Data storage Paradigms

C. Alexander Berg
KTH – Royal Institute of Technology
Stockholm, Sweden
Alexberg@KTH.se

Keywords — Data storage, Soundgood Music School, XML, SQL, KTH, database models.

I. Introduction

Data storage Paradigms is a course that introduces students to the area of databases/data storage, this includes but is not limited to; query languages, XML, relational models, and conceptual modeling

As a part of this course, the students must complete a project, *The Soundgood Music School*. In this project, the students are instructed to create a model, database, and program to be used by a fictional school for their services. Some services that are included are, student enrollment, payments, renting service, etc. This project is divided into four steps, C*onceptual model*, *Logical and Physical model*, S*QL*, and *Programmatic Access*, respectively. Each of these steps has its respected requirement.

The conceptual model is meant to be a representation of a system, focusing on the concept and readability rather than the logic and physical aspects of the project. The requirements of this part focus on notation, naming conventions, a reasonable amount of entities, attributes (and storage of attributes), and business rules, just to name a few.

The logical and physical model on the other hand focuses on the logic and physical aspects of a model. This will essentially be a model which can be used directly to create a database using a relational database and query programming language. The requirements for this part focus on notation, relevance, relations, structure, and business rules, and constraints, just to name a few.

The SQL is created using the model of the previous step. The goal of this part is to create a relational database and deepen the understanding of concepts, principles, and theories for students in this field. The requirements for this part are to demonstrate the code and elaborate on why certain decisions were taken over others.

The last part of this project is to create a programmatic Access to create a back-end solution for the administration of this school. This includes being able to list instruments, rent instruments, and terminate rentals, just to name a few. The requirement of this part concerns functionality, naming conventions, and transaction management.

This report is written by Alexander Berg, who worked with Jabez Kungu Otieno during the duration of this project. The result of the said project will be discussed in the following sections of this report.

II. Literature study

A. Introduction

To finish this project, there was a necessity for a research phase for each of the four tasks. The research methods used include but are not limited to: Fundamentals of Database Systems by Ramez Elmasri and Shamkant B. Nabathe, lectures given by Leif Lindbäck and Paris Carbone, and lectures given by other professors or courses. This will all be discussed further in this section of the report. As the project was split into four steps, this section will be divided similarly.

B. Conceptual Model

A conceptual model is a representation of a system, which focuses on concepts used to help people understand the subject which it represents. This ẃas quite a new topic for the author, having been absent from an earlier course discussing these topics. Since a conceptual model can be represented as a UML, it had to be tackled first. Using Free Code camp's courses on YouTube, the author was able to learn differences between different diagrams, attributes, methods, and naming conventions, just to name a few. After this, the focus was to understand the basics of conceptual modeling. Using both chapters 3 and 4, in the database systems literature and the lectures given by Leif Lindbäck, the author was able to understand how to abstract information relevant to the model, and what information should be attributes or relations. Using this information and following along during these lectures, the author felt prepared to continue and build his conceptual model.

C. Logical and Physical Model

Understanding the difference between a logical and Physical Model to that of a Conceptual Model can be tricky at first glance, however, the lectures given by Leif proved useful as he built his model from a conceptual model. This was vital in understanding logical models and their physical aspects as well as learning concepts such as inheritance, 3NF, and the importance of relevance. This model had to in effect carry all the logic of the SQL code, which meant understanding the logic of SQL. This was carried out by watching an introductory course to SQL in free code camp as well as reading the 5th chapter from the official literature for the course. The author did not go as in-depth as he could as he wanted to wait with this for the next part of the course.

D. SQL

SQL is a domain-specific language used in programming and database management systems. The logic for this language is unique, however, since the previous section required some basic query language understanding it was not much more than to pick up where the author left off. The lectures provided by Paris Cabone and the 4th part in the literature proved useful in understanding how to write and use SQL in a meaningful and effective way. The author had to also understand how to share values between tables, write queries, basic scripting, and database-specific functions, among others. All of which were explained and elaborated upon in the literature and lectures.

E. Programatic Access

Programmatic access is the final layer for the project. The code will create a backend solution for the administration of the sound-good music school. In order to achieve this, the author had to research first about transactions, which is concerned with the reading and writing of data in a database. The lecture given by Paris Cabone was shown to be very useful in understanding how to structure and program transactions in both a safe and effective manner. Part 6 in the official literature proved to be also useful in creating a more in-depth view and perspective for the author. After that, the author also had to understand how to communicate with a database using a language such as Java. For this, the lectures on database applications by Leif Lindbäck proved to be very useful in the basic understanding of these concepts. Even tho the last part of this project was not fully completed at the time of writing this report the basic understanding and research by the course proved useful.

III. Method

A. Introduction

This method section serves as an explanation of how the author achieved the results he did, as in a description of the steps to complete the part of the project. The result of which is shown below:

B. Conceptual Model

The conceptual model was created by firstly listing out every single item or function that was mentioned in the description of the sound good music school as entities. Next up was listing every single function that was not directly mentioned but would be used, for example, payments. Next up was creating relations between these entities using astha professional, software for UML, and other diagrams/models.

At this stage, we would be able to see that some entities would be redundant since they can be applied as attributes or relations, an example of which is a student having a student ID. Lastly repeat the last 2 steps, create entities then reduce the irrelevant entities or convert them to attributes or relations. Eventually, the model was unable to be reduced, at which point it could be consulted with other students and teachers to see if it is accurate and appropriate for this project.

C. Logical and Physical Model

The Logical and Physical Model was created by copying and modifying the conceptual model created at the previous step. Since this model is based on SQL and other query language logic, it would be smart to read up on that before the creation of this model. It is created using the same software, astha professional, and modeled based on PostgresSQL.

Each entity in the logical model will have to be recreated to tables. Late each attribute has to also be recreated, this time to a column with a cardinality of 0 or 1. We also have to change the naming conventions during these steps to snake case from camel case used in the conceptual model. Then we want to create a table for each column with a higher cardinality, such as contact details. Lastly add types for each column. What is meant by this is to specify if the column is an integer, char, date, time, etc. Repeat these steps if necessary, if you are unable to add to a model, consult with other students and teachers during seminars to check if the model is accurate and appropriate.

D. SQL

The SQL is created using the logical and physical model and uses the same DBMS, PostgresSQL. This will make it easier to code if you are basing your code on the

lectures in the course. The code was also written in visual studio code for its integration between java and SQL, as well as its tools which help PostgresSQL development. The bash/zsh terminal also proved useful in the creation of this database as it made it easy to test and execute code, as well as making it easy to integrate this with a CLI for the last part of this project.

The SQL code was created firstly by the Logical and Physical model, using Astha's "soft-wear". This was later tweaked to make the code more suitable for this project. This was then added with further work on queries that will be performed. Lastly, since PostgresSQL does not support all functionality, these have to be written as an addition to this project.

At this point, we should have some resemblance of code that will achieve something. If the code is without errors then nothing needs to be added. This was not the case for this group. At times columns had to be added, showing a misstep in the logical and physical model. This can be solved by creating new tables or rewriting tables. More details on these errors will be in the results section.

E. *Programatic Access*

The programmatic access was tackled in 4 different parts. The first part, connecting to the database was achieved using the official postgres driver and JDBC.

The second part, the user experience, created the overall structure of the entire database. How each function should be written and how the user should interact with the information.

The third was to collect and print data from our database. This was achieved using pre written statements which would be executed within the database. Some of these statements would ask the user for information such as type of instrument. This will then give back the result to the user.

The last was to update data within the database. This was solved in a similar method to that of the collection and printing of data. It still takes in any necessary inputs being studentID or instrumentID, with the main difference being the statements which updates the database instead of selecting specific data.

IV. Result

A. *Introduction*

The result section is the section that focuses on the results of the methods and a detailed description of how each requirement was met with illustrations and examples accompanying it.

This entire project was written in a GitHub repository, the link of which is shown below:

https://github.com/calexanderberg/IV1351

This link includes all the models and code which was created during this project.

B. *Conceptual Model*

The final conceptual model is shown below:

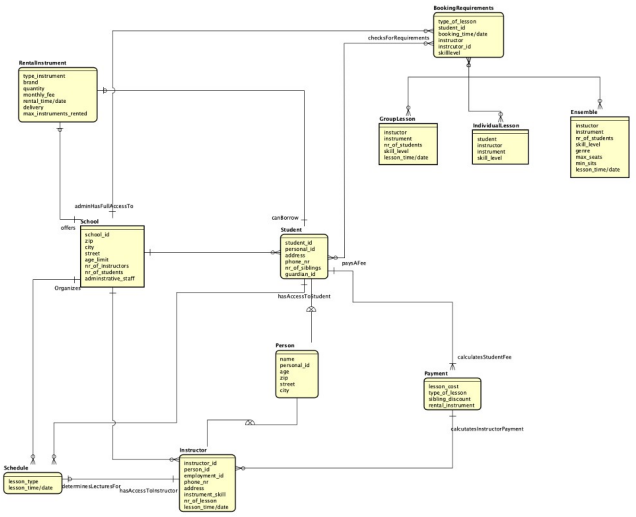

Fig. 1: conceptional model.

This model is centered around the student entity which is shown below:

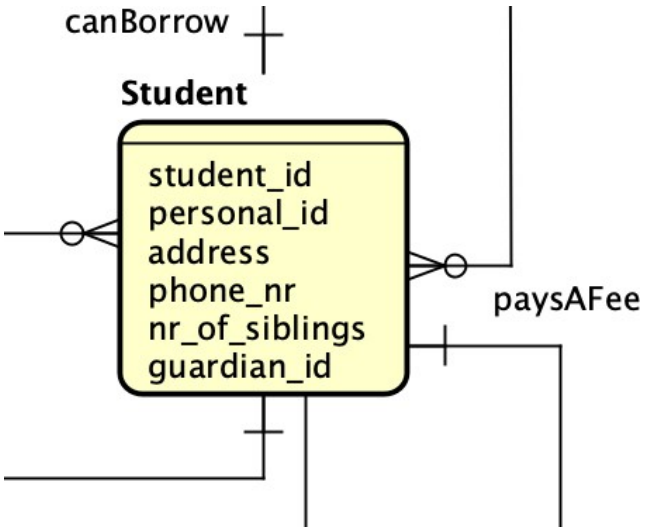

Fig. 2: conceptional model, student.

The student's attributes include student, guardian, and personal id as well as the number of siblings. The latter of which will be used to identify if the student has a sibling in the school, which will be used in a discount for their monthly payment, which will be discussed later.

The id attributes are related to the student through the application and school entity, used to make the student sign up for classes or to the school as a whole, shown below:

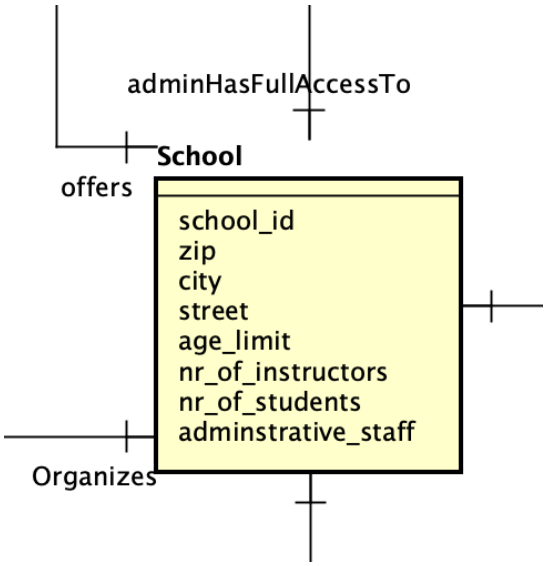

Fig. 3: conceptional model, school.

As shown above, the school has many attributes, such as an age limit for students, administrative staff, students, and relations to schedule, offering rental instruments, etc. These will all be explained in more detail during this report. The school will then organize classes for the person or student entity that has applied to the school, shown below:

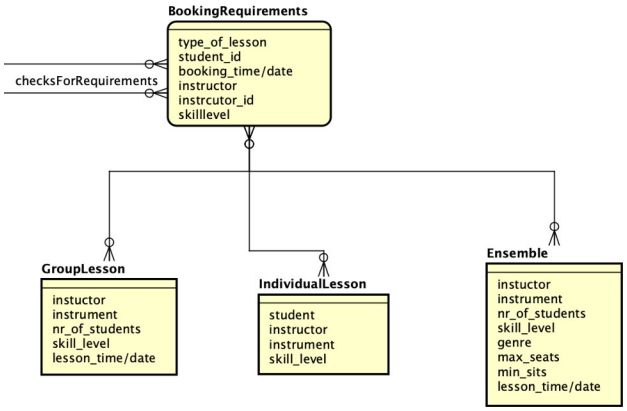

Fig. 4: conceptional model, lessons.

The lessons (or bookings) come in three different entities, individual, group, or ensembles. Each with an instructor, instrument, skill level, and student(s). The biggest difference between these entities would be that the ensembles and group lessons have the attribute of nr_of_students. This means that they have a dedicated lesson_time/date while individual lessons do not as it is a private lesson between a student and an instructor. Each lesson does have a skill level which is an attribute related to the student as seen in fig 2. The lessons are also given a time and date from the instructor attribute as seen below:

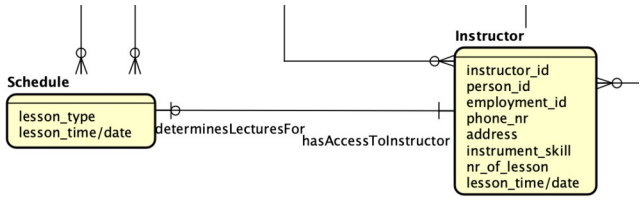

Fig. 5: conceptional model, schedule and instructor.

The scheduling entity organizes the available lectures for instructors to school shown in fig 1 and schedules lessons accordingly. The schedule has a lesson_type, this has the same value as the three entities that the lesson requirement is related to. The instructor entity also has access to the schedule where they can give their availability. Since an instructor is an entity similar to that of a person and student, it has to be given the following attributes: person_id, instrument, and skill level. On top of that, the instructor is also given an employment_id and instructor_id attribute which is used to distinguish them between other employees, instructors, and students. Lastly, it is also given a nr_of_lessons and lesson_time/date attribute, this will be used by schedule to plan lessons accordingly.

For students to attend lessons they need to either bring their own or rent instruments. The entities that take care of the ladder is shown below:

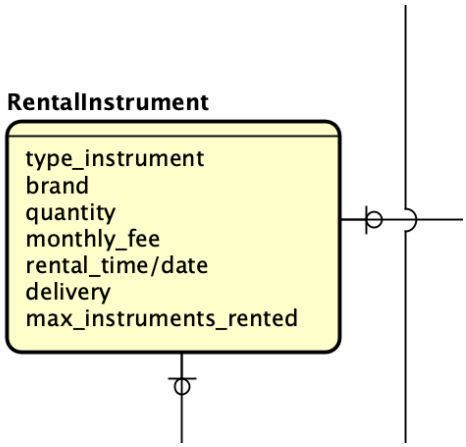

Fig. 6: conceptional model, rental.

The rental instrument entity is related to the student entity. The rental instruments have a couple of attributes including, the type of instrument, the brand of said instrument, the quantity of those instruments, the monthly fee of rental, and lastly the total time and dates of the rentals. Each instrument rental is also paid by the student and is added to its monthly payment which is shown below:

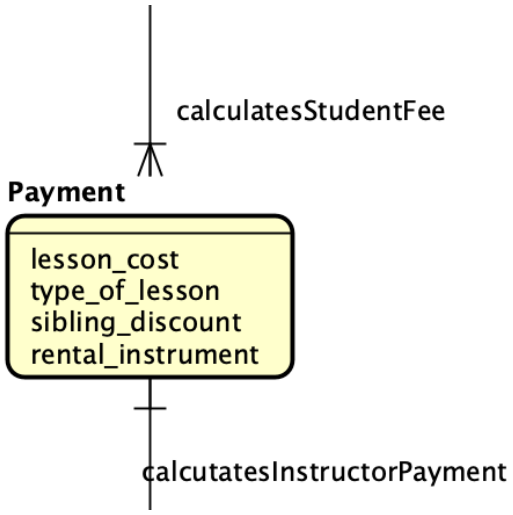

Fig. 7: conceptional model, payment.

The payment entity is related to two entities outside of this figure here; the student who pays a monthly fee, and the instructor, who is given a salary. However, there is also a discount added for students through the sibling discount attribute, this is calculated by the number of siblings per student times said discount.

Lastly is a collection of comments which explains the business rules and constants that were not clearly explained by the model.

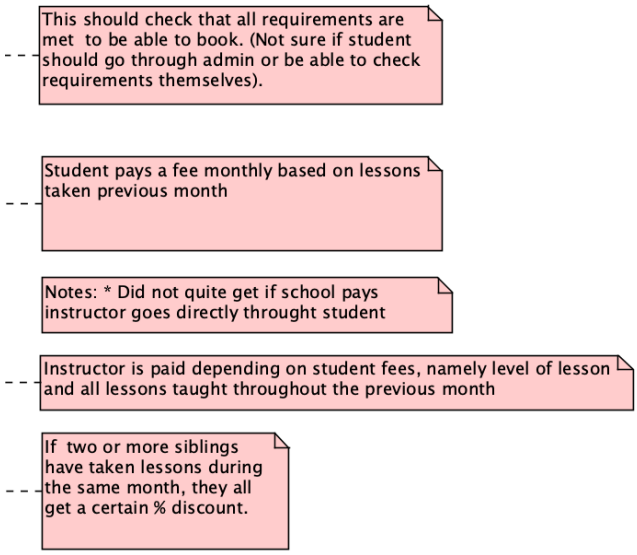

Fig. 8: conceptional model, comments.

This is the basis for the group's conceptual model and how each entity is related to another, this conceptual model was later used as a basis to create a logical and physical model, which is the next section of this report.

C. Logical and Physical Model

The final logical and physical model is shown below:

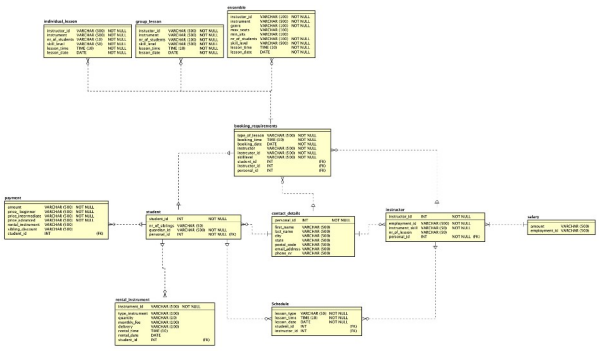

Fig. 9: logical and physical model.

Since this model is based on the conception model, it will be discussed similarly. The model is not, however (like the conception model) centered around students, but is instead is centered around student and instructor, this is shown below:

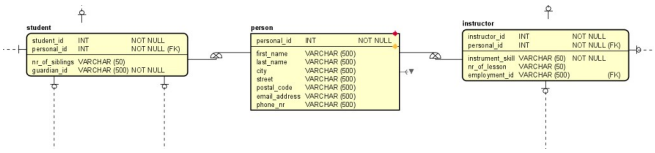

Fig. 10: logical and physical model, instructor and student.

Since the contact details are shared between the two tables it is separated into its table being related to both student and instructor. Each column (besides personal_id) is a char, including phone number (to account for the + for international numbers) and postcode (to account for the space). Note that the personal ID is NOT NULLthis means that without a personal ID, contact details can not be accessed. It is also marked as (FK), meaning foreign key, this means that the value is used to access information in another table. Some columns are student or instructor-specific. For example, both have an ID that identifies them in the system and both have a personal ID to relate it to the contact details. The biggest difference between them is that instructor has a lesson-specific column, which will be discussed later; and the student has the number of siblings and guardian ID, the latter of which can be used to find contact details for guardians by adding it in contact details.

All of these three tables also have a relation to booking requirements, this is explained more below:

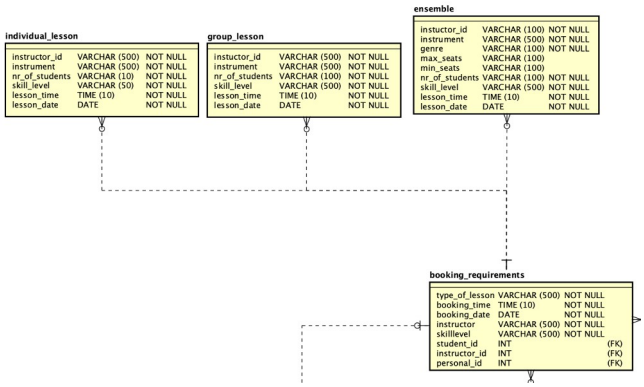

Fig. 11: logical and physical model, lessons.

The booking requirements table has relations to three tables (besides the ones mentioned earlier). Here the lesson time/date was split into two columns to make it easier to log the data. The other columns remained similar to that of the conceptional model, with nr of students, instructor, skill level, and instrument being columns for each lesson type. The only difference is the ensemble, who uses both max seats, min seats, and genre. Since ensembles can be played with multiple instruments, and need a minimum number of seats for it to occur. The booking requirement needs to have information about all students, instructors, and their contact details to have lessons. This is achieved by using their respective columns: student ID, instructor ID, and personal ID, note that these are also marked FK. The rest of the columns are marked NOT NULL as the information for the lessons needs both skill level, type of lesson, booking time, and data as a requirement for the lesson to even start. The amount of lessons that the instructor teaches is later used to calculate his salary:

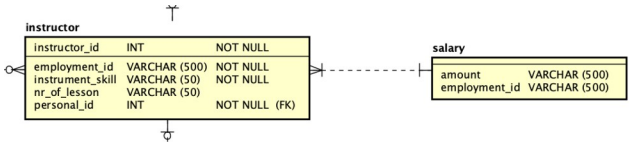

Fig. 12: logical and physical model, salary.

Here, the relation between instructor and salary is related through employment ID. This is because every employee within the school has a salary even if they are not in this model it should still be accounted for in case the school wants to add on to this model in the future. This salary is paid for by the student and payment table:

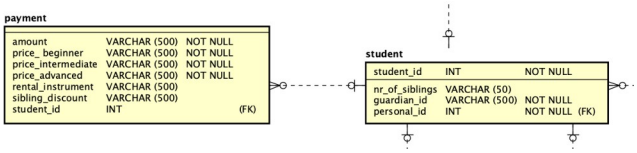

Fig. 13: logical and physical model, payment.

The payment takes a student ID as a foreign key to access the student information, finding what lessons he attended, what level they were at, and if a discount should be applied (ie the student is a sibling of another student). There are three prices for different difficulty levels of lessons. These can not be Null as it is used in the calculation of prices for all students. The same logic goes for amount, if a student is enrolled in a school, it could be assumed that he would at least attend one lesson. However, the student ID is allowed to be null as this payment should only be used when requested with a student ID. The rental instruments can also be null as a student might own his instrument. However, if the student were to rent an instrument the rental instrument table would be used:

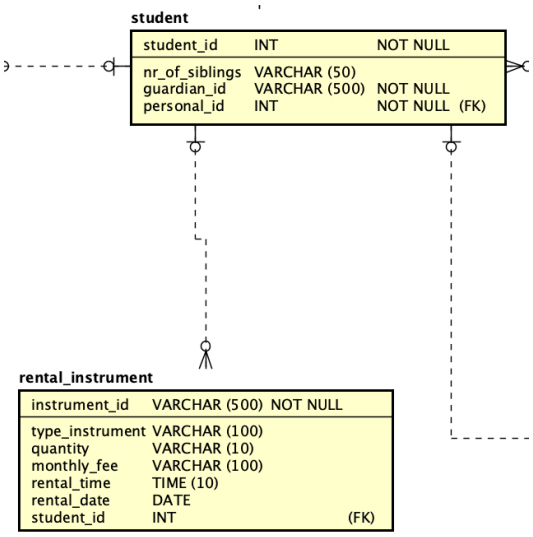

Fig. 14: logical and physical model, rental.

In this example, the rental instrument is accessed first using the unique instrument ID. The instrument ID is given to each instrument and can therefore not be NULL as it wouldn't make sense to have an instrument without an ID. There is also a quantity column, since students are only allowed to rent two instruments at a time, the rental time and date, and the fee is used to calculate the amount that should be applied to the payment. The student ID is used to identify the student which rented the instrument, which is why it is a foreign key. REMOVE DELIVERY. Lastly, the type of instrument is used as an identifier for the stock left for specific instruments.

There is also a unique schedule that was separated from classes and instead put as a table with relation to both student and instructor, which is the final table of this model, shown below:

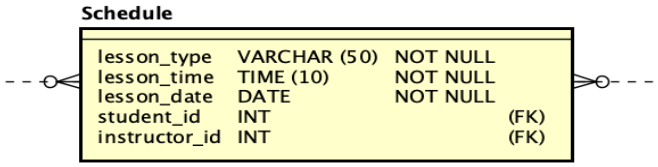

Fig. 15: logical and physical model, schedule.

The schedule is used to organize classes for both students and instructors. Both of their IDs are foreign keys because of this reason. The other three columns, lesson type, time, and date, can not be null as they are part of the definition used for lessons. As the difficulty is not relevant in the schedule it does not make sense to put it here.

As was in the conception model. Below is a collection of comments which helped explain some of the rules and constants in this model.

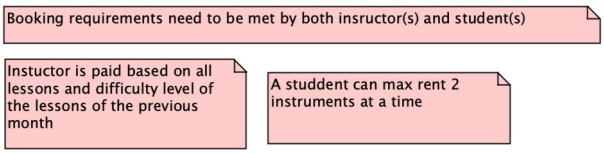

Fig. 16: logical and physical model, comments.

This Logical and Physical model was quite extensive, but it did help the creation of the SQL code which is part of the next section in this report.

D. SQL

The queries used in this program is split into four different tasks, all of which is shown below:

```
SELECT date_trunc('month', booking_date) AS time, count(type_of_lesson) AS count
FROM booking
GROUP BY date_trunc('month', booking_date) ORDER BY time;

SELECT date_trunc('month', lesson_date) AS time, count(ensemble) AS count FROM
ensemble
GROUP BY date_trunc('month', lesson_date) ORDER BY time;

SELECT date_trunc('month', lesson_date) AS time, count(group_lesson) AS count
FROM group_lesson
GROUP BY date_trunc('month', lesson_date) ORDER BY time;

SELECT date_trunc('month', lesson_date) AS time, count(individual_lesson) AS
count FROM individual_lesson
GROUP BY date_trunc('month', lesson_date) ORDER BY time;
```

Fig. 17: Query 1

```
SELECT 'AVG OF ENSEMBLES LESSEON';
SELECT DATE_TRUNC('month',booking_date) AS time,
ROUND((CAST (COUNT(skilllevel)AS DECIMAL)/12)::DECIMAL,2)
AS AVG FROM booking GROUP BY DATE_TRUNC('month', booking_date) ORDER BY time;
```

Fig. 18: Query 2

```
WITH time_report AS (
SELECT instructor_id, count
(instructor_id)


FROM booking WHERE date_trunc('month', booking_date)=date_trunc('month',
current_date)
group by instructor_id order by count DESC    )
SELECT * FROM time_report WHERE count > 2;
```

Fig. 19: Query 3

```
SELECT * FROM ( SELECT max_seats, nr_of_students,lesson_date ,
CASE
WHEN max_seats=nr_of_students THEN 'FULLY BOOKED'
WHEN 2 = max_seats-nr_of_students THEN 'FEW SEATS LEFT'
WHEN 1 = max_seats-nr_of_students THEN 'ONE SEAT LEFT'
ELSE 'JOIN'
END AS is_available FROM ensemble)
ensemble
WHERE EXTRACT('week' FROM lesson_date)=EXTRACT('week' FROM current_date +
interval '1 week');
```

Fig. 20: Query 4

The first and second queries lists the number of lessons per month. The differences between them is that the second one lists the average number of lessons while the first lists all the lessons by type, such as ensemble, individual, etc.

The third query lists all instructors who have given a specific number of lessons during the current month. The last query checks the availability of ensembles giving the user a look at how many seats there are left. These will be explained further in this report.

```
          time          | count
------------------------+-------
 2022-01-01 00:00:00+01 |    19
 2022-02-01 00:00:00+01 |    15
 2022-03-01 00:00:00+01 |    31
 2022-04-01 00:00:00+02 |    29
 2022-05-01 00:00:00+02 |    15
```

Fig. 21: Query all lessons for a specified year

```
          time          | count
------------------------+-------
 2022-01-01 00:00:00+01 |     3
 2022-02-01 00:00:00+01 |     3
 2022-04-01 00:00:00+02 |     3
 2022-06-01 00:00:00+02 |     3
```

Fig. 22: Query all ensembles from a specified year

On figure 21 and 22 we are using our first query to list all lessons (and ensembles for fig 22) for our specified year. This was done by selecting each type of lesson and grouping by month. At the end of the group by statement, the order by helps sort the table during the specified year.

```
          time          | avg
------------------------+------
 2022-01-01 00:00:00+01 | 1.58
 2022-02-01 00:00:00+01 | 1.25
 2022-03-01 00:00:00+01 | 2.58
 2022-04-01 00:00:00+02 | 2.42
 2022-05-01 00:00:00+02 | 1.25
 2022-06-01 00:00:00+02 | 2.08
```

Fig. 23: Query average number of lessons per month

The second query lists the amount of lessons per month for the entire year, by selecting each booking and using the round and cast clauses to round the number to 2 decimal places and to convert an expression of a data type.

```
 instructor_id | count
---------------+-------
 733-18-7056   |     6
 156-61-3856   |     5
 615-15-6851   |     3
 847-06-9365   |     3
```

Fig. 24: Query list of instructors by lessons

For the net query it will list instructors that currently have more than a specific number of lessons for a month, in this case 2. This is achieved by selecting the instructor by ID and counting the lessons associated with said ID.

max_seats	nr_of_students	is_available
10	5	JOIN
10	8	FEW SEATS LEFT
10	5	JOIN
10	9	ONE SEAT LEFT

Fig. 25: Query seats for ensembles for coming week

For the last query the user is able check the availability of seats. If there is 2 or 1 seat the program will output “Few seats left” or “one seat left” respectively. Otherwise “join” will indicate to the user that there are seats left.

E. Programatic Access

The programmatic access, as mentioned earlier is split into multiple sections, however for this results we will go over the code as the user would see it play out. In order to do this we need to first mention the initilization of the code which starts with the first lines of the main function shown below:

```
public static void main(String[] args) throws SgDBException, SQLException, ClassNotFoundException {
    Scanner scan = new Scanner(System.in);
    Connection c = new Main().accessDB();
    c.setAutoCommit(false);
    Statement Statementt = c.createStatement();
    preparedStatements(c);
```

Fig. 26: main function 1

This gives us a scanner to use for user input and calls the function accessDB, mentioned later in this report. It also initilizes the connection c, setting it to not auto commit (making our code only change when we order it to) and preparing our statements.

```
public Connection accessDB() throws SQLException, ClassNotFoundException {
    Class.forName("org.postgresql.Driver");
    return DriverManager.getConnection("jdbc:postgresql://localhost:5432/sg","test", "test")
    ;
}
```

Fig. 27: AccessDB function

In the accessDB function we set our postgres driver and connects to our localhost server named sg. This essentially is the heart of the entire program as without it we wouldnt be able to connect to our database.

After this initilization step the user would see his first output of the system shown below.

```
try {
    System.out.println("Welcome to the SoundGood music school's website.");
    System.out.println("We are able to do the following:");
    System.out.println(
            "1: List of available instruments by type.\n" +
            "2: Show rented instruments.\n" +
            "3: Add rental.\n" +
            "4: Terminate rental.");
    System.out.println("What would you like to do? Please pick a number.");
    int choice = scan.nextInt();
```

Fig. 28: main function 2

This will prompt the user to select what task he wants to preform, this will then initalize a case switch statement in the program. (We will discuss the try method later)

```
switch (choice){
    default:
        System.out.println("ERROR: Number does not exist, please choose a number between 1 to 5.");
        break;

    case 1:
        System.out.println("You picked 'List of available instruments by type.'");
        System.out.println("Choose instrument out of the following:\n" +
                "harp\n" +
                "saxophone\n" +
                "guitar\n" +
                "ukelele\n" +
                "bongo drums\n" +
                "violin\n" +
                "keyboard \n");
        printInstrumentType(input.next());
        break;

    case 2:
        System.out.println("You picked 'Show rented instruments.'");
        System.out.println("The list of instruments is shown below:");
        printAllRentals();
        break;

    case 3:
        System.out.println("You picked 'Add rental.'");
        System.out.println("Please state the studentID:");
        int studentID = input.nextInt();
        System.out.println("Please state the instrumentID:");
        int instrumentID = input.nextInt();
        addRental(studentID, instrumentID, c);
        break;

    case 4:
        System.out.println("You picked 'Terminate rental.'");
        System.out.println("Please state the instrumentID:");
        terminateRental(input.nextInt(),c);
        break;
}
```

Fig. 29: Switch case statement

As the default case is self explanatory, the author will start with focusing on the first case statement. This case prompts the user to list which instrument type he wants to list, this is then used i the *printInstrumentType* function.

```
private static void printInstrumentType(String instrument) throws SQLException {
    ResultSet rental = availableInstruments(instrument);
    while (rental.next()){
        System.out.println("Type of Instrument: " + rental.getString("type_instrument") +
                           " | Instrument ID: "+ rental.getString("instrument_id") +
                           " | Price: " + rental.getString("monthly_fee") + ":- per month");
    }
}
```

Fig. 30: printInstrumentType Function

The printInstrument function takes an input of a string of instrument and uses this to call the function avaliable instruments to connect to a prepared statement which will select all rental instrument which is not rented and where the type of instrument is equal to that of the input of the user. These two functions are shown below:

```
public static ResultSet availableInstruments(String inst) throws SQLException{
    getAvailabilityInstrumentStatement.setString(1,inst);
    return getAvailabilityInstrumentStatement.executeQuery();
}
```

Fig. 31: avaliableInstruments Function

```
getAvailabilityInstrumentStatement = c.prepareStatement("select * from
rental_instrument where is_rented = FALSE and type_instrument = ? ");
```

Fig. 32: getAvaliableInstrumentStatement Statement

If there is a instrument of the avaliable type the rental variable will be used in the while loop and print out the type of instrument type, id and price. Please note as all case statements are structured in a similar way, they will only be explained from here on out.

The second case of the statement is similar to that of the first one with a very small exception.

```
case 2:
    System.out.println("You picked 'Show rented instruments.'");
    System.out.println("The list of instruments is shown below:");
    printAllRentals();
    break;
```

Fig. 33: Case 2

That being that it does not take an input and instead calls the function *printAllRentals*.

```
private static void printAllRentals() throws SQLException {
    ResultSet rental = getAllStatement.executeQuery();
    while (rental.next()) {
        System.out.println("StudentID: " + rental.getString("student_id") +
                           " | InstrumentID: " + rental.getString("instrument_id"));
    }
}
```

Fig. 34: printAllRentals

The print all rental function calls the statement *geAllStatement* directly. This gives the user a full list of all instruments not yet rented, as can be seen in the output below:

```
You picked 'Show rented instruments.'
The list of instruments is shown below:
StudentID: 5551 | InstrumentID: 4
StudentID: 8745 | InstrumentID: 6
StudentID: 8745 | InstrumentID: 7
```

Fig. 35: Output printAllRentals

```
case 3:
    System.out.println("You picked 'Add rental.'");
    System.out.println("Please state the studentID:");
    int studentID = input.nextInt();
    System.out.println("Please state the instrumentID:");
    int instrumentID = input.nextInt();
    addRental(studentID, instrumentID, c);
    break;
```

Fig. 36: Case 3

For the third case, we give the user an option to add rental to a student, here they are prompt to give a studentID and instrumentID which will then be used in the function *addRental*.

```
private static void addRental(int studentID, int instrumentID,Connection c)
throws SQLException {
    if (numberRental((studentID)) == 2 ) {
        System.out.println("The maximum amount of rental has been reached
        for this student.");
    } else {
        try{
            addRentalStatement.setInt(1,studentID);
            addRentalStatement.setInt(2,instrumentID);
            addRentalStatement.executeUpdate();
            c.commit();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 37: printAddRental Function

The *addRental* function uses a simple if else statement to make sure that the student does not have more than the required number of rentals. This is used to ensure that the rules given to the group ensures. Making sure that the rules remain in the system. This is then using a try catch statement to make sure that the program is only commited after everything have run sucessfully. If something fails the table is intact.

In order to check if the student has over two rentals we call the *numberRental* function shown below.

```
private static int numberRental(int id) throws SQLException {
    ResultSet rental = numberRentals(id);
    rental.next();
    return rental.getInt(1);
}
```

Fig. 38: numberRental Function

Since the *numberRental* function outputs an int we need to seperate it with another function *numberRentals* to make sure it collects the necessary data from the database, in this case number of instruments rented and converts this to an int.

If the student has less than two instruments in the function will call *addRentalStatement* which updates the database to add the rental to the system.

For the last case the user is able to terminate a rental.

```
case 4:
    System.out.println("You picked 'Terminate rental.'");
    System.out.println("Please state the instrumentID:");
    terminateRental(input.nextInt(),c);
    break;
```

Fig. 39: Case 4

As every instrument is unique the user only needs to input the instrumentID and use the system to find the studentID, this is done in the *terminateRental* function.

```
private static void terminateRental(int instrumentID,Connection c) throws
SQLException {
    int studId = getStudentId(instrumentID);
    terminate(instrumentID,c);
    addHistoryStatement.setInt(1,instrumentID);
    addHistoryStatement.setInt(2,studId);
    addHistoryStatement.executeUpdate();
    c.commit();
}
```

Fig. 40: terminateRental Function

The terminal rental function calls getStudentId and gives it an input of our instrument ID. This will check the database of any student which might be using this for their classes. After that the instrument will be sent to *terminate* which will terminate the rental.

```
private static void terminateRental(int instrumentID,Connection c) throws
SQLException {
    int studentId = getStudentId(instrumentID);
    terminate(instrumentID,c);
    addHistoryStatement.setInt(1,instrumentID);
    addHistoryStatement.setInt(2,studentId);
    addHistoryStatement.executeUpdate();
    c.commit();
}
```

Fig. 40: terminate Function

This function calls the prepared statement updating the status of the instrument to not rented and setting the student ID assosiated with the instrument to null.

After this has been executed the *addHistoryStatement* will be called which notes the instrumentID and studentID then executing the update and commiting the changes for the database.

After the 4th case the author had also added a catch statement to act as a rollback during the program's execution. This is show below:

```
catch(SQLException e)  {
    e.printStackTrace();
    try {
        c.rollback();
    }
    catch (Exception ex2){
        ex2.printStackTrace();
    }
}
```

Fig. 41: Catch Function

This statement will print out any error which might occur when the program runs and write it to our stack trace. The program will then try to rollback using the c.rollback() function. If this fails it will print out ex2 to our stack trace.

All in all this system gives a user almost complete control over how rentals are handeled, from creating new rentals to terminate or even just listing them. There are definetly more functionality that could be added for future renditions, however that is a task for another day.

V. Discussion

A. Introduction

The discussion section is the last and final section in this report. The purpose of this section is to discuss if the requirements for each part were met, what problems were faced, and lastly what could have been done differently.

B. Conceptual Model

The conceptual model had a plethora of requirements which will be split up and discussed further below:

1) Are naming conventions followed? Are all names sufficiently explaining?

The group followed the standard naming convention which happen to be a camel case. All names are sufficiently explained, the entities and relations all have names explaining what their purpose is, such as payment, content, student, etc.

2) Is the notation (UML or crow foot) correctly followed?

The group followed the crow foot notation, this was correctly followed by the design of the relationships and cardinality. As well as the design of the entities.

3) Is there a reasonable number of entities? Is some important entity missing?

The group does not believe any important entities were missing and kept the number of entities to the lowest without losing the simplicity or the usability of the model. Some entities had to be removed for the reason that they did not serve a purpose in a database, an example of which is the administration who books classes for the instructors. Since this would be a functionality added to the programmatic access it did not require to be part of the model.

4) Are there attributes for all data that shall be stored? Is cardinality specified for all attributes?

All the data that needs to be stored has attributes specified for said data as well as cardinality for each attribute.

5) Are all relevant relations specified? Do all relations have cardinality at both ends and name at least at one end?

All relevant relations are specified, meaning every relation has a name that describes what its purpose is. They also have cardinality on each end.

6) Are all business rules and constraints that are not visible in the diagram explained in plain text?

The group believes it did, in the last part of the result section is a collection of comments explaining the rules and constants which would not be obvious unless stated otherwise, such as booking requirements and siblings having to be enrolled in the school.

7) Did you use inheritance? If you did, what are the advantages of using inheritance in this model?

The model did use inheritance for student and instructor, separating information such as name, eprson_id, age, zip, etc. The problem with inheritance, when used with conceptional models, is that the simplicity or readability could be sacrificed as it might confuse the user who is trying to understand the model since this is a low-level view assuming the user is familiar with inheritance might be a stretch. The advantage of using inheritance is that the model might become easier to translate to logical and physical models as it is a fundamental function of query-based languages, as it will allow attributes to not be repeated over multiple entities.

C. Logical and Physical Model

The Logical and physical model had also other requirements which were split up similarly to the conception model, the result of which is shown below:

1) Are naming conventions followed? Are all names sufficiently explaining?

The group followed the standard naming convention which happen to be snake case. All names are sufficiently explained, the entities and columns have names explaining what their intended purpose is, such as nr_of_lessons or lesson_type. There was no simplification or assumptions made about the user of the model.

2) Is the crow foot notation correctly followed?

In the model crows foot notation is correctly followed. Each relation has a cardinality of the appropriate design and each table is designed appropriately.

3) Is the model in 3NF? If not, is there a good reason why not?

Since all columns are determined by one key in each table and no other column it is 3NF. This helps sim-

plify the model without having to add extra complexity to our model.

4) Are all tables relevant? Is some table missing?

All tables are relevant, the group does not believe that any tables were missing, Some tables were removed due to the logic of the database not making sense, for example having separated rental services and the rental instrument.

5) Are there columns for all data that shall be stored?

Yes, there are no data that is stored which does not use a column. Also, no column has a NOT NULL which will not use some data for the software to operate.

6) Are all relevant column constraints and foreign key constraints specified? Can all column types be motivated?

All column constants relate to information that defines a table, such as classes having a time, date, and students. Some information is allowed to be blank such as if a student has siblings. Sufficed to say, no information which is not needed is not included in this model.

7) Can the choice of primary keys be motivated?

We use the following as public keys, student_id, personal_id, and instructor_id. These values identify individuals which will have unique classes, rentals, and skill levels to that of other individuals. Therefore, these keys need to exist to correctly assign data to that person, such as the information mentioned earlier.

8) Are all relevant relations correctly specified?

All relations are correctly specified, as each column has a foreign key when related to an object, this means that the data which is transferred using the relation is specified.

9) Is it possible to perform all tasks listed in section one, Project Description, above?

Almost, as mentioned before some functions are better left to the programming access part, such as administrators booking classes. It is still possible to book classes using this model in the database however a CLI or anything of the like is not added to the model.

10) Are all business rules and constraints that are not visible in the diagram explained in plain text?

At the end of the result, there is a collection of comments which explained rules such as there being a limit of 2 instruments per student, or that the instructor is paid based on all lessons given.

11) Did you use inheritance? If you did, how does it differ when used in a conceptional model? What are the advantages and disadvantages?

The model did use inheritance in person. The student inherits contact information from the person and the same goes for the instructor. The advantage of using inheritance here instead of the conceptual model is that the conception model's main purpose is to make the user understand the purpose of this project. With that, it is much more fitting to add it in the logical and physical model as it is concerned with the logic and physical aspects o the database. It therefore should be modeled in logical and physical models. The advantage of inheritance is that you do not need to repeat any values if multiple tables use it, such as phone number or address which is used by both student and instructor.

D. SQL

1) Are views and materialized views used in all queries that benefit from using them? Can any query be made easier to understand by storing part of it in a view? Can performance be improve using materialized view?

Both views and materialized views were used in the all queries that were benefiting them. This simplified the queries whenever it was necessary to fetch data from the database. The reason for this is if there is complicated queries or logic in the database, it would be possible ton create views that would hold said logic. Otherwise it would be possible to query that view whenever the data was needed, however only the query is saved in the view and not the actual data.

On a materialized view however it returns data stored in the view not querying from underlying tables making it necessary to refresh occasionally.

The group did agree on the performance benefit of these solutions, making it faster than if the query would be used in transformations of a considerable amount of data.

2) Did you change the database design to simplify these queries? If so, was the database design worsened in any way just to make easier to write these queries?

The group had no previous experience with DBMS languages. The group found it easier to manipulate the database design than to conform with the design, sufficient to say, this did not make the database worse as the group did not ignore the purpose of the data or its underlying purpose.

3) Is there any correlated subquery, that is a subquery using values from the outer query? Remember that correlated subqueries are slow since they are evaluated once for each row processed in the outer query.

There is correlated subqueries in the code. Even with its slow speed in mind it was the best solution the group could envision during the project.

4) Are there any unnecessarily long and complicated queries? Are you using any UNION clause where it is not required?

No, no queries are unnecessarily long and/or complicated, and there was no usage found for UNION clauses.

5) Have you made sure to always used ENUMs or tables instead of free text for constants such as the skill level (be-

ginner, intermediate and advanced)? Why did choose the option that you choose (ENUM or table)? What are the advantages and disadvantages of each option?

The group had no previous experience with DBMS languages, making the ENUM data type not being utilized in the program. The group assumed the ENUM data type would be more time consume and expensive as they believed it would cause problems down the road. In retrospect it would have simplified the database greatly. But it was not implemented as the group found it difficullt to reuse the list in other tables.

E. Programatic Access

1) Are naming conventions followed? Are all names sufficiently explaining?

The program follows the camel case naming convention, the names are all sufficiently explained. There could be room for improvement, in terms of minimizing the length of each name.

2) Is transaction management correct? Are there ACID transactions, which are committed or rolled back correctly?

The ACID transaction, as explained earlier, updates and writes as soon as the program commits. This has been implemented in the program only allowing the table to be updated after the run has been successful.

3) Is the program working as expected and does it meet all functional requirements listed in this task?

All functions work as intended, the list functions list the instruments that are available to rent and can be organized by type. The terminate and rent functionalities, updates the database and adds history to the program helping the user organize his renting ability.

VI. Conclusion

Since the author was unfamiliar with many of the technologies and processes which was utilized during the project, it proved to be a bigger challenge than expected. The project can and should be seen as a fantastic learning opportunity for those who are not familiar with modeling, query languages, or fundamentals in data storage.