address so a relation: zero or more person lives at zero or one address because most likely there is a family living there and the students can have siblings.

3.1.6 Finalise

Now the concept model should be complete, but in the case of the process for this report many iterations and final tweaking was needed before the model was done. Because of how long the description was, it was easy to miss information and overlook certain details.

Explain the procedure you followed to create the conceptual model. You shall mention all steps that are covered in the videos on conceptual modeling. If you did not perform a particular step, explain why the result was better (or at least not worse) without that step. Do not explain the result of each step, only explain the steps themselves.

3.2 Task 2

Converting the conceptual model to a logical and physical model was a quite long process. A new naming convention was introduced, primary and foreign keys were added, deletion handling and much more was introduced to create the new model.

3.2.1 Cleaning up the model

Starting from the conceptual model relations were firstly removed, then a new naming convention was introduced with all lower case letters and underscores separating the words, this was applied to both attributes and entity names. Attributes were now given a datatype and the attribute could also be prohibited from being null. Multivalued attributes were placed in their own entities.

3.2.2 Adding back relations

The relations from the conceptual model were now needed to be reintroduced, but now id:s were needed to be taken care of. Every entity needs a primary key, a unique identifying set of keys or one key. All the strong entities were given their own ids. These entities are the ones like “skill_level” because there is no other class that can make it into a subclass, it is strong on its own. Other examples are “lesson” and “instrument”. After these were given id:s relations could be added back with their ids as foreign key attributes on other entities. For example, the “lesson” entity was given a foreign key to “skill_level” with the “skill_level_id” attribute. The same process was done for all the non-“many-to-many” relations and entities without id:s could either have the other relating entity’s primary key, or it could be given a surrogate key. Surrogate keys were needed if the relation was one to many because otherwise multiple rows in the database could have the same id. Other relations like the one between “lesson” and “group_lesson” had the primary key of lesson in the grouplesson as well.

Many to many relations needed a new solution as arrays are not possible in a sql database. Therefore a new entity is needed. This entity only holds id:s of the many-to-many classes. As an example, the student attends multiple lessons, and one lesson can teach multiple students, here a new entity “student_lesson” is needed with primary keys as the “lesson_id” and “student_id”. After this, a one to many relation from both “lesson” and “student” could be made.

3.2.3 Finishing the model

Now the model is almost done. The model must at this point be checked to be in 3 normalized form (3NF). The model should already be in 1NF as the multivalued attributes have been made into separate entities. 2NF should be taken care of as most entities have surrogate keys. But this needs to be checked to make sure all primary keys relate to the other attributes. to make sure the model is in 3NF all the attributes need to be checked to make sure they are directly related to the entity itself, there should not be a transitive relation between attributes.

3.2.4 Creating and filling the database

The model is now finished and the database can be created. The database is created by firstly exporting the model from the modeling program as an sql file. Then all the notes that cannot be explained in the model are added to the sql script. This includes serializing primary key attributes and setting deletion actions. This script can then be executed to create the database with the columns given in the model. Then another sql script is created to fill the database. The database is filled by firstly adding data to the strong entities, this denotes the ones without a foreign key i.e. “discount” and “skill_level”. Then the database can be filled by creating data for entities that have all their foreign keys data generated.

3.3 Task 3

For all the tasks in this project PostgreSQL was the database management system (DBMS) of choice. PostgreSQL is an open source DBMS and is easy to use. To develop the queries. An administrative program called pgAdmin was used to manage the database and test queries to streamline the testing process. The queries themselves were written inside pgAdmin and the insert data scripts were written in Visual Studio Code (VSCode) because it has many productivity features for easy manipulation of code and text.

The process of making the queries was iterative. The queries in the result are quite long so having multiple tabs open at once and working on the different parts was vital to get clean and well made queries.

To verify the queries work, they were run in pgAdmin and manually verified with the inserted test data from task 2. The manual verification process could look different depending on the query but mostly consisted of verifying each part of the query by itself. An example of this could be when starting a new query, testing a date range first, then commenting that out and then testing a number operation and after that putting it together and seeing if a reasonable result was presented.

3.4 Task 4

Creating the application was quite a tedious process. After watching the lectures for this task, the github repository of the banking application was used as a reference throughout the task. The first steps of the application included getting the view to work. The view in the application is the same as the one in the banking application with slight modifications. After this, the controller and integration were created to get the database data to the program. As a start, a simple find query was created for testing. This was then used to create the model “RentalInstrument” and its DTO interface. Finally, some simple modifications could be made to the code to create the list rentals command.

After this was done the more difficult part of the application was created. Renting and terminating rentals. This was made by referencing update queries from the git repository and modifying it for the purpose of the application. The database functions were made as simple as possible in the beginning and more advanced functionality like locks were added when the core functionality worked. When the commands were implemented it was realized the program did not store historical data, as was mentioned in the beginning of the project. But in the last task this was a necessity. Therefore the architecture of the database needed to be reconstructed, and a rental table was added. Every row in this database is a rental for a student. The columns in the table include start date, end date and if it is returned/terminated.

4 Result

The result chapter of this report is divided into each task. All code made in the project is located in the same github repository.

Git repository

4.1 Task 1

Below is the final conceptual model. The model should contain all the necessary information for the presented system in the project description. This includes: Lessons, students, instructors and renting instruments. As visible in the note below all attributes have a standard cardinality and special cardinalities are specified inside the brackets.

4.1.1 Model with inheritance

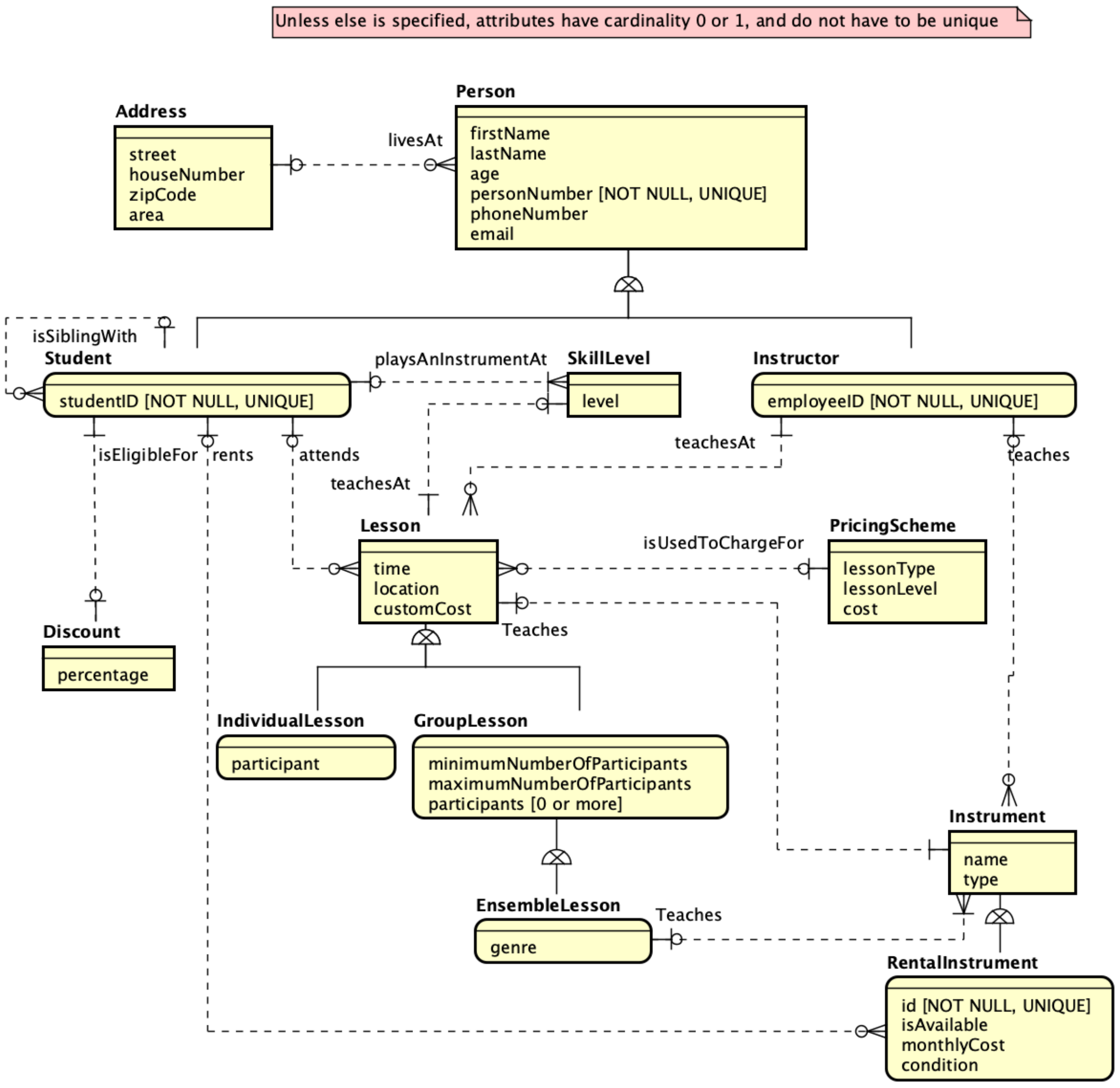


Figure 1 - The conceptual model without inheritance

The lesson and person entities show cases of inheritance, it states that students have many things in common with instructors and the same for lesson types, so they are generalized into a parent entity holding all the shared data.

4.1.2 Requirements

The diagram is made in UML and the IE notation is used

4.1.2.1 Lesson

- Lesson types are shown as different entities.
- Group lessons have participant requirements as attributes.
- Skill levels are a separate entity and related to the lesson.
- Lessons are related to the instrument entity with different cardinality for lesson types.
- Ensembles have a genre.
- All lessons have a time attribute. This can be used in different ways depending on lesson type, this is an application level problem.
- Instructors are related to instruments.
- Instructor availability can be based on lesson course bookings.
- Student course bookings are at application level but accepted students are related to the lesson entity.

4.1.2.2 Student

- Student information is stored in the person and student entities.
- Parental contact can be done by adding another phone number.
- Siblings are taken into account by the student to student relation.

4.1.2.3 Instructor

- Instructor information is stored the same way as student information.
- Instructor course bookings are made by application
- Ensemble availability is matched against instruments

4.1.2.4 Student enrollment

- Attending the school is application level
- Students have a skill level for each instrument
- Availability on lessons is application level

4.1.2.5 Student payment

- Each lesson have a price how these are set are application level
- Sibling discount is calculated based on the student to student relation
- discount is stored in the discount entity
- Pricing schemes are premade pricings for lessons, can be added to a lesson or a custom cost can be set.

4.1.2.6 Instructor payment

- Instructor payments is application level, lesson costs are stored in the lesson entity

4.1.2.7 Renting instruments

- Deliveries are application level.
- Rental instruments are a type of instrument. They have names and types.
- Limitations and payment are application level

4.1.3 Model without inheritance

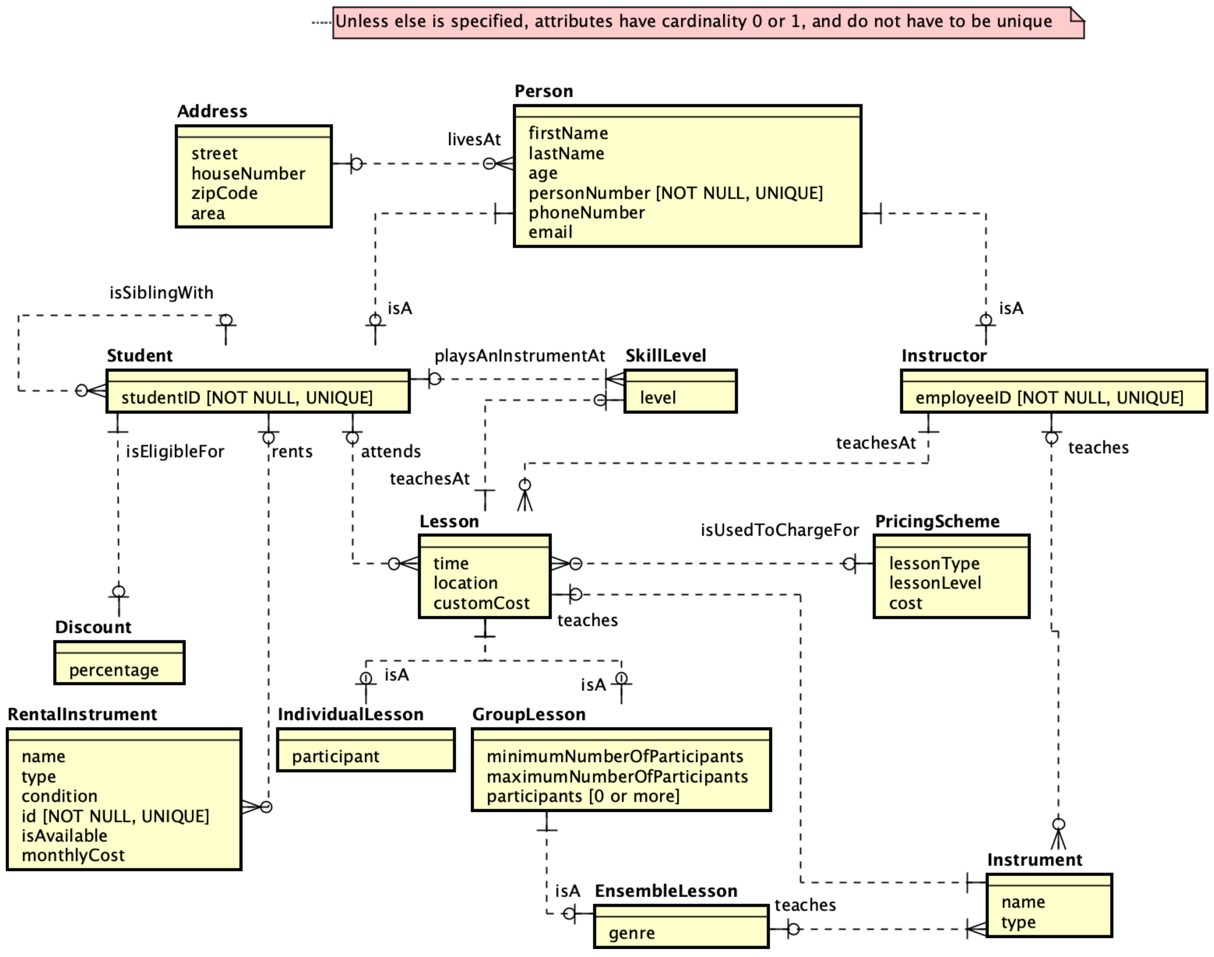


Figure 2 - Conceptual model with inheritance

The conceptual model without inheritance is mostly the same. It shows multiple ways of removing inheritance. One can either add the inherited attributes, as done in “RentalInstrument” or add an “isA” relation between them. Both work and they have their advantages.

4.2 Task 2

The logical and physical mode can be seen below. The notes show some business or database rules which are unable to be modeled.

4.2.1 The model

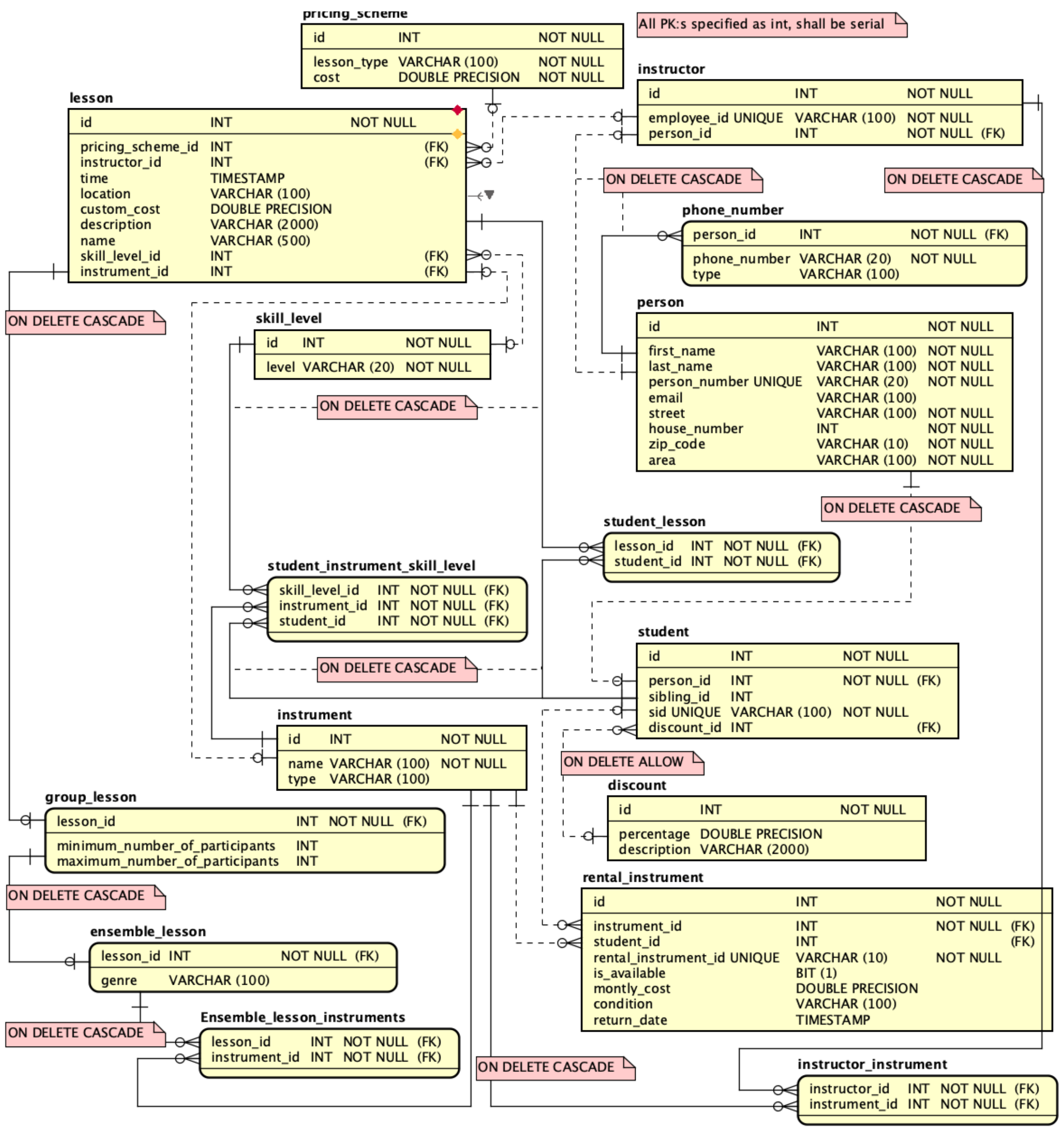

pricing_scheme
id INT NOT NULL
lesson_type VARCHAR (100) NOT NULL
cost DOUBLE PRECISION NOT NULL
All PK:s specified as int, shall be serial
instructor
id INT NOT NULL
employee_id UNIQUE VARCHAR (100) NOT NULL
person_id INT NOT NULL (FK)
lesson
id INT NOT NULL
pricing_scheme_id INT (FK)
instructor_id INT (FK)
time TIMESTAMP
location VARCHAR (100)
custom_cost DOUBLE PRECISION
description VARCHAR (2000)
name VARCHAR (500)
skill_level_id INT (FK)
instrument_id INT (FK)
ON DELETE CASCADE
ON DELETE CASCADE
phone_number
person_id INT NOT NULL (FK)
phone_number VARCHAR (20) NOT NULL
type VARCHAR (100)
ON DELETE CASCADE
skill_level
id INT NOT NULL
level VARCHAR (20) NOT NULL
person
id INT NOT NULL
first_name VARCHAR (100) NOT NULL
last_name VARCHAR (100) NOT NULL
person_number UNIQUE VARCHAR (20) NOT NULL
email VARCHAR (100)
street VARCHAR (100) NOT NULL
house_number INT NOT NULL
zip_code VARCHAR (10) NOT NULL
area VARCHAR (100) NOT NULL
ON DELETE CASCADE
ON DELETE CASCADE
student_lesson
lesson_id INT NOT NULL (FK)
student_id INT NOT NULL (FK)
student_instrument_skill_level
skill_level_id INT NOT NULL (FK)
instrument_id INT NOT NULL (FK)
student_id INT NOT NULL (FK)
student
id INT NOT NULL
person_id INT NOT NULL (FK)
sibling_id INT
sid UNIQUE VARCHAR (100) NOT NULL
discount_id INT (FK)
ON DELETE CASCADE
instrument
id INT NOT NULL
name VARCHAR (100) NOT NULL
type VARCHAR (100)
ON DELETE ALLOW
discount
id INT NOT NULL
percentage DOUBLE PRECISION
description VARCHAR (2000)
group_lesson
lesson_id INT NOT NULL (FK)
minimum_number_of_participants INT
maximum_number_of_participants INT
ON DELETE CASCADE
rental_instrument
id INT NOT NULL
instrument_id INT NOT NULL (FK)
student_id INT (FK)
rental_instrument_id UNIQUE VARCHAR (10) NOT NULL
is_available BIT (1)
montly_cost DOUBLE PRECISION
condition VARCHAR (100)
return_date TIMESTAMP
ensemble_lesson
lesson_id INT NOT NULL (FK)
genre VARCHAR (100)
ON DELETE CASCADE
Ensemble_lesson_instruments
lesson_id INT NOT NULL (FK)
instrument_id INT NOT NULL (FK)
ON DELETE CASCADE
instructor_instrument
instructor_id INT NOT NULL (FK)
instrument_id INT NOT NULL (FK)

Figure 3 - The logical and physical model

4.2.2 Changes

Many changes were made from the conceptual model as a better understanding of the project is made.

- Individual lessons are now reduced to being just a “lesson”
- Instruments are seen as an abstract table, not used for physical instruments.
- A lot of denormalization is done and many insignificant attributes are added throughout the model

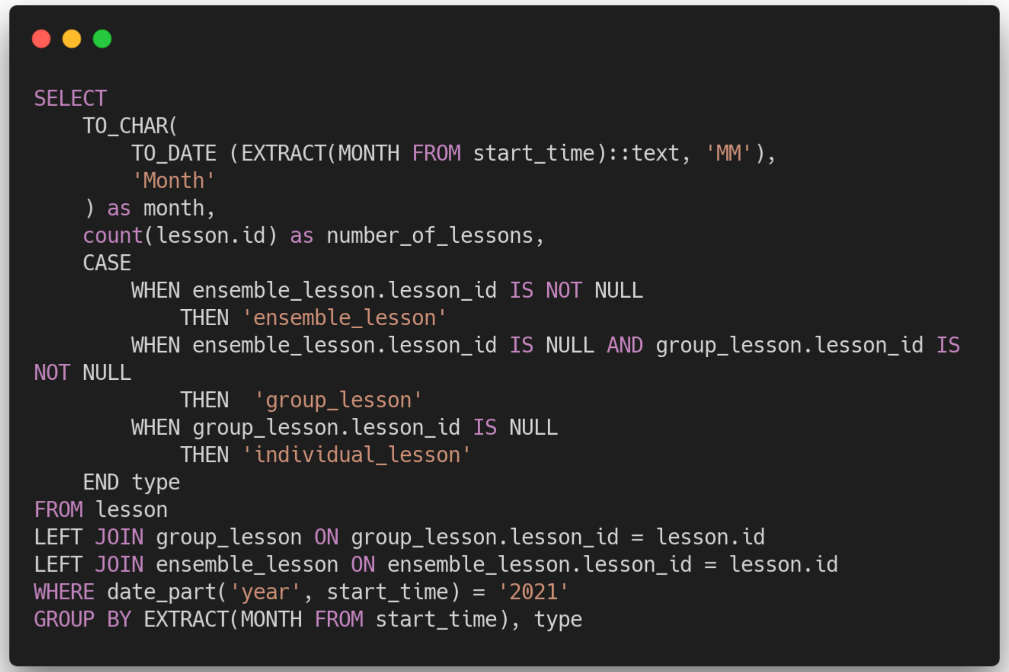
The requirements are still met as the solution still has the same general idea conceptual model

4.3 Task 3

Below all the queries for each sub task can be seen, and its result. The query is also explained in the details part.

4.3.1 Query 1

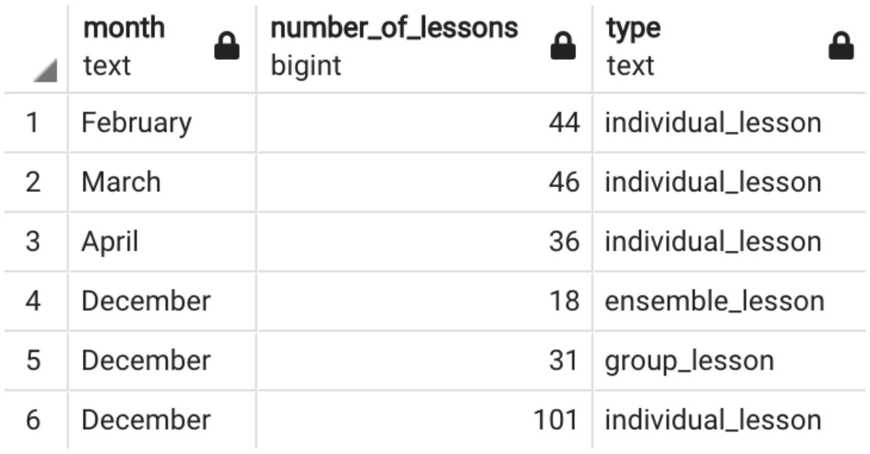
Code



```
SELECT
    TO_CHAR(
        TO_DATE (EXTRACT(MONTH FROM start_time)::text, 'MM'),
        'Month'
    ) as month,
    count(lesson.id) as number_of_lessons,
    CASE
        WHEN ensemble_lesson.lesson_id IS NOT NULL
            THEN 'ensemble_lesson'
        WHEN ensemble_lesson.lesson_id IS NULL AND group_lesson.lesson_id IS
NOT NULL
            THEN  'group_lesson'
        WHEN group_lesson.lesson_id IS NULL
            THEN 'individual_lesson'
    END type
FROM lesson
LEFT JOIN group_lesson ON group_lesson.lesson_id = lesson.id
LEFT JOIN ensemble_lesson ON ensemble_lesson.lesson_id = lesson.id
WHERE date_part('year', start_time) = '2021'
GROUP BY EXTRACT(MONTH FROM start_time), type
```

Figure 4 - code for query 1

Result



	month text	number_of_lessons bigint	type text
1	February	44	individual_lesson
2	March	46	individual_lesson
3	April	36	individual_lesson
4	December	18	ensemble_lesson
5	December	31	group_lesson
6	December	101	individual_lesson

Figure 5 - result for query 1

4.3.1.1 Details

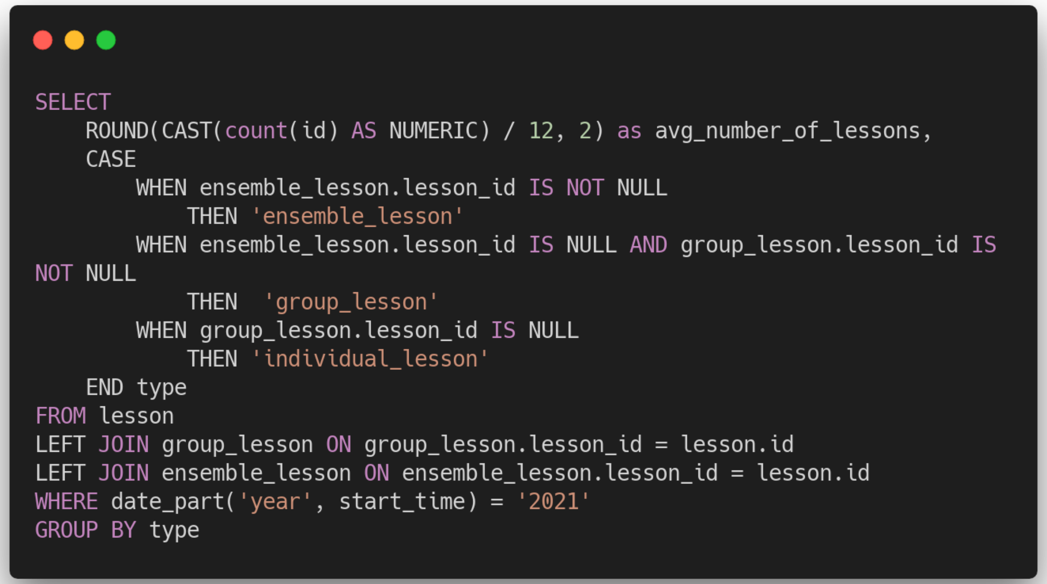
The query works by grouping the lessons into the three types by checking for which tables their id exist in using the “CASE” statement and grouping by it using the “GROUP BY” statement. The results are also grouped by month based on the start_time of the lesson and the grouping is counted using the count() aggregate function.

4.3.1.2 Requirements

- ✓ It shall be possible to retrieve the total number of lessons per month
- ✓ get specific number of individual lessons, group lessons and ensembles

4.3.2 Query 2

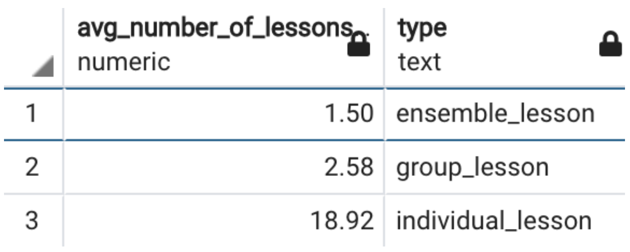
Code



```
SELECT
    ROUND(CAST(count(id) AS NUMERIC) / 12, 2) as avg_number_of_lessons,
    CASE
        WHEN ensemble_lesson.lesson_id IS NOT NULL
            THEN 'ensemble_lesson'
        WHEN ensemble_lesson.lesson_id IS NULL AND group_lesson.lesson_id IS
NOT NULL
            THEN  'group_lesson'
        WHEN group_lesson.lesson_id IS NULL
            THEN 'individual_lesson'
    END type
FROM lesson
LEFT JOIN group_lesson ON group_lesson.lesson_id = lesson.id
LEFT JOIN ensemble_lesson ON ensemble_lesson.lesson_id = lesson.id
WHERE date_part('year', start_time) = '2021'
GROUP BY type
```

Figure 6 - code for query 2

Result



	avg_number_of_lessons numeric	type text
1	1.50	ensemble_lesson
2	2.58	group_lesson
3	18.92	individual_lesson

Figure 7 - result for query 2

4.3.2.1 Details

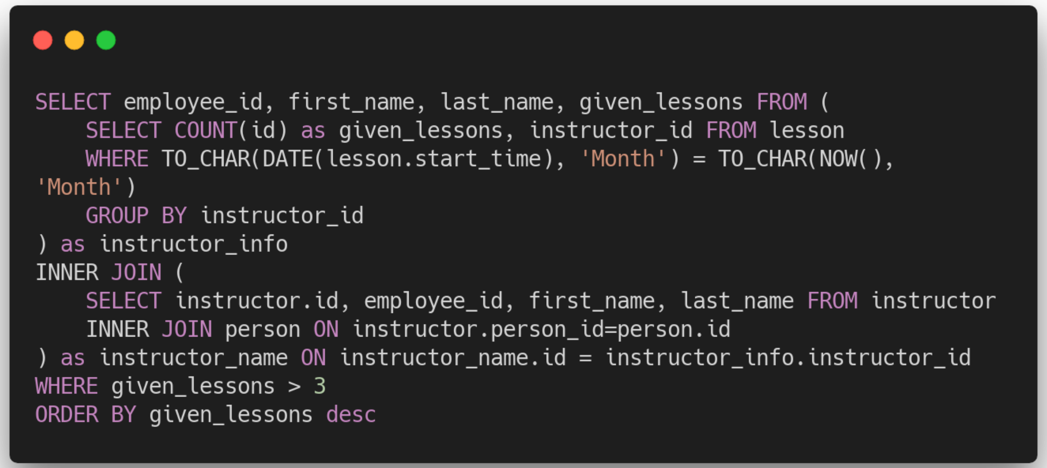
This query is almost the same as query one. The main difference is that the result is not grouped by month. Instead, it is only grouped by type and the final count is divided by twelve because there are twelve months in a year.

4.3.2.2 Requirements

✓ Retrieve the average number of lessons per month during the entire year

4.3.3 Query 3

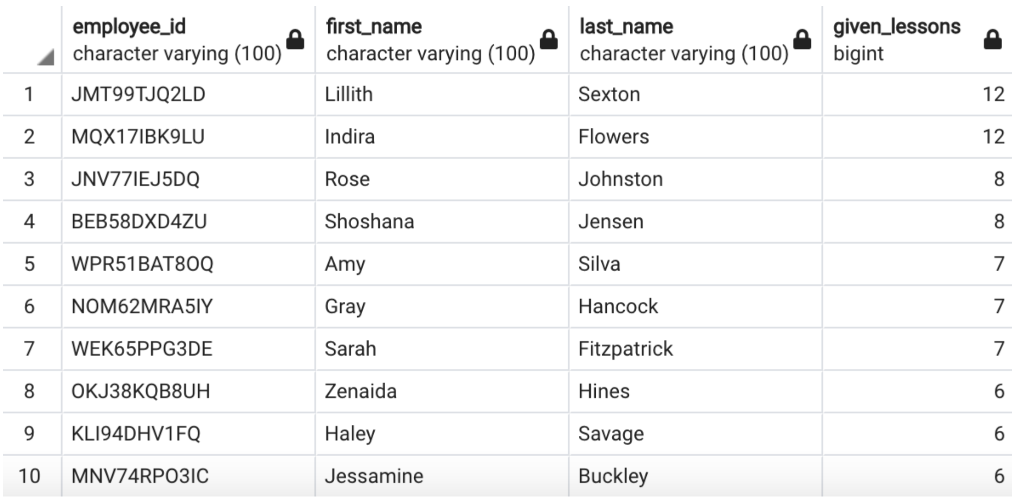
Code



```
SELECT employee_id, first_name, last_name, given_lessons FROM (
    SELECT COUNT(id) as given_lessons, instructor_id FROM lesson
    WHERE TO_CHAR(DATE(lesson.start_time), 'Month') = TO_CHAR(NOW(),
'Month')
    GROUP BY instructor_id
) as instructor_info
INNER JOIN (
    SELECT instructor.id, employee_id, first_name, last_name FROM instructor
    INNER JOIN person ON instructor.person_id=person.id
) as instructor_name ON instructor_name.id = instructor_info.instructor_id
WHERE given_lessons > 3
ORDER BY given_lessons desc
```

Figure 8 - code for query 3

Result



	employee_id character varying (100)	first_name character varying (100)	last_name character varying (100)	given_lessons bigint
1	JMT99TJQ2LD	Lillith	Sexton	12
2	MQX17IBK9LU	Indira	Flowers	12
3	JNV77IEJ5DQ	Rose	Johnston	8
4	BEB58DXD4ZU	Shoshana	Jensen	8
5	WPR51BAT8OQ	Amy	Silva	7
6	NOM62MRA5IY	Gray	Hancock	7
7	WEK65PPG3DE	Sarah	Fitzpatrick	7
8	OKJ38KQB8UH	Zenaida	Hines	6
9	KLI94DHV1FQ	Haley	Savage	6
10	MNV74RPO3IC	Jessamine	Buckley	6

Figure 9 - result for query 3

4.3.3.1 Details

Query three works by first executing a nested operation. This nested operation retrieves how many lessons each instructor has given by grouping lessons by instructor_id. From this data, the rows where the count is higher than a limit set in the "WHERE" statement at the end of the query. This is prettified with a non mandatory part that retrieves the first name and last name of the persons. This could be switched out for instructor_id which is already given in the lesson object, if a simpler query is wanted.

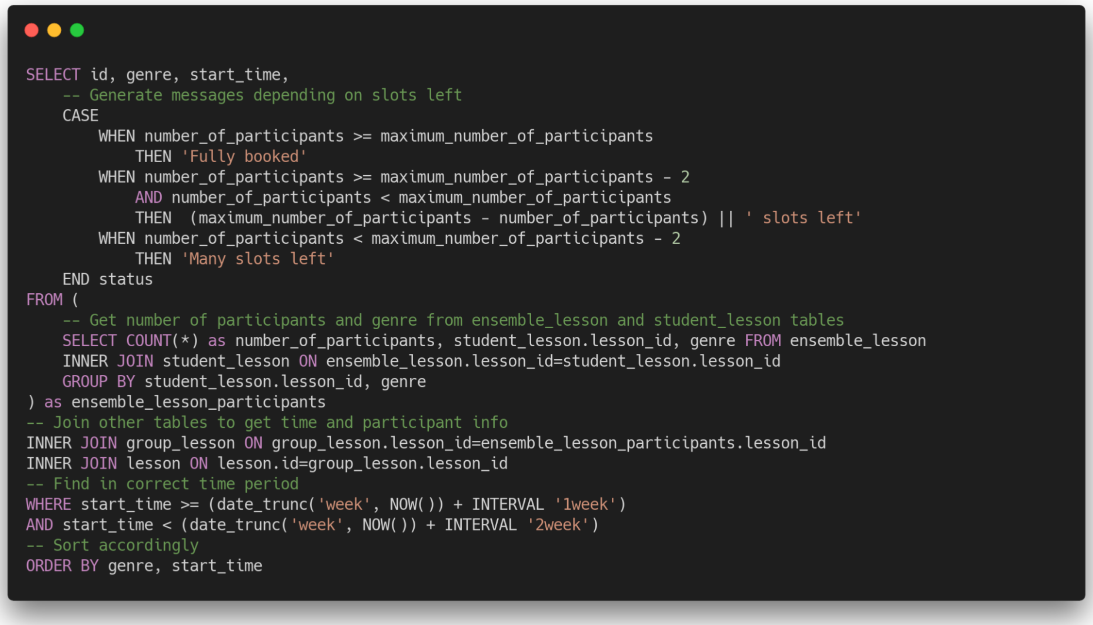
4.3.3.2 Requirements

✓ List all instructors who have given more than a specific number of lessons during the current month.

✓ Sum all lessons, independent of type, and sort the result by the number of given lessons.

4.3.4 Query 4

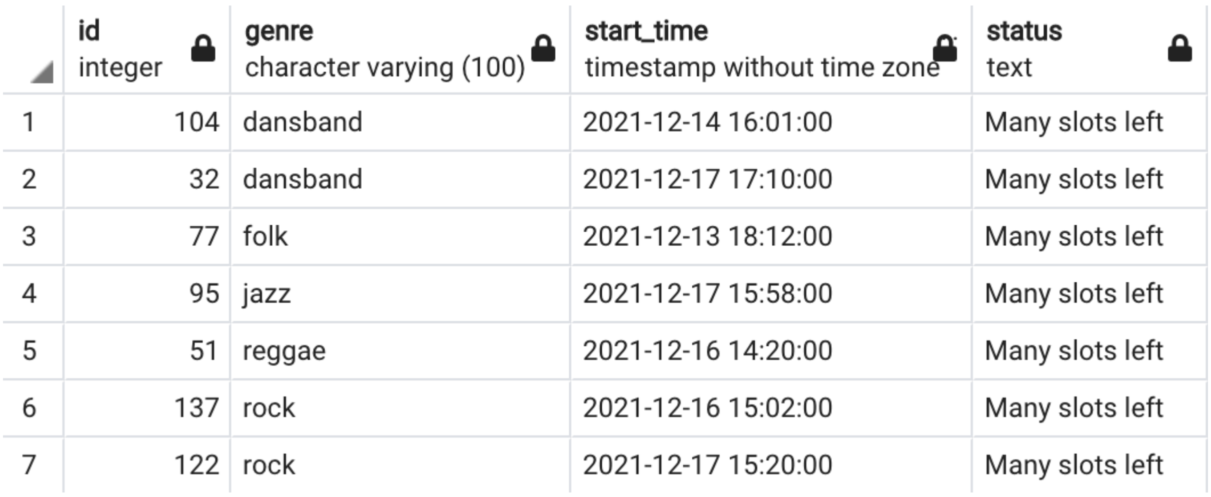
Code



```sql
SELECT id, genre, start_time,
    -- Generate messages depending on slots left
    CASE
        WHEN number_of_participants >= maximum_number_of_participants
            THEN 'Fully booked'
        WHEN number_of_participants >= maximum_number_of_participants - 2
            AND number_of_participants < maximum_number_of_participants
            THEN  (maximum_number_of_participants - number_of_participants) || ' slots left'
        WHEN number_of_participants < maximum_number_of_participants - 2
            THEN 'Many slots left'
    END status
FROM (
    -- Get number of participants and genre from ensemble_lesson and student_lesson tables
    SELECT COUNT(*) as number_of_participants, student_lesson.lesson_id, genre FROM ensemble_lesson
    INNER JOIN student_lesson ON ensemble_lesson.lesson_id=student_lesson.lesson_id
    GROUP BY student_lesson.lesson_id, genre
) as ensemble_lesson_participants
-- Join other tables to get time and participant info
INNER JOIN group_lesson ON group_lesson.lesson_id=ensemble_lesson_participants.lesson_id
INNER JOIN lesson ON lesson.id=group_lesson.lesson_id
-- Find in correct time period
WHERE start_time >= (date_trunc('week', NOW()) + INTERVAL '1week')
AND start_time < (date_trunc('week', NOW()) + INTERVAL '2week')
-- Sort accordingly
ORDER BY genre, start_time
```

Figure 9 - code for query 4

Result



	id integer	genre character varying (100)	start_time timestamp without time zone	status text
1	104	dansband	2021-12-14 16:01:00	Many slots left
2	32	dansband	2021-12-17 17:10:00	Many slots left
3	77	folk	2021-12-13 18:12:00	Many slots left
4	95	jazz	2021-12-17 15:58:00	Many slots left
5	51	reggae	2021-12-16 14:20:00	Many slots left
6	137	rock	2021-12-16 15:02:00	Many slots left
7	122	rock	2021-12-17 15:20:00	Many slots left

Figure 11 - result for query 4

4.3.4.1 Details

Query four is quite a long one. The first few rows include a “CASE” statement. This is to generate the messages based on how many slots are left. Three cases are stated in the project description, these cases generate different messages to the “status” column.

The nested “SELECT” operation is to generate the number of participants for each lesson. This is done by joining the ensemble_lesson table with the student_lesson and then grouping by the lesson’s id. This is because the join will generate a duplicate of the ensemble lesson for each entry of student_lesson containing its lesson_id. By doing this the number of participants can be generated for each ensemble_lesson.

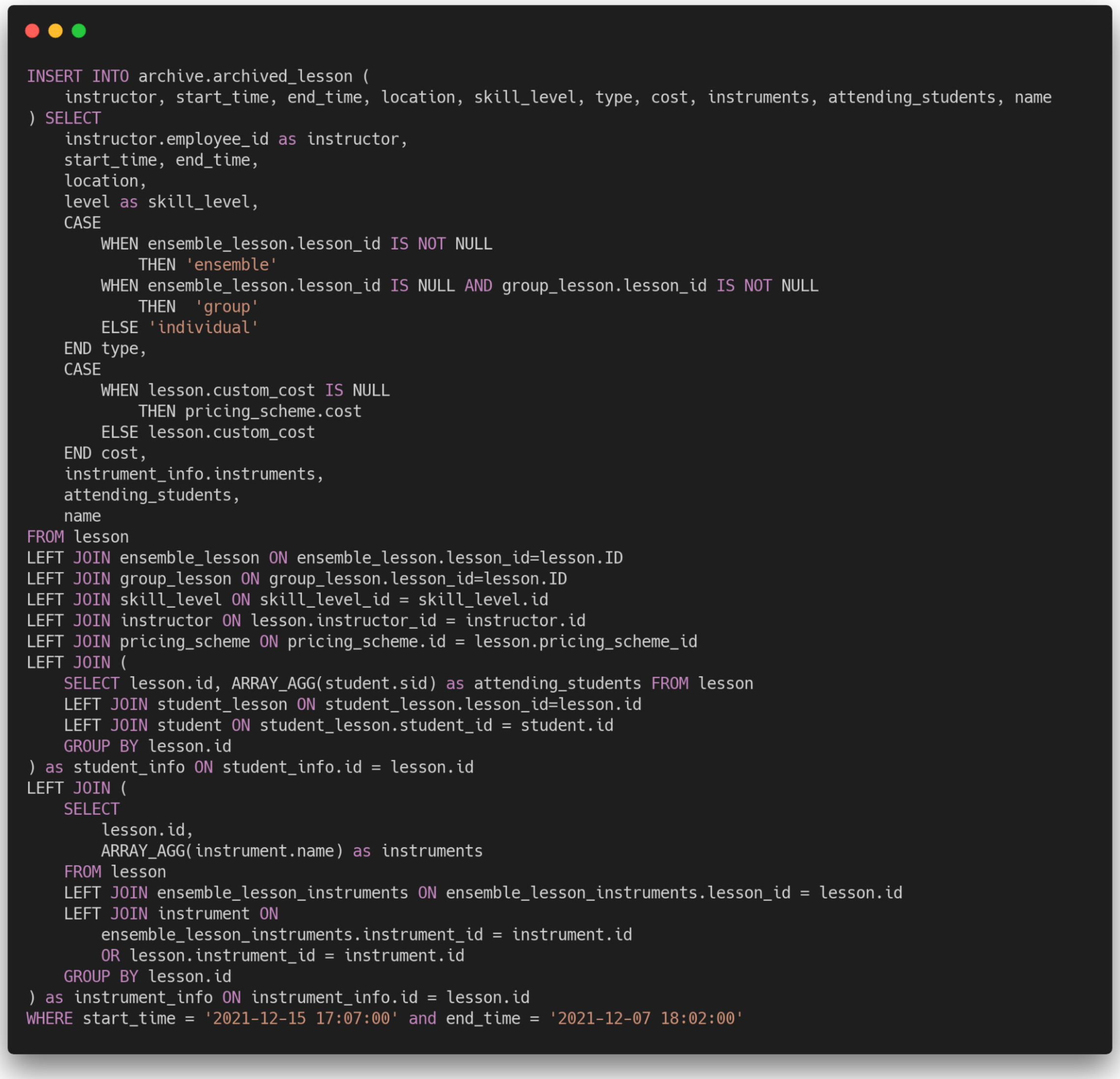
After this is done. The ensemble_lesson is joined with group_lesson for max and min number of participants and with lesson to get the starting time of the lesson. Now the lessons can be filtered out to match the period wanted. To get the period “next week”, the “NOW” postgres function is used. After this the current week is taken out using the “date_trunc” function and then an interval can be added either 1 week or 2 week for the lower and higher bound respectively.

4.3.4.2 Requirements

- ✓ List all ensembles held during the next week
- ✓ Sort by music genre and weekday
- ✓ For each ensemble, tell whether it's fully booked, has 1-2 seats left or has more seats left.

4.3.5 Historical

Code



```sql
INSERT INTO archive.archived_lesson (
    instructor, start_time, end_time, location, skill_level, type, cost, instruments, attending_students, name
) SELECT
    instructor.employee_id as instructor,
    start_time, end_time,
    location,
    level as skill_level,
    CASE
        WHEN ensemble_lesson.lesson_id IS NOT NULL
            THEN 'ensemble'
        WHEN ensemble_lesson.lesson_id IS NULL AND group_lesson.lesson_id IS NOT NULL
            THEN  'group'
        ELSE 'individual'
    END type,
    CASE
        WHEN lesson.custom_cost IS NULL
            THEN pricing_scheme.cost
        ELSE lesson.custom_cost
    END cost,
    instrument_info.instruments,
    attending_students,
    name
FROM lesson
LEFT JOIN ensemble_lesson ON ensemble_lesson.lesson_id=lesson.ID
LEFT JOIN group_lesson ON group_lesson.lesson_id=lesson.ID
LEFT JOIN skill_level ON skill_level_id = skill_level.id
LEFT JOIN instructor ON lesson.instructor_id = instructor.id
LEFT JOIN pricing_scheme ON pricing_scheme.id = lesson.pricing_scheme_id
LEFT JOIN (
    SELECT lesson.id, ARRAY_AGG(student.sid) as attending_students FROM lesson
    LEFT JOIN student_lesson ON student_lesson.lesson_id=lesson.id
    LEFT JOIN student ON student_lesson.student_id = student.id
    GROUP BY lesson.id
) as student_info ON student_info.id = lesson.id
LEFT JOIN (
    SELECT
        lesson.id,
        ARRAY_AGG(instrument.name) as instruments
    FROM lesson
    LEFT JOIN ensemble_lesson_instruments ON ensemble_lesson_instruments.lesson_id = lesson.id
    LEFT JOIN instrument ON
        ensemble_lesson_instruments.instrument_id = instrument.id
        OR lesson.instrument_id = instrument.id
    GROUP BY lesson.id
) as instrument_info ON instrument_info.id = lesson.id
WHERE start_time = '2021-12-15 17:07:00' and end_time = '2021-12-07 18:02:00'
```

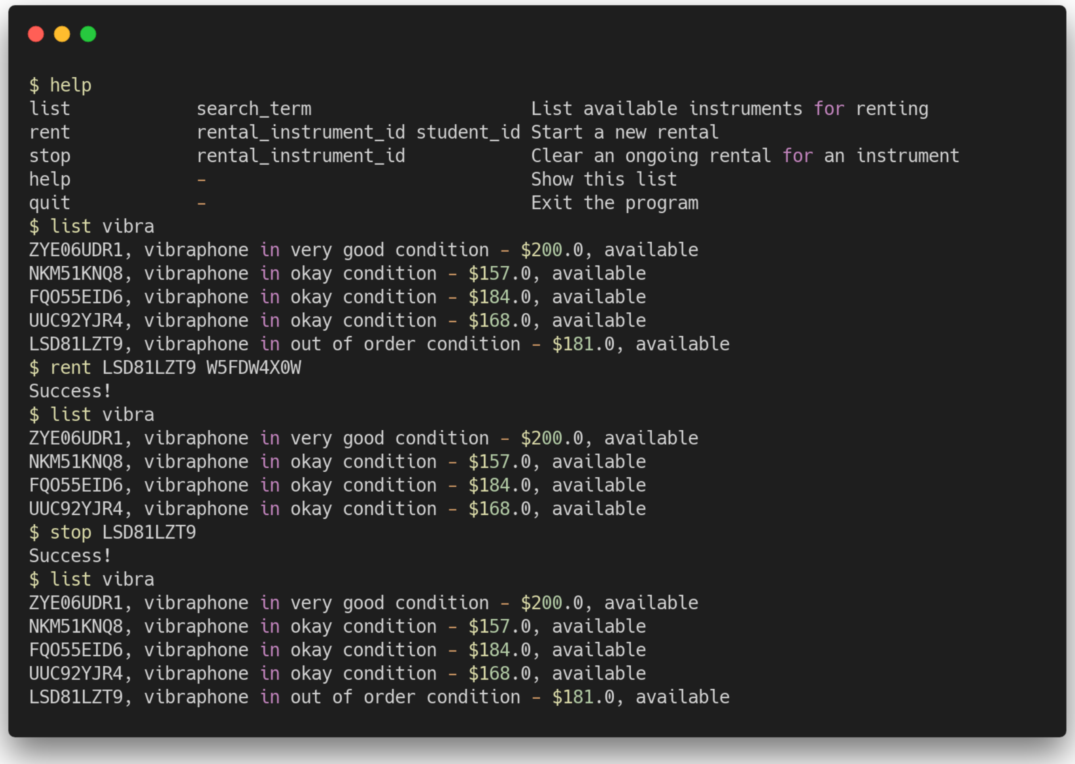
Figure 12 - Code for copying to historical database

4.3.5.1 Details

To copy data from the database to the historical database a lot of denormalization is done. Some data is removed such as the description because it was decided it was unnecessary. And some data needs to be converted to text, such as what type of lesson it is. More on why this is done in the discussion part.

4.4 Task 4

4.4.1 Sample run and user experience



```
$ help
list            search_term                     List available instruments for renting
rent            rental_instrument_id student_id Start a new rental
stop            rental_instrument_id            Clear an ongoing rental for an instrument
help            -                               Show this list
quit            -                               Exit the program
$ list vibra
ZYE06UDR1, vibraphone in very good condition - $200.0, available
NKM51KNQ8, vibraphone in okay condition - $157.0, available
FQ055EID6, vibraphone in okay condition - $184.0, available
UUC92YJR4, vibraphone in okay condition - $168.0, available
LSD81LZT9, vibraphone in out of order condition - $181.0, available
$ rent LSD81LZT9 W5FDW4X0W
Success!
$ list vibra
ZYE06UDR1, vibraphone in very good condition - $200.0, available
NKM51KNQ8, vibraphone in okay condition - $157.0, available
FQ055EID6, vibraphone in okay condition - $184.0, available
UUC92YJR4, vibraphone in okay condition - $168.0, available
$ stop LSD81LZT9
Success!
$ list vibra
ZYE06UDR1, vibraphone in very good condition - $200.0, available
NKM51KNQ8, vibraphone in okay condition - $157.0, available
FQ055EID6, vibraphone in okay condition - $184.0, available
UUC92YJR4, vibraphone in okay condition - $168.0, available
LSD81LZT9, vibraphone in out of order condition - $181.0, available
```

Figure 13 - Printout of a sample run

Above one can see how a user might navigate the program. The list command is used to see availability of certain instruments. After an instrument i found one can rent it by specifying which instrument using the instrument id and which student using the student id. To terminate a rental one can use the stop command and just specify the instrument id of the rental to be stopped.

4.4.2 ACID transactions

The database communication needs to be robust for data to be correct, and to minimize the possibility of corrupt data or unwanted errors. To make sure the code is well made it needs to have the ACID properties.

The first property is very important in this program, atomicity. This is because there are multiple updates in the database at once. For example, when a new rental is started, both the instruments availability and the rental row itself needs to be added or updated, and if something goes wrong, nothing should be saved. It is problematic if for example an instrument's availability is updated but the rental is not saved.

Consistency is also important to consider. In the program a lot of data validation is done. Firstly the user inputs are validated. Then the actual data from the database is validated too. And only if everything is fine, changes are saved to the database.

In this application isolation is another property to consider. It involves concurrent transactions and multiple users being able to do tasks at the same time. In the code for example only the rows that are going to be changed are locked for update. This is very important as the DBMS will be able to handle other updates while the java code is being executed.

Durability is the last property and is mostly a job for the DBMS. It means committed data needs to be stored to non volatile data storage in case of a system crash. This cannot be modified in the application and must be done by the DBMS.

4.4.3 MVC

The program is written in the model-view-presenter pattern (MVC). This means there are three main packages in the program. The view is the closest package to the user. It makes sure the program is easy to understand and use. In this program the view is quite simple so there is not much focus on this. The controller Connects all the other parts of the program to the view and functions like an integration or API to the view. The model contains abstract data types which are used throughout the project. This contains the types of data handled in the program.

On top of this, there are other packages to help the main three. There is a startup package for handling starting up the program. Lastly there is the integration package. This package's purpose is to integrate external sources into the program. In the case of this application it is used to query the database and to handle connecting, disconnecting and error handling.

4.4.4 The new Rental system

In this task, some changes were made to the database the design below the changes can be seen in a redone logical and physical model

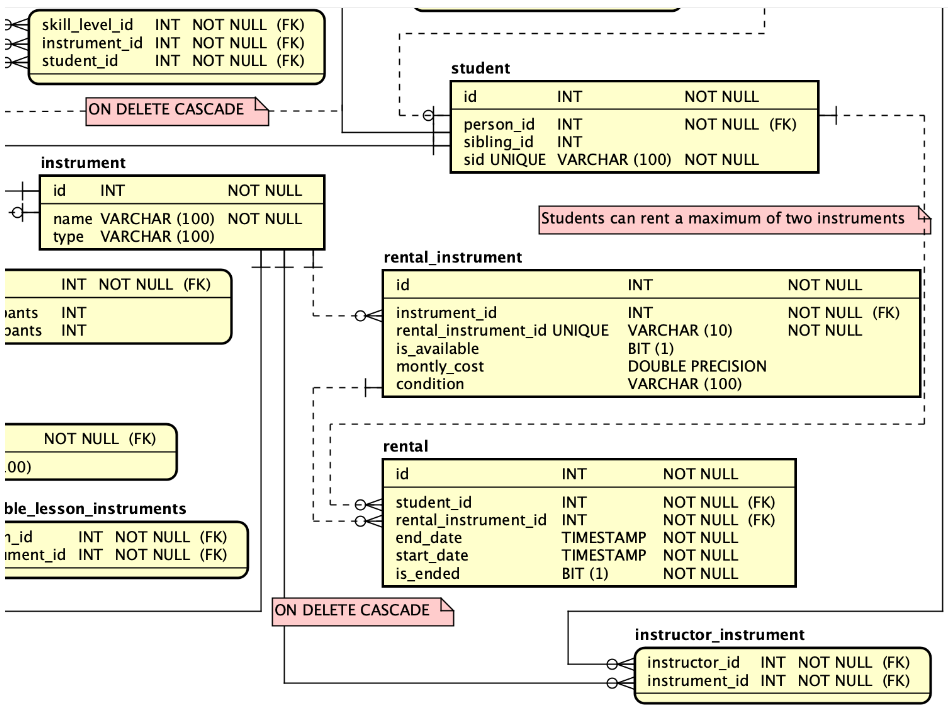

skill_level_id INT NOT NULL (FK)
instrument_id INT NOT NULL (FK)
student_id INT NOT NULL (FK)
ON DELETE CASCADE
student
id INT NOT NULL
person_id INT NOT NULL (FK)
sibling_id INT
sid UNIQUE VARCHAR (100) NOT NULL
instrument
id INT NOT NULL
name VARCHAR (100) NOT NULL
type VARCHAR (100)
Students can rent a maximum of two instruments
rental_instrument
id INT NOT NULL
instrument_id INT NOT NULL (FK)
rental_instrument_id UNIQUE VARCHAR (10) NOT NULL
is_available BIT (1)
montly_cost DOUBLE PRECISION
condition VARCHAR (100)
INT NOT NULL (FK)
INT
INT
NOT NULL (FK)
rental
id INT NOT NULL
student_id INT NOT NULL (FK)
rental_instrument_id INT NOT NULL (FK)
end_date TIMESTAMP NOT NULL
start_date TIMESTAMP NOT NULL
is_ended BIT (1) NOT NULL
INT NOT NULL (FK)
INT NOT NULL (FK)
ON DELETE CASCADE
instructor_instrument
instructor_id INT NOT NULL (FK)
instrument_id INT NOT NULL (FK)

FIgure 14 - The new rental system.

5 Discussion

5.1 Task 1

In this task it was quite difficult to know what was needed in the model as this project is quite large. What makes it even harder is that the majority of text in the project description is logic not needed in a database system.

5.1.1 Naming conventions

In the model the naming convention is followed. The used naming convention is java-like. Entities are named with upper camel case and associations and relation names are written using camel case. The names in my opinion are descriptive enough for their purpose. If some names are unclear, the context in the form of relations and their cardinality will be enough to understand what everything is.

5.1.2 UML notation

The UML notation should be properly followed, it was stated in the lectures that it was not extremely important to get the relation type completely correct, so there may be some issues. But the cardinalities and naming of the relations have a lot of thought behind them and they should be correct. There are sometimes multiple ways to relate two entities depending on how the designer understands the description.

5.1.3 Entities

The number of entities is not very large considering how much the school has to offer. But all entities have a real purpose and many other ones were considered but did not serve enough of a purpose. There should not be any missing entities as many iterations of going through everything was made but there are always different ways to model the school. The entities all also have attributes for all possible data that could be stored related to the entity. Attributes also all have a cardinality and id:s have a special unique and not null cardinality. There are also cardinalities for multiple of the same such as participants in a group lesson. Every entity has at least one relation connecting them to other relations. The relations have correct cardinalities in both ends and one side of the relation always has a linking name, for example “Person” “livesAt” “Address” where “livesAt” is a child to parent verb phrase (name).

5.1.4 Inheritance

In the model in the results chapter, inheritance is used several times. This is not the only way to do the model as shown in the model without inheritance. But there are some advantages and disadvantages using the inheritance relation. Using the “isA” relation is one alternative to the inheritance relation. The “isA” relation basically tells the reader that the child entity is a sort of the parent entity. It works the same way as the inheritance relation but it is not as readable as the inheritance relation. Using the inheritance relation gives the reader the ability to at a quick glance see that two entities are related in that way. However using the “isA” approach makes the model more consistent and some may argue, it is easier to read. Mostly this comes down to personal preference

Another way to remove an inheritance relation is to remove the relation entirely. This can be done when both the generalized and specialized entities are useful in the model. The “Instrument” and “RentalInstrument” entities are connected in the initial model with an inheritance relation but this can be removed. The “RentalInstrument” entity can simply have the same attributes as the instrument. This is mostly a bad idea but it can be done in certain situations. The reason to avoid this is because it creates duplicate data which is most of the time a bad idea.

As a conclusion, the inheritance relation is not really needed but gives the model a more readable and structured design.

5.2 Task 2

After completing task 2 the program feels much more like a database and all the tables can now make sense.

5.2.1 Tables and columns

In the model the new naming convention is followed. The new naming convention used in SQL and databases is snake case. Snake case separates words with underscores so the conceptual model entity “SkillLevel” is now called “skill_level”. Snake case also implies that all letters should be lowercase. The names of columns in the tables and the tables themselves should have sufficiently explaining names. The names are mostly written out and not shortened. The attributes are also mostly self explanatory to anyone who has read the project description as the terms used as names are often taken directly from the text.

All tables in the model have everything required for 3NF. Surrogate keys are mostly used as primary keys. The surrogate key makes it easier to normalize as there is only one primary key. Starting from the conceptual model the address entity has been added into the person table as that was the only relation for it. This process can hurt normalization but in this case it is okay as the person's only primary key is “id” and the purpose of the person is to hold data directly related to the person and that includes the person’s address.

All tables in the database are relevant because their purpose is directly specified in the project description. This is true for all tables but the “ensemble_lesson_instruments” table. This may not be necessary as it is not directly stated in the text but is rather a matter of interpretation of the project description, this may be true for other tables as well. There should not be any tables missing as the project description is now carefully considered twice now.

The columns in the tables are also taken directly from either the conceptual model or the project description. they should hold all the data necessary to run the system the description is asking for. Some columns were added when transitioning from the conceptual model because a clearer view of the actual database that is being created is understood. The data types in the columns and their constraints are new for the logical and physical model. The data types, especially the varchar ones and their length is mostly preference. The length was decided by trying to come up with the most extreme case and adding a bit more. The not null constraint was added to the columns where it seemed necessary, meaning the program will not function correctly if the attribute is null. The column types are correct.

Primary keys in the model are mostly made up of surrogate keys. This is because it is a safe choice and it leaves a lot of room for customisation and changing values. This may not sometimes be the most efficient option but it is the safest. Especially now when there is not a customer available to talk to and discuss future plans and possible changes with.

5.2.2 Relations

The crow foot notation is followed correctly in the model. The relations themselves resemble the ones in the conceptual model. There the crow foot notation was considered carefully in task one so it should be correctly followed. The relations should be correctly specified with the right cardinality with many-to-many relations having a dedicated table and one-to-many relations connected to it.

5.2.3 Inheritance

In a logical and physical model there are four options for using inheritance. The first option is what is used in the model in the results chapter. This applies to the lesson table. Here the primary key is used in specialized tables like ensemble lesson and group lesson. This is one of multiple ways of doing it. In the course literature there are three other ways of doing it.

The second option was used in the conceptual model with inheritance. It involves creating two seperate tables and having matching names for the ones columns they share. This is a great alternative if both specialized tables have many unique columns and a generalized table would not include many columns. In this database this is not the case and therefore the first option is a greater alternative. There is also the downside with it being easy to botch as a simple spelling mistake can make querying and organizing hard.

The third option involves adding a type column and adding all the other special columns to the original, general table. This could be an alternative to the first option used in the model. However the fact that this includes setting data to null in the designing process disqualifies it for this database as it is easily avoidable. The same goes for the fourth option. This option includes setting flags to true or false depending on which type the row is specialized into. This way seems quite complicated and more like a workaround. On top of this, these options hurt normalization, while it could still be argued to be in 3NF it is much more borderline than the other two options.

As a conclusion on the inheritance part, option one and two are useful in different cases, three and four however should most likely be avoided.

5.2.4 Summary

The model presented in the result section should be able to cover all needs presented in the project description of the Soundgood Music School. Some rules are not possible to model so they are written in plain text as a note inside the model. This is also the case for column constraints and in the real database that kind of logic is implemented manually if possible.

There are a few changes in the model to the original conceptual model but this is just a part of the development process. Everything gets clearer as the project is further worked on.

5.3 Task 3

Getting to the point of the SQL query in the result took numerous iterations and rewriting parts of the code. This seems to be the best way to write SQL queries, to take small steps and then combine them after they work individually. This also helps to make sure the queries are valid as they are tested a lot.

5.3.1 View or materialized view

The queries also need to be easily accessible for them to be useful. This is where views come in. It is a way to store the query and then a developer can query the view instead of copying all the SQL code in the query itself. There are two options for views: they could either be materialized or not. A materialized view is updated when data in the database is changed, therefore it is speedy when querying. A non-materialized view however only stores the query and executes it when queried. Both alternatives are useful and both serve a purpose.

A normal view is used in this project for queries which are executed rarely. These are the OLAP queries, the ones used by administrative staff. While they probably do not like to wait for a query of this size will most likely take a couple of milliseconds which is unnoticeable. What is improved however is data insert and update times because no materialized view is updated when data is changed.

The only materialized view is the one used in the Soundgood website. The result of this query is updated every time data is changed in a table used by the query. The reason for this being materialized is because a fast loading time on websites is vital for a good user experience.

5.3.2 Denormalization

When creating the historical database many changes were made to the columns in the lesson table. This is mainly because the historical database does not have the same purpose as the main one. The historical data will most likely never be changed, only new data will be inserted. Because of this the table is *Denormalized*. This means for the most part that tables are combined. Almost all the lesson related data is denormalized into a single table. This process can be quite difficult and a lot of special SQL functions are needed. The lesson type for example is converted into strings instead of joining and excluding as in the main database schema. The dynamic solutions like pricing schemes are converted to a static cost which is set from either the custom cost or the pricing scheme cost.

The advantage of doing all this denormalisation is that when querying the historical database, no joins should be needed. This both makes it easy and helps performance as historical data can get quite large after years of use. The disadvantage however is that changing data can be quite the hassle. For example inserting a new student to a lesson goes from adding a simple row in a table to finding the historical database lesson row and adding an id to an array inside the row. Also joining with other database data can be slower as no foreign keys exist. The joining will have to match other ids which may have been changed. Overall working with the denormalized data becomes quite difficult.

5.4 Task 4

The application presented in the results chapter works as intended and described in the task description it meets all the requirements while keeping the architecture and conventions.

5.4.1 Naming and architecture

The code in the application follows the MVC architecture and the java naming conventions. This is to make sure the code is as easy to follow as possible. Another developer should be able to navigate the program without having to ask the original developer what the purpose of certain code is. The naming convention is camel-case for variables and functions and upper-camel-case for class definitions. The names themselves follow a pattern and are descriptive enough. The MVC architecture is followed as described in the result chapter, and all this is to make the code easy to understand. Duplicated code is also kept to a minimum with numerous helper functions and finalized variables.

5.4.2 Transactions

The ACID transaction properties are widely considered in the program. All the four properties are considered and the program has code related to the first three (see results chapter). The transactions are rolled back if an error occurs during the execution of the program. When a function is finished the updates are committed. Locks are used to make sure only one process is editing specific data at the time. Error handling is also taken into account to make sure the user is presented with information if anything goes wrong.