Project Report

Data Storage Paradigms, IV1351

Oscar Persson - opers@kth.se

2021-03-15

1. Introduction

The task at hand was to create an accessible database system application for the fictional Soundgood Music School. The overarching purpose of the assignment was to end up with an application handling all of the school's data and all of its transactions.

The project was divided into five tasks, each task adding another layer of intractability to the database, first of all being to create a conceptual model for the database which must cover the entire application description given. The conceptual model had to be created using either UML or one of all crow-foot notations. Secondly we were to create a logical and physical model for the database, which also had to cover the entire application description given. This one had to be written using a crow-foot notation and was then made into a database from the model. Thirdly we were to write OLAP queries on the database. The fourth task required us to write a java program with a Command Line Interface (CLI) which can access the database and perform transactions to the database. Task 5 pertains to performance increases by using indexes to speed up the queries made to access the database within task 3.

To summarize the end result of this assignment was to have a completely accessible database model of the fictional Soundgood Music School, through multiple access points.

2. Literature Study

Task 1

In order to start on task one, which was to create a conceptual model of the Soundgood Music School database one firstly needed to learn about the requirements for the conceptual model itself, but also research the different options one could take, for example using UML rather than a crow's foot notation. As most lectures and the book presented the options as either UML or IE notation (as our crow-foot notation) it was decided to limit the choice between these two. After looking at the two choices I decided to go with IE notation, not because there seemed to be any discernible difference between the functionalities of each two (they are just ways to describe the same thing), but because it was found to be more intuitive.

The lecture regarding conceptual models[1], as well as the book *Fundamentals of Database Systems chapter 3*[2] were used in order to research how the conceptual model worked. Through looking at the examples in the video lecture while concurrently reading in the book an overarching view of the requirements was achieved, andt the model was built based on that using the steps explained in the video lecture.

[1] Lecture - Canvas, Conceptual Model, Leif Lindbäck
[2] Fundamentals of Database Systems, Ramez Elmasri

By looking at the object oriented development PDF linked in the lecture page for the conceptual model[3] lecture the five steps were researched further in-depth before the creation of the model.

Task 2

The research that was made in order to finish step 2 of the project consisted of the lecture material for this particular section[4]. Specifically the material from the lecture about logical and physical models[5]. The decision to focus on this material was made because of the nature of the assignment, creating a "phys-logical" model, as opposed to both, which most other material explained. The video and accompanying slides were also sufficient to gain knowledge about the 11 steps for creating this "phys-logical" model. The video was used in order to gain knowledge about how to apply the 11 steps for creating the logical model, as well as the examples from the video. The culmination of this was a sufficient basis for how to follow the steps.

Information from the book[6] regarding the differences between a conceptual model and a logical model was also investigated, especially the differences in terms between the two types of models.

Task 3

Most of the research regarding SQL was made using the website Code Academy[7]. The website provides a decent basis for how to use SQL commands, and when to use them. By allowing examples in a temporary database hosted on their website a baseline understanding was achieved which was then built upon by their so called "SQL Cheat Sheet" which contained all possible standard SQL commands, along with a short explanation to their purpose, and an example for usage areas.

The lecture videos were loosely used for research about the SQL language, but as Code Academy was used before watching the video it provided only a small amount of information within usage areas for SQL.

The result of this was a baseline understanding of SQL which allowed for expansion within the usage areas related to the assignment.

Task 4

Knowledge of the Java language has been acquired from beforehand throughout all the courses at KTH.

The lecture about Transactions was firstly used in cooperation with the book *Fundamentals of Database Systems*[8] in order to get a basic understanding of transactions and their usage within SQL databases, as well as how they are adapted when used within Java code.

[3] http://leiflindback.se/iv1350/object-oriented-development.pdf
[4] Lecture - Canvas, SQL - The Structured Query Language, Leif Lindbäck
[5] Lecture - Canvas, Logical and Physical Models, Leif Lindbäck
[6] Fundamentals of Database Systems, Ramez Elmasri
[7] https://www.codecademy.com/
[8] Fundamentals of Database Systems, Ramez Elmasri

The lecture video about Database Applications[9] was used in order to get a basic understanding about the components and the connections between the database and the program. The lecture provided knowledge about the Java Database Connector (JDBC), how to establish a connection to the database, and how to provide queries and save the responses. The lecture then provided information about how SQL queries should be written and embedded in order to be run in an efficient way, inside of the java code.

Task 5

In order to get a decent understanding of how indexes work within SQL a multitude of online sources were utilised. A baseline understanding of what the usage area for this specific tool was used for was achieved through the accompanying course book *Fundamentals of Database Systems*[10]. Online sources were utilised as well above this in order to get a more practical understanding about how to apply indexes, specifically on the database created through this project. In order to achieve this sources such as tutorialspoint[11] and EssentialSQL were used[12]. Through this an understanding of what indexes are and when to use them were acquired.

3. Method

Task 1

The software decided to use was LucidChart. The reason for this choice was because the interface was found to be comprehensible as well as the access between multiple platforms such as iPadOS and browser through the cloud. As it could export and import SQL it would seem to be a decent choice, with the multiple platform usage to be the main advantage over a program like astah.

Following the outlined steps for creation of the conceptual model the creation of the model started with noun identification to find the class candidates. Scanning through the assignment description all nouns were picked out of the description, adding them as classes (or rather candidates) to the model.

The second step, using a category list to find class candidates was made in conjunction with step one, even though both not being mandatory, in order to guarantee that one would find every possible candidate. Reading the more in-depth description of the database, using the category list given in the video further candidates as classes were found, through the requirements of the actions and contents of the database.

One step that was decided to be put aside was the requirement of no duplicated during the category list. By being able to add duplicates one could spend more time completely finding all nouns, rather than being cautious of having added the noun/class candidate beforehand. This was made as a time saving effort, but also in order to not miss anything, and is amended in the next step.

[9] Lecture - Canvas, Database Applications, Leif Lindbäck
[10] *Fundamentals of Database Systems, Ramez Elmasri*
[11] https://www.tutorialspoint.com/sql/sql-indexes.htm
[12] https://www.essentialsql.com/what-is-a-database-index/

The third step was to remove redundant class categories. This was done by placing all classes found into lucidchart, sorting them by usage according to the description. If a class candidate was not used it was to be discarded, and vice versa. Then by scattering them one could detect if two or more candidates filled the same purpose, discarding one of them while keeping the one which fit the description the best.

The fourth step was to decide which classes would fit better as attributes. This step was decided to be done the other way around; finding candidates which definitely should be classes, rather than attributes. Doing this allowed the possibility to find candidates which would fit as attributes to these classes, removing a bit part of the candidates immediately, leaving fewer to actually consider according to the step. The rest of this step just consisted of finding classes which rather should be an attribute, where descriptive candidates were found, added as attributes.

The fifth step was to add associations between the classes remaining. While doing this it was decided to pick out the most central associations first, drawing associations to the classes they would interact with according to the assignment description. Following logic of how it could be assumed things would interact in the real world the finalized model was created with these associations.

The sixth step, considering any changes to the model, was made progressively all the time during the creation of the model whenever something was found to be lacking or strange, but also done during the end of the process.

Following the steps resulted in a conceptual model for the Soundgood Music school according to the specifications of the assignment.

Task 2

The eleven steps that were used for creating the logical model for the Soundgood music school are as follows:

1) The first step consisted of creating a table for each entity.

2) The second step was adding the columns for the attributes. During this step the creation was limited to attributes with a cardinality of 0..1 or 1..1, and was followed as specified.

3) The third step is regarding attributes of a higher cardinality. This is because the attributes may only contain one value. By breaking them out into individual tables one can then assign multiples to each original table for the attributes.

4) The fourth step was regarding the types for the attributes, where a type was to be specified for each column of the table.

5) The 5th step regards the constraints of the attributes and their types, and was done in conjunction with the 4th one, as the type and the constraints are intertwined in their usage.

6) During the 6th step primary keys (PKs) were to be assigned to each table.

7) The seventh step was split into parts, but was treated as one continuous step during its execution. During the seventh steps relations between the tables/entities are to be created. Creating relations also requires creating the foreign keys, which are then created in tandem, and picked using the primary key in the strong end as the foreign key in the weak end. The step includes specifying constraints for the foreign keys, but this step was already done during the 5th step, as all constraints were specified during.

The rest of the steps were followed as outlined using the conceptual model as a basis for its creation. Using the description all actions specified in the assignment were extracted and specified. Having these as a concrete concept allowed for testing of the model with all possible actions. If the model passes the test it may be considered complete. Any action that cannot be performed requires modification of the model in order to accommodate the action, allowing it to be performed. This was tested multiple times in order to make sure that a change for one action does not break another action.

Task 3

The Database Management System (DBMS) decided for usage within this database was MariaDB, which is similar to MySQL, seeing as it is a forge thereof. This was installed via XAMPP, which also installed the administration tool PHPMyAdmin. Through the usage of these programs I implemented the database that was modelled in step 2.

Relation constraints were then implemented through the web interface for each foreign key.

The database was then populated with relevant example data for each possible query that was specified in step 3. In order to simplify this process SQL queries that populated each table accordingly. If tables required other tables to be populated as well, a common SQL query were developed that would populate and “connect” the relevant tables. Each query was then analysed in order to find out which data needs to be used, and would be populated with random example data accordingly.

The tests were developed with the required SQL queries in mind. The data chosen was data that would appear in the SQL query, as well as data that would not appear when the SQL query was run, assuming the SQL query was correct.The assumption was then made that if the correct data appeared when the query was run, and the incorrect data didn’t appear; the query is correct.

MariaDB does not natively support materialised views, something that was not taken into account initially when creating the database. A workaround for this can be achieved by creating a table using the specific query made from the assignment specification. The result of this is a table containing all the relevant data, as a snapshot from the database at the time of creation. In order to keep the data up to date triggers can be added which will regularly update the data in the table.

Task 4

Task 4 was split into multiple steps, each with their own substeps to be completed. The major steps were defined as follows:

- Writing the SQL queries
- Writing a java program that establishes a connection to the database
- Writing a Command Line Interface program
- Writing an input parser

The input parser was taken from Leif Lindbäck's CmdLine class made for the bank application. The reason for this was because it wasn't relevant to the assignment at large, and already existed. All other code was written from scratch with some inspiration from Leif Lindbäck's bank application[13].

The first step was writing the SQL queries. The queries were written outside of the java program and tested directly to the database, to see if they returned correct results. The reason for this was because the program will run them in the same way, but making sure they run properly beforehand eliminates one possible point of failure later. Test data was provided for all the queries in two forms: data that should appear, and data that shouldn't appear. The queries were then written and run, and if the correct data appeared they were considered correct.

The Command Line Interface was then written as a separate component. It was written to be able to run completely on its own, where the class containing the appropriate functions for connecting to the database were only written into the necessary places in order to keep troubleshooting local to each class. The requirements for the command line interface were as follows:

- It should contain all commands specified in the assignment
- The program shall run and accept commands until termination by a quit command
- The program shall accept input arguments and parse them correctly into the command that was run

The class connecting to the database will then also be written, establishing a persistent connection for the current session, from which all commands are run. The commands will be implemented as functions, where the SQL code is put, with arguments passed into the functions as variables into the SQL command strings. The resulting functions will then be called from the CLI.

The IDE chosen to develop this program was IntelliJ. The reason behind this choice was because IntelliJ's seamless SQL handling, but also the ease of adding the JDBC for MariaDB to the program.

This is a modular approach to the database application which allows for easy troubleshooting and a vast possibility for extensions.

[13] https://github.com/KTH-IV1351/jdbc-bank

4. Result

Task 1

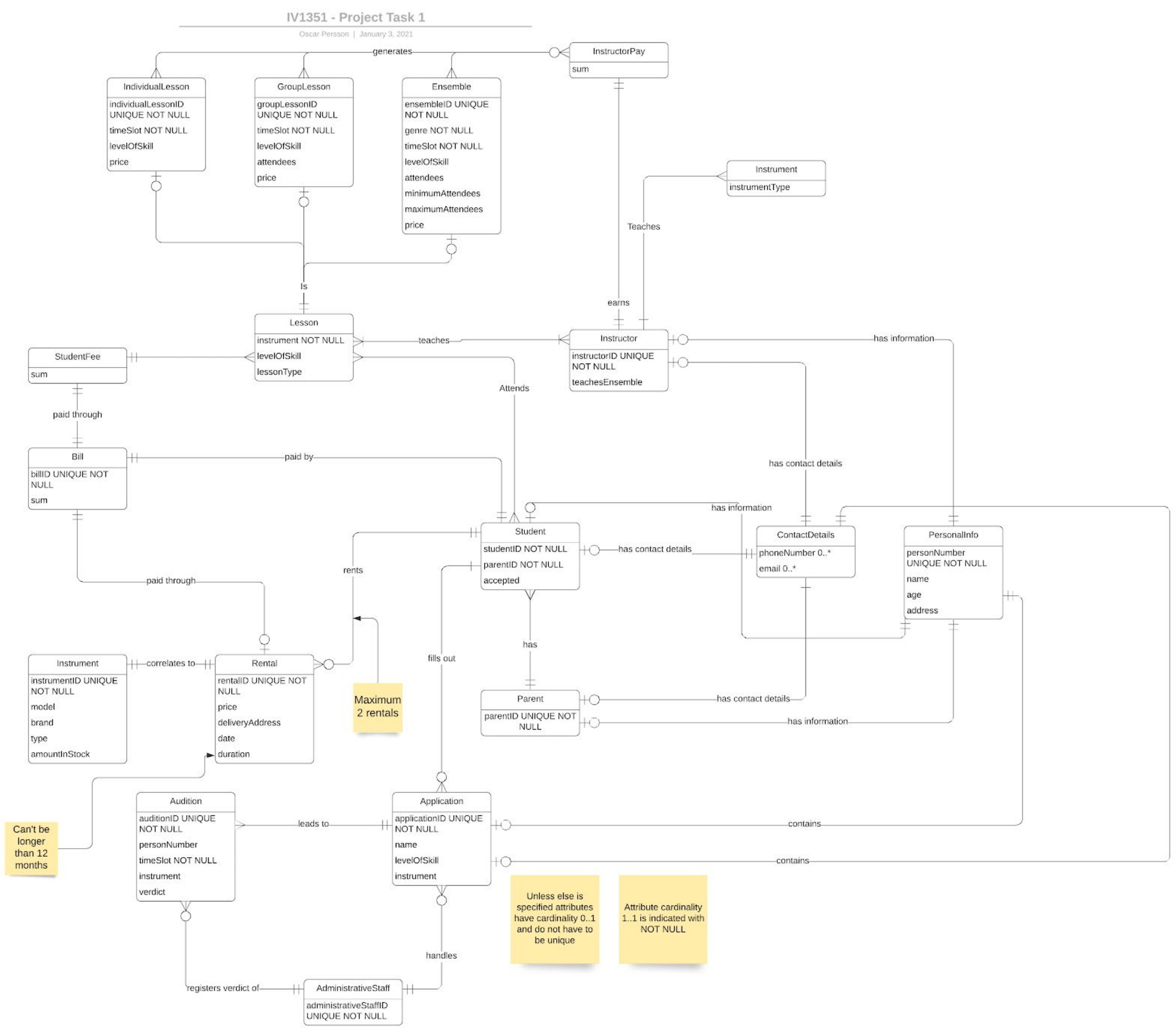

Figure 1: Conceptual model from task 1 describing the Soundgood Music School

Figure 1 describes the resulting conceptual model based on the Soundgood Music School business description. The diagram itself was made using crow-foot notation and was made using the five steps outlined in the method section, regarding the subsection for task one. The idea of the model lies behind the separation between people, lessons, and the school itself. Each person, according to the requirement description needs to have specified contact details and some personal information attached to each other. As this is shared between all people these were made into separate classes. Specifying which kind of person, instructor, student or parent, was then made through a separate class connected to these aspects for the person itself. This concludes the person section.

The lesson part has an overarching class for all lessons with a common set of attributes, it is then related to a more specific type of lesson containing the more specific attributes for those types of lessons. The lesson may only be one of these lesson subtypes.

There are also classes for more administrative aspects of the description, for example renting an instrument, where the rental is connected to an instrument that is rentable, both making up a rental that may be assigned to a student. As specified in the model a student may only have two rentals at maximum, and a rental may only last for 12 months.

Task 2

The logical model created differs slightly in comparison to the conceptual model. The reason for this was the complete recreation necessary when switching software. A model was initially created in LucidChart, but was scrapped as it wouldn't export the SQL correctly for import into the MariaDB database.

The resulting model was the result of the transformation from the conceptual model, having taken the eleven steps for creating a logical model into consideration. The model was split into subsections where each pertains to a certain functionality, for example:

- Rentals
- Lectures
- Administration
- Personal information
- Etc.

The constraints were made by what felt logical dependency wise, regarding the foreign keys. Foreign keys were specified by Astah, automatically, but every relation drawn was taking this into account in order to keep it optimal.

The names were changed in order to adhere to the standards of a logical model, as opposed to a conceptual model, but the names themselves were mostly kept the same as the same method for deciding them were the same as for the conceptual model.

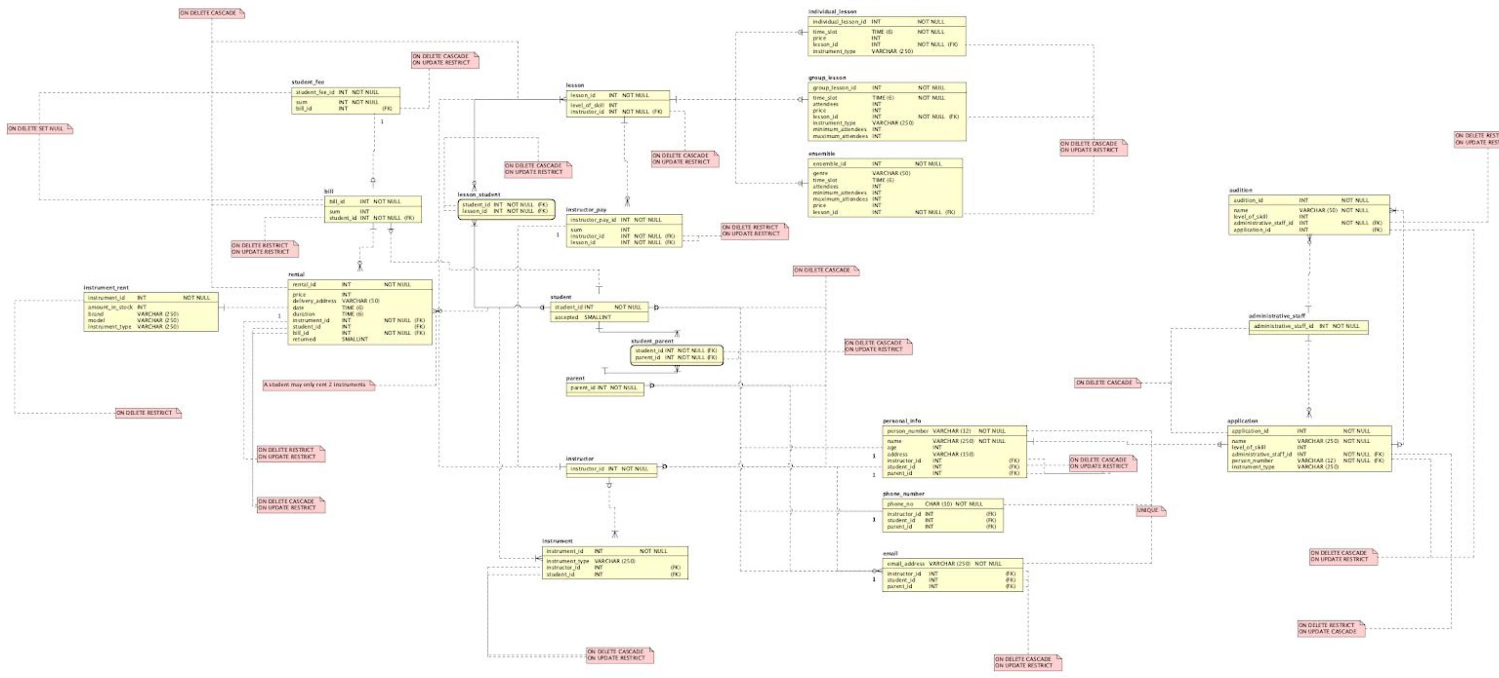

Figure 2: The logical model

Link to sql script
Higher resolution image/Astah file

Task 3

The queries created are as follows:

Query	Explanation

SELECT
    'TOTAL' AS 'Month',
    'ALL' AS 'Instrument',
    COUNT(rental_id) AS 'Amount'
FROM rental
WHERE YEAR(date) = 2021
UNION
(
    SELECT
        'TOTAL' AS 'Month',
        instrument_rent.instrument_type AS 'Instrument',
        COUNT(instrument_rent.instrument_type) AS 'Amount'
    FROM rental JOIN instrument_rent
        ON instrument_rent.instrument_id = rental.instrument_id
    WHERE (YEAR(rental.date) = 2021)
    GROUP BY(instrument_rent.instrument_type)
)
UNION
(
    SELECT Month, Instrument, Amount
	Query 1:
	The query starts by selecting the total amount of instruments rented during the specified year. It then selects the total amount of each instrument rented during the specified year. Lastly it selects the total amount of instruments rented each month during the specified year.
	
	All these three types of queries are then unionised and displayed as a total.

```
                FROM (
                SELECT
                        MONTH(date) as 'Month',
                        'ALL' AS 'Instrument',
                        COUNT(instrument_rent.instrument_id) AS 'Amount'
                FROM rental JOIN instrument_rent
                        ON instrument_rent.instrument_id =
rental.instrument_id
                WHERE ( YEAR(date) = 2021 )
                GROUP BY( MONTH(rental.date) )
                ORDER BY Amount DESC) AS result
)
```

```
SELECT
        'TOTAL' AS 'Month',
        'ALL' AS 'Instrument',
        COUNT(rental_id) AS 'Amount'
FROM rental
WHERE YEAR(date) = 2021
UNION
(
        SELECT
                'TOTAL' AS 'Month',
                instrument_rent.instrument_type AS 'Instrument',
                COUNT(instrument_rent.instrument_type) AS 'Amount'
        FROM rental JOIN instrument_rent
                ON instrument_rent.instrument_id = rental.instrument_id
        WHERE ( YEAR(rental.date) = 2021 )
        GROUP BY( instrument_rent.instrument_type )
)
UNION
(
        SELECT 'Average' AS Month, 'ALL' AS Instrument, (SUM(Amount)/12) AS
Average
                FROM (
                SELECT
                        MONTH(date) as 'Month',
                        'ALL' AS 'Instrument',
                        COUNT(instrument_rent.instrument_id) AS 'Amount'
                FROM rental JOIN instrument_rent
                        ON instrument_rent.instrument_id =
rental.instrument_id
                WHERE ( YEAR(date) = 2021 )
                GROUP BY( MONTH(rental.date) )
                ORDER BY Amount DESC) AS result
)
```

Query 2:
The first part of this query works the same as in the first query, as they are based on the same basis.

The second part of the query sums all instruments rented and divides it by the amount of months in a year in order to receive an average count of instruments rented per month during that year.

```
(
SELECT
        MONTH(time_slot) AS 'Month',
        'Individual Lesson' AS 'Type',
        COUNT(*) AS 'Amount'
FROM individual_lesson
WHERE YEAR(time_slot) = 2021
GROUP BY Month
)
UNION ALL
(
SELECT
        MONTH(time_slot) AS 'Month',
        'Group Lesson' AS 'Type',
        COUNT(lesson_id) AS 'Amount'
FROM group_lesson
WHERE YEAR(time_slot) = 2021
GROUP BY Month
)
UNION ALL
(
SELECT
```

Query 3:
The query is split into four parts in accordance with the assignment.

The first part regards individual lessons, where the amount of individual lessons each month are selected and presented per month, during a specified year.

The second part does the same, but for group lessons, where each amount each month is selected and presented per month.

The third part works in the same way

```
		MONTH(time_slot) AS 'Month',
		'Ensemble' AS 'Type',
		COUNT(lesson_id) AS 'Amount'
FROM ensemble
WHERE YEAR(time_slot) = 2021
GROUP BY Month
)
UNION ALL
(
SELECT
		Month AS 'Month',
		'ALL' AS 'Type',
		SUM(Amount) AS 'Amount'
FROM
(
(
		SELECT
				MONTH(time_slot) AS 'Month',
				'Individual Lesson' AS 'Type',
				COUNT(lesson_id) AS 'Amount'
		FROM individual_lesson
		WHERE YEAR(time_slot) = 2021
		GROUP BY Month
)
UNION ALL
(
		SELECT
				MONTH(time_slot) AS 'Month',
				'Group Lesson' AS 'Type',
				COUNT(lesson_id) AS 'Amount'
		FROM group_lesson
		WHERE YEAR(time_slot) = 2021
		GROUP BY Month
)
UNION ALL
(
		SELECT
				MONTH(time_slot) AS'Month',
				'Group Lesson' AS 'Type',
				COUNT(lesson_id) AS 'Amount'
		FROM ensemble
		WHERE YEAR(time_slot) = 2021
		GROUP BY Month
)
)gary
GROUP BY Month
)
```
	but selects ensembles.
	
	The fourth part regards all/the total amount of lessons per month. This is achieved by unioning all the three previous queries, summing them and sorting them per month.
	
	The complete query is achieved by a union of all the four previous queries.

```
SELECT
		Month AS 'Month',
		'ALL' AS 'Type',
		(SUM(Amount)/12) AS 'Amount'
FROM
(
(
		SELECT
				MONTH(time_slot) AS 'Month',
				'Individual Lesson' AS 'Type',
				COUNT(lesson_id) AS 'Amount'
		FROM individual_lesson
		WHERE YEAR(time_slot) = 2021
		GROUP BY Month
)
UNION ALL
(
		SELECT
				MONTH(time_slot) AS 'Month',
				'Group Lesson' AS 'Type',
				COUNT(lesson_id) AS 'Amount'
		FROM group_lesson
```
	Query 4
	The query performs the same action as the one above. The number is then divided in order to achieve an average before presented as the query result.

```
		WHERE YEAR(time_slot) = 2021
		GROUP BY Month
)
UNION ALL
(
		SELECT
				MONTH(time_slot) AS'Month',
				'Group Lesson' AS 'Type',
				COUNT(lesson_id) AS 'Amount'
		FROM ensemble
		WHERE YEAR(time_slot) = 2021
		GROUP BY Month
)
)gary
GROUP BY Month
```

```
(
SELECT
		personal_info.name AS 'Name',
		'Current Month' AS 'When',
		Amount AS 'Amount'
FROM
(
		SELECT
				lesson.instructor_id AS 'Instructor',
				Amount AS 'Amount'
		FROM
		(
				SELECT
						LessonID AS 'LessonID',
						SUM(Amount) AS 'Amount'
				FROM
				(
				(
						SELECT
								lesson_id AS 'LessonID',
								MONTH(time_slot) AS 'Month',
								COUNT(lesson_id) AS 'Amount'
						FROM individual_lesson
						WHERE YEAR(time_slot) = 2021 AND
MONTH(time_slot) = MONTH(NOW())
				)
				UNION ALL
				(
						SELECT
								lesson_id AS 'LessonID',
								MONTH(time_slot) AS 'Month',
								COUNT(lesson_id) AS 'Amount'
						FROM group_lesson
						WHERE YEAR(time_slot) = 2021 AND
MONTH(time_slot) = MONTH(NOW())
				)
				UNION ALL
				(
						SELECT
								lesson_id AS 'LessonID',
								MONTH(time_slot) AS'Month',
								COUNT(lesson_id) AS 'Amount'
						FROM ensemble
						WHERE YEAR(time_slot) = 2021 AND
MONTH(time_slot) = MONTH(NOW())
				)
				)gary
				GROUP BY LessonID
		)gary2
		JOIN lesson
				ON lesson.lesson_id = LessonID
)gary3
```
	Query 5:
	The query selects data from two parts.
	
	The first part selected all instructors that have given more than a certain amount of lectures during the current month. The limit is specified as a threshold for the user to input on their own “[INSERT THRESHOLD]”. The query selects three pieces of information and presents them:
	● Name
	● The current month
	● The amount of lectures taught
	This data is selected from a union of all three lessons types, and extracted per instructor. The threshold then cuts off the instructors who have held less lessons.
	
	The second part works in a similar way, but sorts the amounts and cuts them off after the specified limit of three.
	
	The full query combines the two aforementioned queries and displays the joined result.

```
JOIN personal_info
    ON personal_info.instructor_id = Instructor
WHERE Amount > [INSERT_THRESHOLD]
)
UNION ALL
(
SELECT
    personal_info.name AS 'Name',
    'Last Month' AS 'When',
    Amount AS 'Amount'
FROM
(
    SELECT
        lesson.instructor_id AS 'Instructor',
        Amount AS 'Amount'
    FROM
    (
        SELECT
            LessonID AS 'LessonID',
            SUM(Amount) AS 'Amount'
        FROM
        (
        (
            SELECT
                lesson_id AS 'LessonID',
                MONTH(time_slot) AS 'Month',
                COUNT(lesson_id) AS 'Amount'
            FROM individual_lesson
            WHERE
  (
   YEAR(time_slot) = YEAR(NOW() - INTERVAL 1 MONTH)
                AND
                MONTH(time_slot) = MONTH(NOW() -
INTERVAL 1 MONTH)
  )
        )
        UNION ALL
        (
            SELECT
                lesson_id AS 'LessonID',
                MONTH(time_slot) AS 'Month',
                COUNT(lesson_id) AS 'Amount'
            FROM group_lesson
            WHERE
  (
   YEAR(time_slot) = YEAR(NOW() - INTERVAL 1 MONTH)
                AND
                MONTH(time_slot) = MONTH(NOW() -
INTERVAL 1 MONTH)
  )
        )
        UNION ALL
        (
            SELECT
                lesson_id AS 'LessonID',
                MONTH(time_slot) AS'Month',
                COUNT(lesson_id) AS 'Amount'
            FROM ensemble
            WHERE
  (
   YEAR(time_slot) = YEAR(NOW() - INTERVAL 1 MONTH)
                AND
                MONTH(time_slot) = MONTH(NOW() -
INTERVAL 1 MONTH)
  )
        )
        )gary
        GROUP BY LessonID
    )gary2
    JOIN lesson
        ON lesson.lesson_id = LessonID
```

```
)gary3
JOIN personal_info
        ON personal_info.instructor_id = Instructor
ORDER BY Amount DESC
LIMIT 3
)
```

```
SELECT
        DAYNAME(ensemble.time_slot) AS 'Day',
        ensemble.genre AS 'Genre',
        IF
        (
                ensemble.maximum_attendees - COUNT(lesson_student.student_id)
<= 0, 'FULL',
        IF
        (
                ensemble.maximum_attendees - COUNT(lesson_student.student_id)
> 2, 'Three or more',
                ensemble.maximum_attendees - COUNT(lesson_student.student_id)
        )) AS 'Remaining'
FROM ensemble
JOIN lesson_student
        ON lesson_student.lesson_id = ensemble.lesson_id
WHERE WEEK(ensemble.time_slot) = WEEK(NOW() + INTERVAL 1 WEEK)
ORDER BY ensemble.genre, Day
```
	Query 6:
	The query selects the genre and day scheduled for ensemble lessons during the current week + 1 (next week). The result is then parsed according to specification through the assignment and displayed accordingly by the return.
	
	The amount of spots are cross checked by the amount of students who are assigned to an ensemble lesson.

```
SELECT
        table1.Instrument AS 'Instrument',
        table1.Price AS 'Price',
        table1.Availability AS 'Availability',
        IF
        (
                table2.NextLesson IS NULL, 'None', table2.NextLesson
        ) AS 'Next Lesson'
FROM
(
        SELECT
                instrument_rent.instrument_type AS 'Instrument',
                rental.price AS 'Price',
                IF
                (
                        rental.student_id IS NULL, 'Available', 'Unavailable'
                )
                AS 'Availability'
        FROM rental
                JOIN instrument_rent
                ON rental.instrument_id = instrument_rent.instrument_id
        GROUP BY instrument_rent.instrument_type
        ORDER BY rental.price ASC
        LIMIT 3
) table1
JOIN
(
        SELECT
                time_slot AS 'NextLesson',
                instrument_rent.instrument_type AS 'Instrument'
        FROM
                group_lesson
        JOIN instrument_rent
                ON group_lesson.instrument_type =
instrument_rent.instrument_type
        WHERE time_slot > NOW()
        GROUP BY instrument_rent.instrument_type
)table2
ON table1.Instrument = table2.Instrument
GROUP BY Instrument
```
	Query 7:
	The query is divided in two parts.
	
	The first part is regarding the instruments, taking their availability and price into consideration. The query lists the three instruments of the lowest price, it then checks if a student is bound to the rental, in which case it is unavailable, and vice versa.
	
	The second part finds the next group lesson per the specific instrument and displays it.
	
	The full query is a combination of the two parts created through a join.

Table 1: Contains all queries created as well as a brief explanation for each. The year used in the examples is 2021.

The queries were then tested during writing by inputting example data in the same way as specified in the method. If the correct data was displayed, and the incorrect data was not, as according to specification through the assignment, the queries were assumed to be correctly working.

Task 4

Like specified in the method, the program was split into four modules, each module fulfilling a separate function.

The query which lists all instruments takes an argument string for which the instrument is specified through. It then checks the rental status along with the instrument catalogue available and finds any instrument that fits the criteria as well as matching with the argument instrument. The query then collects all data from the two tables containing the matching information, as picking specific data at this point would be unnecessary code. The returned data is then saved as a ResultSet and the returned instruments are looped through and printed out according to specification.

The second query, binding a rental to a student first checks all non-returned rentals bound to the student in question, returning them as a single number through a COUNT command. The number is then parsed and checked against the specification in the assignment; relating to not allowing a student more than two rentals at a time. If this limit will be exceeded by the current rental, the program terminates the command and returns a message saying that the student is already renting at full capacity. If the person specified has not rented two instruments the rental table pertaining to the current instrument that is available for rent will be assigned to the student id specified, and the relevant data will be updated alongside it.

The third query begins by saving the information regarding the instrument into variables within the java code. This is because otherwise it would be impossible to retrieve the certain information after marking the rental as returned. The rental that is free, bound to the instrument as well as the student is then marked as returned, and the return date is set to the current date (As that is the duration of the rental, no matter if it was returned on, before, or after time). A new rental table is then created and bound to the instrument, as well as populated with the information that does not pertain to when a student is renting the instrument in question, which may then be bound to a new student who may wish to rent the instrument.

Regarding transactions, set auto commit was set to off in order to allow for transaction handling as this project is using MySQL/MariaDB. As the queries are of a rather simple nature, that is they only do one particular action, they are considered to either be fully completed or essentially having failed. Because of this the queries were only committed when the entire function had been run. If an error were to occur they are rolled back.

The command line interface runs completely independently. The persistence of it is created through an infinite loop controlled by a boolean variable. The QUIT command then flips this boolean variable, which then terminates the program. Input is parsed through the CmdLine provided by Leif, of which the command line interface then extrapolates the relevant data. Each command in the command line interface corresponds to a function within the ServerAccess class, which accesses the server through SQL commands; thus each command calls a function and provides the correct arguments.

add example run

Task 5

The first index suggested will be regarding query number 7 of the assignment. As the prices are extracted in either ascending or descending order, in the case of the query: ascending, limited by three. Creating a query based on the prices listed column for the rental table would optimise the query by eliminating the need for sorting it every time it is run. This adjustment would optimise the query for every time it is run, especially as the database is scaled. Creating an index in the form of a B-tree would allow for a quicker lookup of the data that is relevant to the query, trading memory used in order to receive a performance increase. The reason a B-tree was chosen to be quicker was because it is ordered by nature, thus the relevant data may quickly be acquired. A hashed index can not guarantee this property and was thus not chosen for this.

Another suggestion could be an index sorted by year for rentals. This could possibly be used in more queries than one, but currently it will be suggested with query number 1 in mind. Most of the query is relying on the year that can be specified. Sorting the rental table by the date, specifically on the year of the rental would speed this entire query up substantially. This query could also make use of a B-tree, as the sorted nature of one would easily allow for the cutoff of the specified year.

The last suggestion for an index pertains to query 5. The suggestion here is to sort all the lessons by instructor id through the lesson table. Doing this sort would allow for a quicker lookup when counting the amount of lessons each instructor has taught. In theory this should accelerate the process of counting the lessons held by one instructor and speed up the query as the size of the database increases. This is a benefit as well, as this is a foreign key and is used for a join with the personal information table.

5. Discussion

Task 1 *

The naming conventions followed were as specified, upper camel-casing for the class names, and lower camel-casing for the attributes, which is as specified for conceptual models. As the names were slightly changed from the result of the five steps to be more comprehensible for someone reading just the conceptual model one could conclude the names to be sufficiently explaining in general, especially when in conjunction with the relations, although this is not necessary by design.

Entity amount was considered to be sufficient,, considering all entities are there to maintain and uphold the requirements of the database specification. As this was the result of the five step method outlined in the lecture another amount may have been achieved through different methods, but by following the method these are the essential entities that were left. As the project progressed one could further see classes that could be omitted, and as SQL hadn't fully been taken into account while creating this model, there are some classes made to contain that could easily be extrapolated using an SQL command instead of explicitly storing them as a class.

All relevant relations are specified with cardinality and name, though in only one end. While the placement of the name could be considered ambiguous the direction it is supposed to be read can easily be inferred through the context as names have been chosen very carefully.

While there are some attributes which could have been omitted in hindsight, as they either could be inferred through other means in later steps or they were more relevant to later steps, it was decided to leave them be as this model was created before the knowledge about the steps, and they don't contribute to worsening the model. It's just data that could be inferred through other means.

Task 2

The model created represents the database as interpreted through the assignment. The model was created keeping the requirements in mind while making it, in order to reduce the workload needed to appropriate it to the correct standards after its creation.

Regarding the naming rules, each table was made to adhere to them. The standard for the names followed was lower casing for each separate word, using an underscore for separation or spaces. The sufficiency of the names was based on the noun collection from the first step's first step. In order to not have any names be ambiguous or to be non-descriptive an attempt was made to keep them as separated as possible, that is to keep names from possibly being mistaken for other names. As the names that are necessary for the function of the model are included, and they are kept as separate as possible one could consider them to be distinct enough. By keeping the names as short as possible as well it's forced through implication that they are unique and explaining for their purpose.

As this model was based on the conceptual model created in the previous step it also suffers from a similar issue to the conceptual one. Namely the problem of relevant tables. During testing no missing tables were found that would affect functionality of the database, but there have been additions made that may be considered redundant. The reason for the existence of these tables is partially because they existed in the conceptual model, so they made it into this model as well, but also because at the time the knowledge of how to extract information implicitly was not achieved. Thus the assumption that everything needed to be retrieved explicitly led to the creation of unnecessary tables. An example of this would be the "bill" table relating to each student, which instead would be retrieved by other measures using SQL code. No table has been found to be missing however.

If I were to change anything in the model, except the redundant tables, I would change the relations and tables regarding the rentals, which added a lot of further work during step 4. The reason for this is the fact that a rental is bound to an instrument, and the rental is then bound to the student. This leads to a lot of unnecessary tables in the database as it gets populated. A fix for this would be to reverse the relation, storing the instruments, creating rentals only when necessary. As this was discovered too late and the SQL relations in step 3 had been written, it was decided to keep it as is and work around it, which ended up working, but introduced more problems than necessary.

Some attributes chosen were also found to be redundant during later steps. This is because the data easily could be extrapolated through other means, for example an SQL query. The attributes were simply not used throughout the assignment, as this was discovered during later steps. They do take up space that could otherwise be saved, but they have no effect on the system as a whole.

A switch had to be made regarding software used for the model. Initially the logical model was created using lucidchart, as well, but it had to be remade using Astah. The reason for this is because the output SQL code in order to create the database that was generated through lucidchart contained errors, or was in the wrong format for MariaDB to parse. This led to information being lost, and many modifications having to be made anew. Thus the switch to astah permanently was made.

There was also an issue regarding the types of variables, as Astah didn't contain support for either booleans. This was circumvented by keeping it as an int in the logical model, while changing it to a boolean in the database itself.

Task 3

Even though materialised views do not exist in MariaDB/MySQL, and an alternative method has been used instead, they will be referred to as materialized views, anyways throughout the report.

While there are other alternative methods for creating faux-materialised views for MariaDB natively, there are also plugins which add the appropriate functionality. An example is the plugin FlexViews which adds the functionality of materialised views for MariaDB, which are then incrementally updated, automatically.

The first query was saved as a view. Even though the query is relatively complex, and requires access to a conjunction of tables the query is run rather irregularly and not very often. The data doesn't change too frequently either, and would thus take up unnecessary disk space if saved as a materialised view.

Query number two was saved as a view. The motivation behind this choice stems from the frequency of which it will be run, and the nature of the query. As only the current year will be the year which has changing values, as well as the storage space necessary in order to store every materialised view from previous years will only take up unnecessary disk space, a view was motivated for this particular query. The frequency of running the query is also only performed a few times per week, which is why a view was chosen for this query.

The third query has no logical performance basis for why it was decided to be run as a view, but is motivated by necessity. As the individual lessons are booked like appointments, the bookings may be considered to be rather volatile. Thus it is important that the correct amount is updated every time. A materialised view will only return the latest refresh of the query, and does not necessarily guarantee that a new booking will be displayed immediately, no matter when during the week the query is run. Thus a view was chosen for this query.

Query number four can be stored as a materialised view. This is because while it will still be updated in the same vein as query 3, the averages do not display the same urgency for an up to date retrieval. As it's not run all too frequently and there is no such necessity for an up to date retrieval, a materialised view was chosen for this query.

The fifth query, regarding instructors who have given a certain amount of lectures during the current month, was decided to be run as a materialised view. The reason for this is the relatively large number of joins, which contribute to a less performant query, if run completely every time. This in conjunction with the frequency of which it is run, makes a materialised view seem like the best possible choice, as it would need to keep the data up to date every day that it is run. Creating a view for this would probably still yield a similar result, considering the small size of the database, but relative to scale this seems like the best possible option for this query in this database.

Query 6 and query 7 were chosen to be implemented as materialised views. The queries would benefit from the optimisation received from querying them once and saving them to a table, only updating them occasionally. The urgency to receive up to date information isn't as relevant for these queries, according to assignment specification. Updated information to the queries will also only pertain to certain data, which would easily be handled by a materialised view implementation.

The materialized view does not exist in mariaDB. The workaround for this is to create a table and enable triggers for data updating within either the graphical user interface PHPMyAdmin, or through the CREATE TABLE command, encapsulating the query. Creating normal view proceeds in a similar way, by encapsulating the query within a CREATE VIEW statement, and giving it a name.

Task 4

During writing the program java naming conventions were followed according to specification. Names were chosen to be as descriptive as possible, but also as concise as possible. This means that they are reduced to the base components that describe its functions. As the variable names only apply to people who are able to read the code there are also comments that would further explain functionality or other shortcomings, if any were to appear.

As specified in the result the commands are of a simple nature made by the program, and thus the queries can either be assumed to have executed completely correctly or not. The transaction handling reflects this as it only commits when all steps are fully completed. This could be considered to be correct, referring to the previously mentioned simple nature of the commands, where they are made to only fulfill one function, and thus if that one function fails, the entire query can be assumed to have failed and will thus be rolled back. This boolean functionality of the queries were deemed to be sufficient for this purpose during testing.

During all tests the program works as intended and specified through the assignment. There are however some functionalities above this that are not included, but would help improve the overall quality of the program. This being custom messages for when input data is not as intended. For example, if the user inputs a student ID that does not exist, the program will just roll back as it fails. Preferably the program should tell the user that something is amiss, and specify what, rather than just rollback, but that required further queries and error handling that was outside of the scope of this assignment. Above this the program runs exactly to specification, and usually there are graphical user interfaces connected to programs like this which would prevent these kinds of errors from ever happening, and thus they have been omitted as a possibility. A person using a command line interface is assumed to be able to handle the program appropriately.

The setup during the logical model regarding rentals bound to an instrument proved to be problematic during this step. This is because a rental is always bound to an instrument with its own properties, where they together make an entry in the rental catalogue. As this step was not read when creating the logical model there are better ways to set this up that would have been preferable to use. This specifically regarding the step where a rental and its data may not be removed when the instrument is returned, but the result works the same.

A problem did arise during this step where XAMPP on mac refused to allow connections through a written program, a needed port would not be open. Thus this part had to be written over multiple computers. In hindsight this should have been tested beforehand, but the functionality between the two platforms was assumed to be the same.

Task 5

Firstly, knowing about these indices when creating the queries would possibly have made the creation of the queries with respect to the possibility for indexes to be easier. Identifying places where indices could be used was found to be difficult as they weren’t known of during their creation.

For some of the query index suggestions the foreign keys were taken into consideration, but for others the priority lied within columns found to be very relevant to the query that was to be modified. An example for this is the choice of the price column in the first suggestion which was chosen even though it’s not a foreign key. The reason for this choice is because the entire query in a way revolves around this one column, and thus would massively speed up the results. This pertains more to the need to prioritise columns that appear often in where clauses.

For the small dataset currently existing in the database no substantial change will be made by creating these indexes for the queries, but as the database grows, which in theory could be an infinite growth, the indexes will keep the database access faster than without. The indexes suggested have taken this in mind.

The second suggestion would possibly be able to be used for multiple queries where a similar WHERE statement is executed, and this is known. The query chosen for the example, however was one of many in order to give a more specific example.