Project Report - Task 2

Data Storage Paradigms, IV1351

Reza Hosseini rezahos@kth.se

November 2022

The updated parts of the task2 for re-submission

1. the model is redone due to changes that were done on the CM model. The changes are mentioned in the task1 re-submission report but a few of them are as: payment entities are removed due to them being derived data. pricing of lessons for both students and instructor is moved to the corresponding lesson entity. renting price for instruments is moved to the instrument-stock entity. The updated model is shown in the Result section of this report.

2. all Foreign Keys have now the type INT in the model as well as in the database.

3. lesson and rental prices are now fixed and moved to their proper place.

4. all tables that are not cross-reference tables have now only one id as their PK.

5. pros and cons that are discussed in the Discussion section have now evaluations based on the project and the Logical model.

6. a complete model using inheritance is now added to the Discussion section.

1 Introduction

In this task we further develop the Conceptual Model that we had and by following some procedures we build a new model that becomes the Logical and Physical model of the project. The goal with the task is to have a minimal and normalized database in the end that contains all the necessary data that the Soundgood Music School requires and is ready to be used and insert data into. To do some evaluation on the database, we insert some imaginary data to test the functionality of the database.

2 Literature Study

To understand "Normalisation", we first watched the Youtube video by Paris Carbone:

- IV1351-2020-Normalisation

Then we did take a brief look at the main textbook we have `Fundamentals of Database Systems` by Elmasri and Navathe, specifically these chapters that explained the idea of normalisation:

- Chapter 14 - Basics of Functional Dependencies and Normalisation for Relational Databases
- Chapter 15 - Relational Database Design Algorithms and Further Dependencies

We followed the instructions on Canvas page about `Logical and Physical Models` which included three Youtube videos from Leif Lindbäck:

- Logical and physical models, part 1
- Logical and physical models, part 2
- Logical and physical models, part 3

We made also a continuous use of `tips-and-tricks-task2.pdf` document though out the whole task.

3 Method

Programs and tools that were used during this task:

- Astah: Create the Logical and Physical model.
- VS Code: Create `schema generator` and `data generator` files.
- pgAdmin 4: Create, modify, test and evaluate tables and also insert data and evaluate it.
- generatedata.com: Generate data for the database.
- Latex: Create the task report.

To create the Logical and Physical model we followed a 11-step guide that made sure the conversion (from the Conceptual model) is correctly done:

1. Create a table for each entity:
   Basically every entity in the CM became a table in this model.
2. Create a column for each attribute with at most one value(cardinality 0..1 or 1..1):
   Columns cannot have multiple values, therefore we just take care of single-valued attributes
3. Create a new table for each column with higher cardinality
   The solution to columns not being able to have multiple values is to have separate tables for those attributes with higher cardinality
4. Specify type for each column:
   Every single column needs to have a specified type; The ones we used were:
   - SERIAL - Unique and auto incremented by PostgreSQL
   - VARCHAR - String with a specified max-length
   - INT - Integer
   - TIMESTAMP holds date and/or time

5. Consider column constraints

6. Assign PK to all strong entities

7. Create one-to-one and one-to-many relationships
   These relationships are quite easy since one relationship line can describe it

8. Create a cross-reference table for each many-to-many relation
   Many-to-many relationships are handled by a separate table that holds foreign keys to both tables it is connected to

9. Assign FK to tables representing multi-valued attributes, created in bullet three. PK of these tables is the FK and the multi-valued attribute combined

10. Verify that the model is normalized
    Since we already had examined that our model should be normalized before the conversion, this step was partially skipped. However, we did briefly double checked that we have a normalized model

11. Verify that it is possible to perform all planned operations on the data
    Verification was done by inserting data and running different kinds of queries.

4 Result

The final model as well as the SQL scripts for both generating the database and inserting data into it, is found in the following GitHub repository:
`https://github.com/Rezaavoor/Soundgood-Music/tree/master/Task2`

And you can see the final result of the model in figure 1:

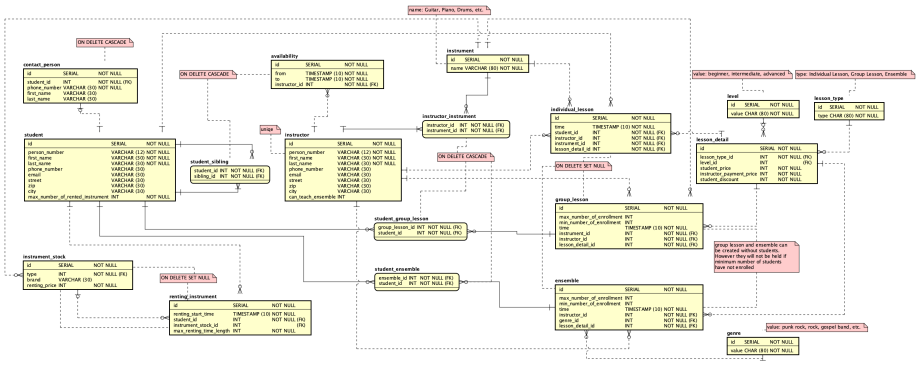

Figure 1: The Logical and Physical Model with no inheritance

The Logical and Physical model that was created consists of 17 different tables which are briefly explained below:

- student:
  Holds all the data. Person number is unique. Can have zero or multiple contact persons. Can rent 0, 1 or 2 instruments at a time

- contact person:
  holds the necessary data of a contact person to a student. Has a FK to a student

- renting instrument:
  all the instruments that are in the stock of the school and are meant to be rented do have their own table here. An instrument has a FK to a student but it can be null too. A null value on a student means it is not rented by anyone and is available. renting start time is also by default null unless a student rents the insturment, then the start time gets the value of the current time at that moment

- student sibling:
  students can have sibling relationship with each other through the cross table student sibling. Once a student, who has a sibling in the system, registers itself we create two student sibling rows where in one of them the student is stored as student and on the other one he/she is stored as sibling.

- instructor:
  Almost the same as student. Has a boolean column showing if the he/she can teach ensemble or not which will be used by the Admin who schedules the ensemble lessons. Can be available at different time periods or can be unavailable. Can take zero or more individual, group and ensemble lessons. Can at least one instrument but can also more.

- availability:
  Specifies a time period which shows availability of an instructor. Has a FK to an instructor.

- instrument
  Works kind of like a lookup table. Each instrument that is teached in the scholl has its own table.

- instructor instrument:
  cross-reference table meaning, one teacher can teach many instruments and also one instrument can be taught by many teachers.

- individual lesson:
  Have a specified time, level, student, instructor and instrument.

- group lesson:
  Have a specified time, level, instructor, instrument and min/max number of enrollments. Note that a group lesson can be created without any enrollments but will not be held if the minimum number of enrollments is not achieved. Also note that every table is for a single group lesson session.

- ensemble:
  Similar to group lesson. Has a specified genre.

- level:
  A lookup table for different levels. At this moment there are three different levels: beginner, intermediate, advanced

- student group lesson:
  cross-reference table between student and group lesson

- student ensemble:
  cross-reference table between student and ensemble

5 Discussion

Things to note about this task:

- Create data for database:
  To create data and insert it into the database, we used the website `https://generatedata.com/` to generate data for our students and instructors. For other parts of the database the data was created manually.

  One thing to note is that the data should follow other rules of the database otherwise you end up with a database that says against itself. For instance, a student can only rent maximum of 2 instruments at a time. Of course you can technically add more to a student using SQL queries but that will cause problems. So ideally you should only change the database data using a front-end application that automatically follows the extra rules.

- Enumerated Type:
  There are multiple ways of handling enumerated types. The way we chose to do it was by creating an extra table which holds different values of the type. In our case there are only 3 instances of the table `level` which have the values `Beginner, Intermediate and Advanced`. Other tables have conneciton with this table using FKs.
  The advantage of this approach is that it is universal through all DBMS and works with all of them. In comparison, you have the option of creating ENUM types in PostgreSQL which then works exactly as other normal types. But one disadvantage with that is that updating it means you have to change the database again while using the first approach, all you need to do is just creating another instance of the table and it does not bother the old data and tables at all.

- Relation to self:
  In our model there is a many-to-many relationship between students which may be a bit confusing. But all it says is that a student can have multiple sibling-relationships and at the same time be sibling to multiple people (even confusing when writing it down!). The only downside of this approach is that for each sibling you have to create two instances of the "student sibling" table and on the second one, swap the studentId with siblingId. Doing that makes it possible to have full control of all siblings of all students.
  Alternatively you can only create one instance per sibling. But now you have to be careful when checking if a student has a sibling or not; You need to check if that student is in the sibling column or not as well to get correct results:

```
SELECT * FROM student_sibling where student_id=1 OR sibling_id=1;
```

- Evaluate all required operations
  In order to evaluate the database we ran different queries and checked the results. The application pgAdmin4 was a huge help to do so since it had a nice GUI and showed the results in visually nice tables.
  Here are some examples we ran:

  – Show who is taking group lessons, what instrument, what level and when:

```
select
    s.first_name as name,
    i.name as instrument,
    l.value as level,
    g.time as time
from
    student as s join student_group_lesson as sg on s.id=sg.student_id
    join group_lesson as g on sg.group_lesson_id=g.id
    join level as l on g.level_id=l.id
    join instrument as i on g.instrument_id=i.id;
```

  the result was as below:

name	instrument	level	time
Jessamine	Guitar	Beginner	2022-11-28 19:10:25
Cairo	Guitar	Beginner	2022-11-28 19:10:25
Sopoline	Guitar	Beginner	2022-11-28 19:10:25

Table 1: List of students who are taking group lessons and its details

  The results show that 3 students are attending a beginner group lesson which uses guitar as instrument.

– Show all rented instruments and show who is renting them:

```
select
    i.type as name,
    i.brand as brand,
    s.first_name as name
from
renting_instrument as i join student as s on i.student_id=s.id;
```

instrumnet	brand	student
Guitar	Nisi LLP	Jessamine
Violin	Est Vitae Company	Jessamine
Piano	Faucibus Id Incorporated	Cairo
Guitar	Ac Foundation	Cairo

Table 2: List of rented instruments and its details

- Inheritance: In the same way we implemented the model using inheritance in the Conceptual model, we can also use the power of inheritance when defining the logical model and also creating the database. For experiment purposes we create a table called `person` which both `student` and `instructor` inherit from. In that case the code that creates our tables in the databse looks like this:

```
create table person(
    id serial PRIMARY KEY,
    person_number VARCHAR(80) UNIQUE NOT NULL,
    first_name VARCHAR(80) NOT NULL,
    last_name VARCHAR(80) NOT NULL,
    phone_number VARCHAR(80),
    email VARCHAR(80),
    street VARCHAR(80),
    zip VARCHAR(80),
    city VARCHAR(80)
);

create table student(
    max_number_of_rented_instruments INT NOT NULL
) inherits (person);

create table instructor(
    can_teach_ensemble BOOLEAN
) inherits (person);
```

A complete version of the model using inheritance is shown in Figure 2:

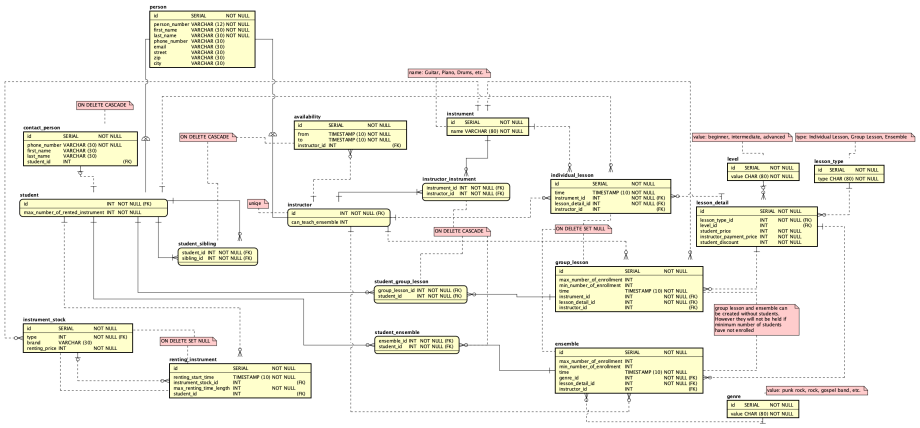

Figure 2: The Logical and Physical Model with inheritance

There are a couple of important things to consider here:

- One benefit of using inheritance in the Logical Model is that you do not store same information in two different tables. Since both Student and Instructor tables have properties like person-number, first-name, last-name and etc, it is better to merge those tables and let the shared data be stored in one place. As you can see in the model, the table Person stores all those shared data and Student and Instructor inherit from that and that makes those two tables a lot simpler with less repetitive properties.
- Another benefit of having inheritance could be that if an instructor(person) decides to learn a new instrument, we can see that in our database, meaning a person can be a instructor and a student at the same time. This is practical when for instance a instructor who teaches guitar decides to learn piano. In that case, that person is both a instructor and a student and you can see that in the model and the database.
- One downside of using inheritance, in the way we have implemented it, is that both student and instructor have person-id as their PK. That is not a big problem as information are still separated properly into two tables however it may cause confusions in whether an id is a student or a instructor. For instance if a student registers and gets the id=1, and an instructor right after registers, then the id for the instructor becomes id=2 since they both are person according to our Logical model. You can see this by looking at the model and seeing that both student table and instructor table have person-id as their primary keys.

Putting these all into consideration, we should be very very cautious when using inheritance in our databases which could make the maintenance of the database very difficult. Due to the issues above, we decided to drop the inheritance feature and went forward with a database without any inheritance.