Project Report - Soundgood Music School

Data Storage Paradigms, IV1351

Casper Kristiansson, Casperkr@kth.se

2022-01-10

Contents

1 Introduction

Building a database without structure will most likely end up failing or being extremely bad. In this project the author will structurally build a database for the Soundgood music school. The author solved the task by building a conceptual model, logical model and then translating it to SQL code to create the database itself. The database can perform various of different tasks such as managing the school´s instruments, lessons, students, and instructors. A command line interface will be developed to manage some aspects of the database. The author will be discussing and motivating each step of designing the database and how each step of the project was solved. Karolina Sjökvist also contributed to this project.

2 Literature Study

Task 1

Initially, the author went through course lectures on how to create a domain model and an example of a conceptual model. The author gained an understanding of the concept of a conceptual model and how to create one using various steps. This knowledge provided the author with the tools necessary to solve Task 1.

Task 2

The next task consisted of creating a logical or a physical model. First, the author watched the lecture on "Logical and physical models," which covered all the steps from converting a conceptual model to a logical model. The lecture consisted of what steps is required to converting a conceptual model to a logical or physical model. Second, the author read chapter 14 in the book *Fundamentals of Database Systems Seventh Edition,* Pearson (Elmasri & Navathe, June 8th 2015) which was used to detect and learn whenever a relation database scheme was good or bad. Third, the author applied that knowledge and was able to improve the model. For the SQL commands such as create the database and create "fake" data, the author reviewed the lecture on "SQL - The Structured Query Language."

Task 3

Task 3 involved creating different types of queries which would be performed on the database. To solve this task the author watched the lecture on "SQL - The Structured Query Language" which covered basic SQL commands. But to get enough information to solve the task the author also read the chapters 6 and 7 in the book Fundamentals of Database Systems which covered a range of basic to advanced SQL commands.

Task 4

The last task consisted of creating a programmatic access to the database which would cover the renting instrument part of the database. The author solved the task by primarily watching the lectures on "Database Applications" which consisted of accessing the database using Java and performing various number of queries on the database. In the lecture about "Architecture and Design of a Database Application" the author was able to get sufficient knowledge on how to create a programmatic access program to the database using the MVC pattern.

3 Method

Conceptual Modeling

The goal of this task is to create a conceptual model for a music school. The author created the conceptual model by using the modelling program Astah. The model itself consist of an ER diagram (Entity-relationship model). Designing a conceptual model consist of five steps:

1. Noun Identification
2. Category lists
3. Remove unnecessary entities
4. Find attributes
5. Find relations

The first step was to identify the nouns that consists in the description of the music school. The nouns that the author found would either become an entity or an attribute in the model. After the author finished that step, the second step was to identify more nouns using a categories list which can be found in the book "Object Oriented Development," chapter 4. Using this method, the author can make sure that the model will consist of all the nous needed for the music school. The third step of the process was to remove unnecessary entities or rename them with better nouns.

The fourth step of the process is to find the attributes for each entity. Each entity that exists will either be a string, Boolean, number, or timestamp. It is also important to add the cardinality and if the attribute is allowed to be without value. The fifth and last step of designing the model is to declare and find relations between entities. The entities could either have an identifying, non-identifying, or many-to-many identifying relation.

Logical and Physical Model

The next task of the project is to create a logical or a Physical model from the conceptual model that was created in the previous task. The author chooses to create a logical model. Converting a conceptual model to a logical model consist of six steps:

1. Deciding which relations should be kept
2. Specifying column types
3. Adding column constraints
4. Making sure that the model is normalized and that all different database operations can be performed on the model
5. Specifying the different attributes
6. Relations (many-to-many, one-to-many)

The first step consists of deciding which relations should be kept in the model. The author started by creating a table for each entity from the conceptual model. The author then created the columns which consisted of a cardinality of 0..1 or 1..1. The attributes which consist of more than one cardinality will have their own table. For example, instrument, sibling, student phone, student email. These tables will be connected using either a one-to-many or many-to-many relations. For example, the author created a many-to-many relations between student and lesson. This is because there are multiple students on each lesson and each student can have multiple lessons. But for student email, student phone the author created a one-to-many relationship because it should be possible for each student to have registered multiple contact details.

The second step was to specify each column type. In PostgreSQL there is a SERIAL pseudo-type which is used on ids. Each table consists of a PK (primary key) which primarily is used to find specific rows in the table, to be consistent the author chooses to have the ids of each table as the primary key. Otherwise, the rest of the values in the table would either be a VARCHAR or a INT. The author also thoroughly decided which column should consist of NOT NULL elements, for example you cannot add a person or instructor without adding a person number.

In the third step, the author added column constraints to the model. It is important to add constraints to columns to decide what to do if certain actions are performed. For example, what should happen if someone removes an instructor id from a lesson or if someone removes a phone number from a student. Each of these specifications can be created using commands like ON DELETE CASCADE, ON DELETE SET NULL or ON DELETE NO ACTION. For the different ids in lesson the author decided to add SET NULL ON DELETE because if a lessons price type, instructor, or room for some reasons should be removed the entire lesson should not be removed from the database. But if someone decides to remove a phone number to a student it would be unnecessary to store just the student id and therefore the author uses ON DELETE CASCADE.

The fourth step is to make sure the model is normalized. The author followed to create the logical model was to make sure that the model follows all the different database operations. This required that the author needed to make sure that all the upcoming tasks like searching for lessons could be performed on the model. It was also important to try and reduce data redundancy and improve data integrity by making sure that we do not store unnecessary amount of data or that it exists unnecessary relations.

The fifth step was to add all the different attributes to the model. This could simply be done by moving the different attributes from the conceptual model to the logical model. But as mentioned in step one some of the attributes became their own many-to-many relation. The last step of the task was to add the different relations between the entities. Because some of the attributes became entities new relations was needed to be established.

The additional step of this task was to create a SQL script to create the database and a script to create "fake" data.

SQL

The next task of the project was to create SQL queries to manage different tasks that will be performed on the data in the database. Those queries include:

- Number of lessons per month during a specific year
- Number of different lesson types per month
- Average number of lessons for each month
- Count the number of lessons different instructors has given
- List ensembles that will be held upcoming week and mark them if they have 1-2 seats left, or more seats left

When designing the database, the author chooses to use PostgreSQL as the database management system. The author uses pgAdmin which is develop for PostgreSQL to run the different queries and to view the different tables and views. Before the author was running the different queries in the database, they were developed in the code editor visual studio code.

The goal of the task is to create a view for each of the different queries which can be done using either a materialized or a normal view. When running a query using a view command will result in a logical view being created which "looks" like a normal table. The view will consist of the commands specifications and therefore can be used to verify that the command performs the right action on the data. For example, if the goal is to verify that the first command "number of lessons per month during a specific year" works, the author needed to manually check the number of lessons that were given a couple of months during the year. It can later be verified that both the manual check and the view will consist of the same data and therefore the query is valid.

Programmatic Access

The next task involved creating a command line user interface to oversee functionalities for renting instruments. Those functionalities include listing all available instruments, rent an instrument and terminate active rentals. The author decided to solve this task using the Model View Control (MVC) design pattern. The author followed these steps when solving the task:

1. Creating a database connection
2. Setting up SQL prepare statements (queries)
3. Building the structure for the MVC design pattern

The first step of solving the task is to create a connection to the local Soundgood music school database. To complete this step, it was necessary to download the library PostgreSQL JDBC driver. After establishing a connection to the database, it was possible to execute different SQL queries. The second step is to create prepare statements which holds a SQL query and is precompiled, which than can efficiently be used to execute a query multiple times using the program (PreparedStatement Java Platform SE 7, 2021).

For the third and last step, the author continued solving the task by creating the different package for the MVC structure which followed the naming convention "main.java.se.kth.iv1351.soundGoodMusicSchool." Additionally, the author divides the program into the different sections: controller, integration, model, startup, view. The integration layer would hold the methods which would execute SQL queries on the database. Several types of objects would be stored in the model layer such as instruments which holds the distinctive characteristics of an instrument such as price, brand, and type. The controller layer task would not be managing any logic but rather make the correct function calls from the other layers. The last layer, view, exists to represent the user with a command line interface, which the user could use to choose to either list, rent or terminate a rental of and instrument.

4 Result

Conceptual Modeling

The author solution to the conceptual model can be seen in figure 3.1. The conceptual model main structure is built around "Student" and "Instructor" entity. The "Lessons" entity is used for inheritance for each of the lesson types ("Ensembles", "Individual Lessons", "Group Lessons"). Each lesson is also given an instructor, room, and price scheme IDs. The author has added NOT NULL and the specific cardinality for all the different attributes.

A student can rent instruments from the renting instruments entity. The renting instruments entity keeps track of all the active rentals and a history of them. The model that the author designed is able perform all the required business transactions.

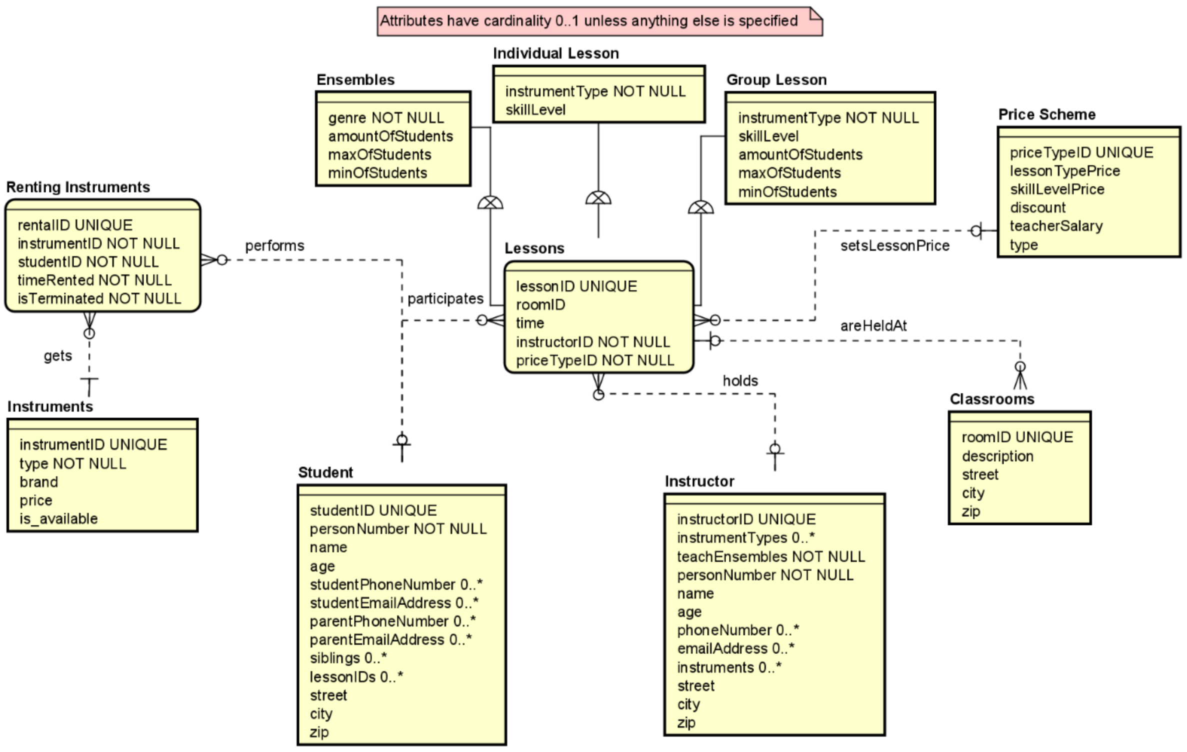


Figure 3.1: The author solution to the conceptual model.

Logical and Physical Model

The solution to the logical model can be found in figure 3.2. In the figure it can be seen that it follows the same structure as task one where it focuses on the student and instructor. The author created the required many-to-many relations for example student_lesson for which consist of all the different lessons a student is signed up. To represent the 0..* cardinality it was added a one-to-many relationship between instructor and instrument. The author chose to replace the inheritance and merged them together to entity lesson.

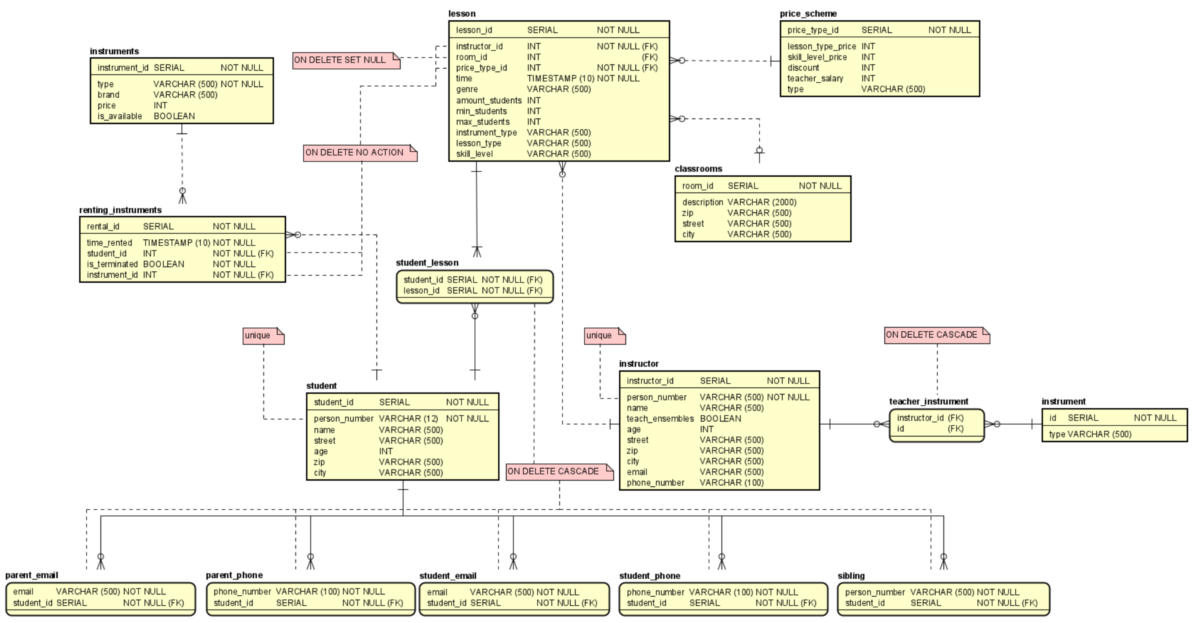


Figure 3.2: The author solution to logical model.

GitHub link to both the SQL script to create the database and a script for the data input: https://github.com/CasperKristiansson/Data-Storage-Paradigms-IV1351/tree/master/Task%202

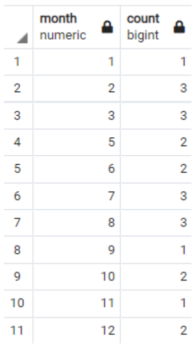
SQL

Each solution for the queries will be shown below in the different figures.

The first query that was created was the view for counting the number of lessons given per month during a year. As seen in figure 3.3 the author creates a view called "lesson_count_month" which selects all the different month from time and count the number of lessons that fulfills the requirement of the year and month. By using the group command is used to arrange the data into groups by its month to match the EXTRACT (month FROM time), otherwise the count command would only return a number for the entire year. In the first extract command we also specify that the month column name should be named month.

```
CREATE VIEW lesson_count_month AS
    SELECT
    EXTRACT(month FROM time) AS month,
    count(*) FROM lesson WHERE EXTRACT(YEAR FROM time) = '2021' GROUP BY EXTRACT(month FROM time)
    ORDER BY EXTRACT(month FROM time) ASC;
```

Figure 3.3: The SQL query for creating a view and counting the number of lessons given per month during a specific year.



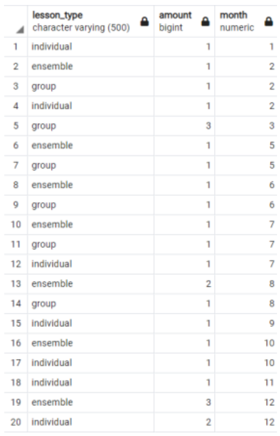
	month numeric	count bigint
1	1	1
2	2	3
3	3	3
4	5	2
5	6	2
6	7	3
7	8	3
8	9	1
9	10	2
10	11	1
11	12	2

Figure 3.4: The view that is created after running the query from figure 3.3

The next goal is to create a view for the number of different lessons that exists each month during a specific year which can be seen in figure 3.5. This can be achieved using the COUNT(*) command and group them using MONTH by using the EXTRACT command. The view that gets created can be seen in figure 3.6.

```
CREATE VIEW lesson_count_type_month AS
    SELECT lesson_type ,COUNT(*) as amount ,EXTRACT(MONTH FROM time) as month from lesson
    WHERE EXTRACT(YEAR FROM time) = '2021'
    GROUP BY month, lesson_type;
```

Figure 3.5: The SQL query for creating a view and counting the number of lessons for each lesson type



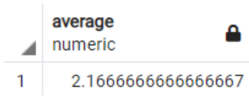
	lesson_type character varying (500)	amount bigint	month numeric
1	individual	1	1
2	ensemble	1	2
3	group	1	2
4	individual	1	2
5	group	3	3
6	ensemble	1	5
7	group	1	5
8	ensemble	1	6
9	group	1	6
10	ensemble	1	7
11	group	1	7
12	individual	1	7
13	ensemble	2	8
14	group	1	8
15	individual	1	9
16	ensemble	1	10
17	individual	1	10
18	individual	1	11
19	ensemble	3	12
20	individual	2	12

Figure 3.6: The view that is created after running the query from figure 3.5

By using the EXTRACT command from the query in figure 3.9 it is possible to check whenever a lesson fulfills the requirement of a specific year. Then its only necessary to divide the count by 12.0 to get an average number of lessons given that year per month. The view that gets created can be seen in figure 3.10 which consist of a float number to be the average number of lessons.

```
CREATE VIEW lesson_average_year AS
    SELECT count(CASE WHEN EXTRACT(YEAR FROM time) = '2021' THEN 1 ELSE NULL END)/12.0 as average FROM lesson
```

Figure 3.7: The SQL query for creating a view and counting the number of lessons given a year and dividing it by the amount of months

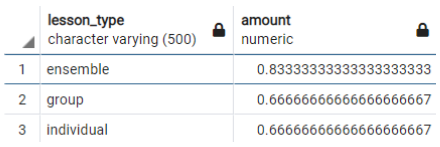


	average numeric
1	2.1666666666666667

Figure 3.8: The view that is created after running the query from figure 3.7

```
CREATE VIEW lesson_average_type AS
    SELECT lesson_type, COUNT(*)/12.0 as amount from lesson
    WHERE EXTRACT(YEAR FROM time) = '2021'
    GROUP BY lesson_type;
```

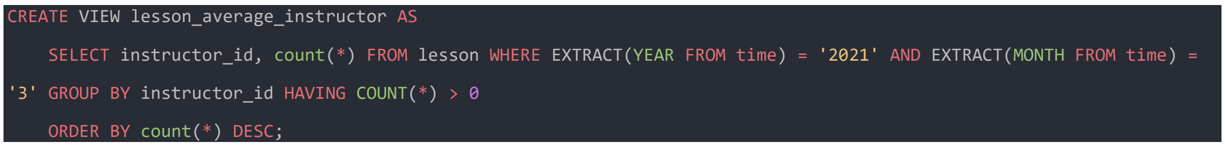
Figure 3.9: The SQL query for creating a view and counting the number of lessons given in a year for the different types of lesson



	lesson_type character varying (500)	amount numeric
1	ensemble	0.83333333333333333333
2	group	0.66666666666666666667
3	individual	0.66666666666666666667

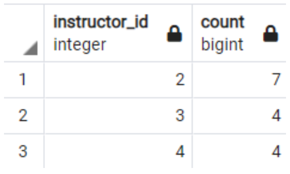
Figure 3.10: The view that is created after running the query from figure 3.9

The next SQL query is to count the number of lessons a teacher has given and if that count is bigger than a specific number it should be shown. The view can be used to figure out what teachers might be overworking themselves. The query uses as before EXTRACT to figure out that the lesson is during the year 2021 and the month three. Then the result is grouped by instructor id but only instructor ids which has a count bigger than three. In the view it can be seen that the teacher with the id 2 has given seven lessons during 2021.



```
CREATE VIEW lesson_average_instructor AS
    SELECT instructor_id, count(*) FROM lesson WHERE EXTRACT(YEAR FROM time) = '2021' AND EXTRACT(MONTH FROM time) =
'3' GROUP BY instructor_id HAVING COUNT(*) > 0
    ORDER BY count(*) DESC;
```

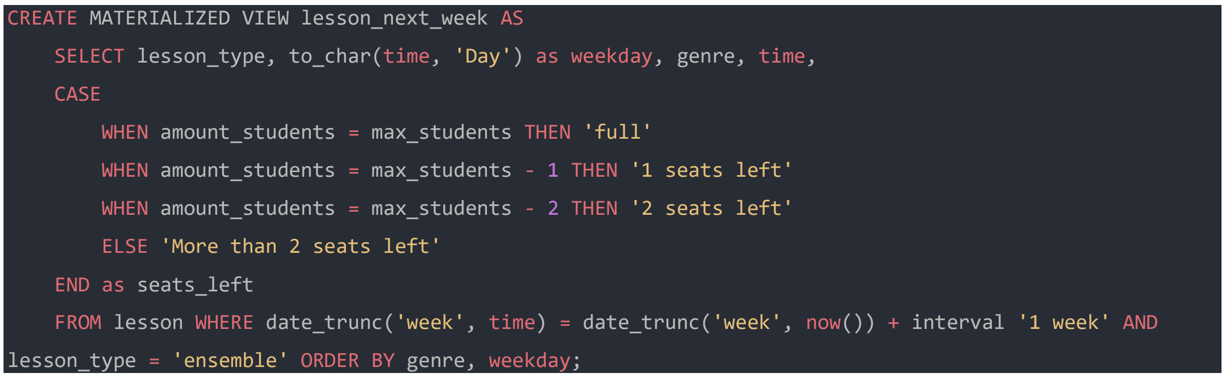
Figure 3.11: The SQL query for creating a view and counting the number of lessons given a year and dividing it by the amount of months



	instructor_id integer	count bigint
1	2	7
2	3	4
3	4	4

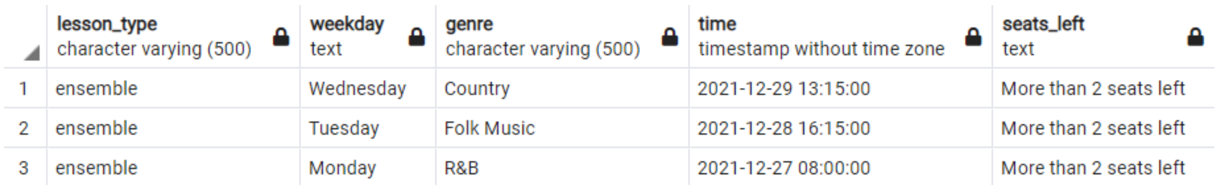
Figure 3.12: The view that is created after running the query from figure 3.11

The last query that was needed to be solved was to figure out what the next weeks ensemble lessons and mark them with a specific number of seats left. For example, if only one seat remains the column should state that. This can be completed using CASES. In the materialized view below the data is sorted by genre and then the weekday. It is possible to automatically determine what lessons are given next week using data_trunc which can add weeks to a certain date.



```
CREATE MATERIALIZED VIEW lesson_next_week AS
    SELECT lesson_type, to_char(time, 'Day') as weekday, genre, time,
    CASE
        WHEN amount_students = max_students THEN 'full'
        WHEN amount_students = max_students - 1 THEN '1 seats left'
        WHEN amount_students = max_students - 2 THEN '2 seats left'
        ELSE 'More than 2 seats left'
    END as seats_left
    FROM lesson WHERE date_trunc('week', time) = date_trunc('week', now()) + interval '1 week' AND
lesson_type = 'ensemble' ORDER BY genre, weekday;
```

Figure 3.13: The SQL query for creating a materialized view and checking which ensembles lessons are given the next week and the number of seats left.



	lesson_type character varying (500)	weekday text	genre character varying (500)	time timestamp without time zone	seats_left text
1	ensemble	Wednesday	Country	2021-12-29 13:15:00	More than 2 seats left
2	ensemble	Tuesday	Folk Music	2021-12-28 16:15:00	More than 2 seats left
3	ensemble	Monday	R&B	2021-12-27 08:00:00	More than 2 seats left

Figure 3.14: The materialized view that is created after running the query from figure 3.13

GitHub link for the different queries:

https://github.com/CasperKristiansson/Data-Storage-Paradigms-IV1351/tree/master/Task%203

Programmatic Access

The author chose to represent the result of this task by selecting examples and important parts of the program.

In figure 3.15, the author developed a method called *accessDB()* which can establish a connection to the local Soundgood Music School database.

```
private void accessDB() {
    try {
        Class.forName("org.postgresql.Driver");
        this.connection = DriverManager.getConnection("jdbc:postgresql://localhost:5432/SoundGoodMusicSchool",
"postgres", Login.PASSWORD);
        connection.setAutoCommit(false);
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
}
```

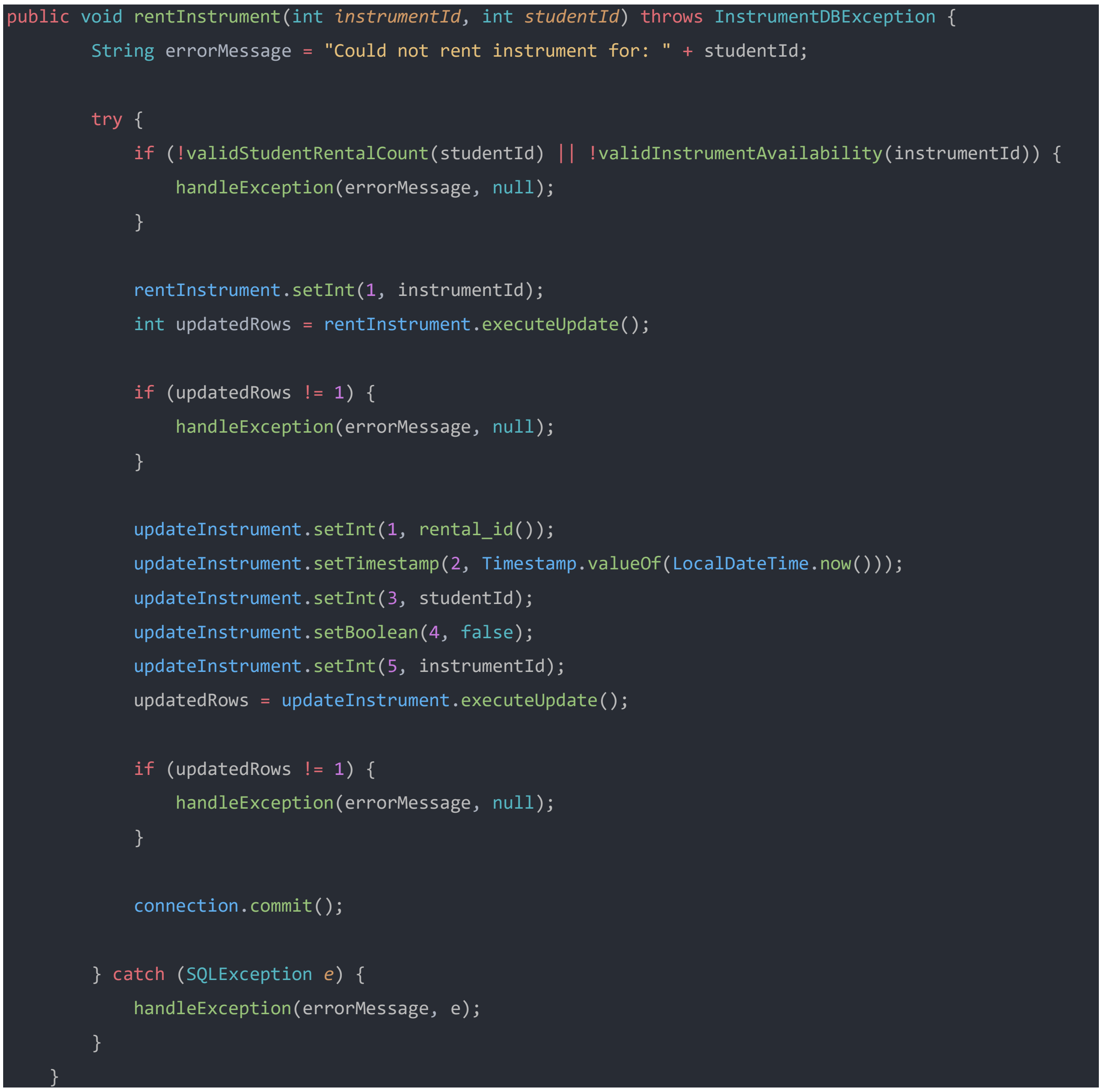
Figure 3.15: The method for establishing a connection to the database

The figure 3.16 consist of one of the prepare statements which is used in the program. The example command performs the SQL query which selects the number of rows for a specific table which in this case is the rent_instrument table.

```
getNumberOfRentals = connection.prepareStatement("SELECT COUNT(*) FROM " + RENT_INSTRUMENT_TABLE_NAME);
```

Figure 3.16: Example of a prepare statement which is a precompiled SQL command

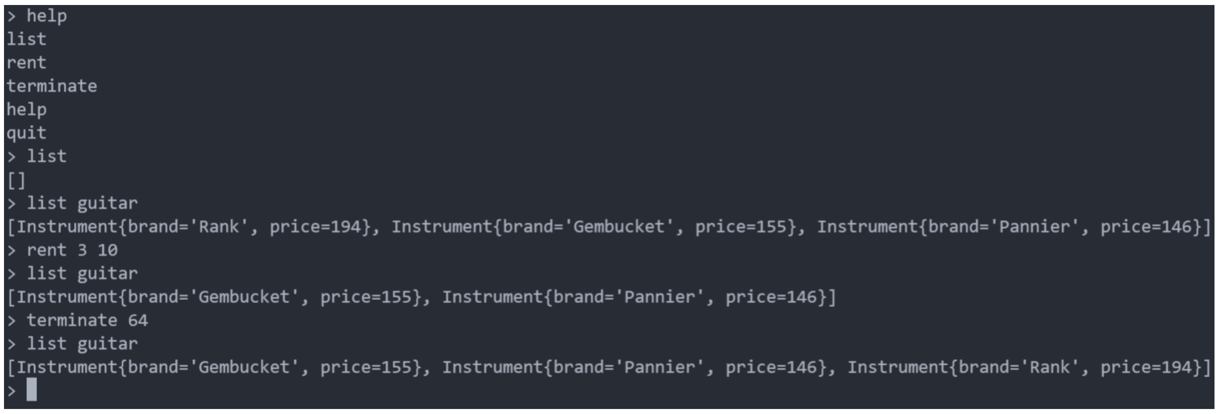
The integration layer will consist of the methods which will execute queries. The figure 3.17 contains the method for renting an instrument. When renting an instrument two queries needs to be executed. The first one needs to mark an instrument as occupied and the other one needs to create a new rental. The method also consists of calling a method called validStudentRentalCount which checks if a student has already reached its maximum of number of active rentals.



```java
public void rentInstrument(int instrumentId, int studentId) throws InstrumentDBException {
        String errorMessage = "Could not rent instrument for: " + studentId;

        try {
            if (!validStudentRentalCount(studentId) || !validInstrumentAvailability(instrumentId)) {
                handleException(errorMessage, null);
            }

            rentInstrument.setInt(1, instrumentId);
            int updatedRows = rentInstrument.executeUpdate();

            if (updatedRows != 1) {
                handleException(errorMessage, null);
            }

            updateInstrument.setInt(1, rental_id());
            updateInstrument.setTimestamp(2, Timestamp.valueOf(LocalDateTime.now()));
            updateInstrument.setInt(3, studentId);
            updateInstrument.setBoolean(4, false);
            updateInstrument.setInt(5, instrumentId);
            updatedRows = updateInstrument.executeUpdate();

            if (updatedRows != 1) {
                handleException(errorMessage, null);
            }

            connection.commit();

        } catch (SQLException e) {
            handleException(errorMessage, e);
        }
    }
```

Figure 3.17: The method for renting an instrument

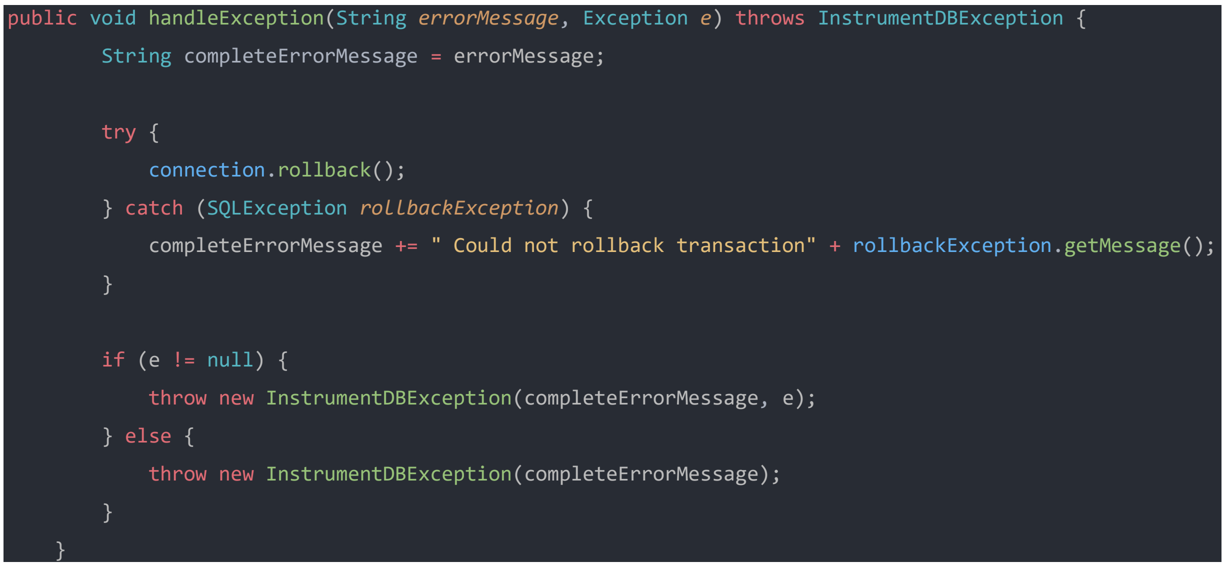
In figure 3.18 a sample printout of the program can be seen. It shows all the different commands that can be executed. That includes listing all commands, listing all instruments, renting, and terminating a rental.



```
> help
list
rent
terminate
help
quit
> list
[]
> list guitar
[Instrument{brand='Rank', price=194}, Instrument{brand='Gembucket', price=155}, Instrument{brand='Pannier', price=146}]
> rent 3 10
> list guitar
[Instrument{brand='Gembucket', price=155}, Instrument{brand='Pannier', price=146}]
> terminate 64
> list guitar
[Instrument{brand='Gembucket', price=155}, Instrument{brand='Pannier', price=146}, Instrument{brand='Rank', price=194}]
> 
```

Figure 3.18: Printout of a sample run of the program

To deal with the exception handling of the program and in the case that some queries might fail, the author tries to achieve good ACID transaction handling using queries rollback. In figure 3.19 the author tries to rollback the quires incase if something goes wrong.



```java
public void handleException(String errorMessage, Exception e) throws InstrumentDBException {
        String completeErrorMessage = errorMessage;

        try {
            connection.rollback();
        } catch (SQLException rollbackException) {
            completeErrorMessage += " Could not rollback transaction" + rollbackException.getMessage();
        }

        if (e != null) {
            throw new InstrumentDBException(completeErrorMessage, e);
        } else {
            throw new InstrumentDBException(completeErrorMessage);
        }
    }
```

Figure 3.19: Exception handling using rollback

GitHub link of the program:

https://github.com/CasperKristiansson/Data-Storage-Paradigms-IV1351/tree/master/Task%204

5 Discussion

Conceptual Modeling

When solving the task, the author made sure to punctually follow all the steps and guidelines when creating a conceptual model. The author made sure to follow the general naming convention for all the entities and attributes as well giving each a sufficient explaining name. During this step it was also important to have a reasonable number of entities and attributes which the author made sure to thoroughly examine if each needed to exist. For example, the author removed and remodeled the instrument entities because it had unnecessary data stored in both renting instrument and instruments.

When designing the conceptual model, the author made sure to follow the guidelines for the UML notation correctly. Each attribute in the diagram has a type and a cardinality specified. When solving the last step of the task the author made sure that the entities relations are relevant and follow the desired business rules.

The author chooses to use inheritance to represent the different lesson types. In figure 3.1, a main lesson table exists which holds the main attributes of a lesson such as lesson id, instructor id and classroom id. After that, the different lesson types, ensemble, group, and individual will inherit all the information from the lesson table. Using this method will avoid in redundant data rather than keeping all lesson data in one table. A disadvantage of using inheritance is that the solution might become excessively complex when it does not need to, that is why it is important to only use it when the relations between the child and parent are meaningful.

Logical and Physical Model

When designing the logical model, the author made sure to follow the guidelines for the UML notation correctly and the naming convention. The logical follows mostly the third normal form (3NF) except on two separate occasions. 3NF is used to avoid duplicate data, avoid data anomalies and simply management of data. But in the case of student and instructor where they can both be identified either using the id or their person number. The author's motivation behind creating an id number is to not display a person's person number which should remain private.

Because the author chooses to thoroughly design the database, it can run all the different operations (business transactions), no tables are missing, and no unnecessary tables exist in the database. Each table only consist of relevant columns and data. The author made sure that all column constraints and foreign key constraints are specified with either ON DELETE SET NULL, ON DELETE CASCADE or ON DELETE NO ACTION. All primary keys are well chosen so it can be easily understood and easy to use to select specific data.

When designing the logical model, the author noticed that the conceptual model consisted of couple of flaws. For example, the system for renting an instrument was reconstructed. This is because otherwise it would not follow the task four goal which is to keep track of past and current rentals. By using this system, the model has a table of all instruments and a table for renting instruments. If someone wants to rent an instrument it gets added to the renting instrument table which consist of all essential information about the rental. The author also decided to create a pricing scheme to easily check and update prices for different types of lessons.

The author's solution to the logical model has one flaw which is the lesson table. Because the author decided to merge all the different lesson types, it is needed to store a bit of null values. For example, the individual lesson does not consist of max or min students and therefore its values become null. A better system would be to continue the design from task one and keep the inheritance between the lesson and lesson types. That way each table will not consist of null/empty values.

SQL

The main reason to either use views (virtual table) or materialized views in SQL is because it is a great way to optimize a database. For example, if the task is to execute queries on three different tables, rather than having to join them together multiple times it might be a better idea to create a temporary view of them which than is used to execute queries on, Fundamentals of Database Systems Chapter 7.

The substantial difference between a view and a materialized view is that a view is a logical virtual copy of a table while a materialized view is stored on the disk, it is a physical copy of a table. Because of this a "normal" view will have slower processing speed while a materialized view will have faster processing speed. A downside to materialized view is that it requires memory space while a view does not.

When creating the different views for the queries the author chooses to use materialized view for the query that would be executed weekly to get the different ensembles the upcoming week. This is because that schedule will be used by multiple students often. Therefore, it might be better to save it as a materialized view due to the faster processing speed. But for the rest of the views the author chooses to use a normal view because those queries would not be executed often because they were mainly focused on the lesson data for an entire year. Because of that it would be unnecessary to save it as a materialized view which would require memory space.

Therefore, it is important to be mindful when choosing to implement either a view or a materialized view on different queries.

Programmatic Access

Because the author made sure in previously tasks that the database can perform all the required functionalities, implementing the command line interface and its program was simple. A user can list all instruments that is available and with a specific instrument type, it can rent an instrument and terminate an ongoing rental.

Because of the database structure, some queries will be needed to be rolled back if the other one fails. For example, if a user rents an instrument, it requires to execute two queries. One to mark the instrument as occupied, and one to create a rental. If the second one fails, the first one will be able to be rolled back and therefore the instrument will not be marked as occupied. This also applies when a user is terminating a rental.

When the author designed the program, it was important to implement ACID transaction (atomicity, consistency, isolation, and durability). As discussed above the program has a good atomicity because if a request to rent an instrument fails at a certain part, the other queries will be rolled back. The program has consistency of data because if an instrument is being rented there will never be a point where the instrument will be occupied and not being rented at the same time, Fundamentals of Database Systems Chapter 20. Before a student can rent an instrument it is first checked that the instrument is availed. This is because first off you should not be able to rent an instrument if its already rented and this also helps for the isolation because if two rentals happen at the same time. What will happened is that one of them will fail because the other one will already have marked the instrument as occupied.

Because the author turned off auto commit for the atomicity its important to always commit after the different commands have been executed. After the commit command have taken place, the database will have stored that data and therefore the program will have good durability incase if for some reason the program just disconnects. All classes and packages follow the naming convention. The author also made sure that all different layers in the MVC pattern manages the correct logic.

General Discussion

The purpose of the task was to implement and represent information handling and business transactions for the Soundgood music school. By following the different sub tasks the author was able to structurally design a database which can manage all the different business goals. Even when designing the database in the structurally way the author still needed to go back to the first/second task and redesign the Conceptual Model and Logical Model. This was because when implementing the different business tasks the logic did not work as indented. But because the task was solved in a structurally way it was easy to go back and change and update the model and the database.