Seminar 2

Axel Månson Lokrantz

2022-11-23

Introduction

The task for this seminar was to translate our conceptual model from seminar one into a logical and physical model with enough physical aspects to be able to create a database in SQL. Prior to solving the task, the conceptual model was adjusted according to the feedback we received. To pass the task, we were required to cover the entire description specified in the detailed description and use crow foot notation for all relations in the diagram. The diagram was created together with Daniel Dahlberg in UML modeling tool Astah at the royal institute of technology in Stockholm.

Literature Study

To prepare for this seminar I watched all the physical and logical model lectures on canvas, read the attached tips and tricks pdf and skimmed through the corresponding chapter 9.1 in Fundamentals of Database Systems. I also participated during the lecture about normalisation on how to make schema designs for a data base.

Method

Typically when creating a database the conceptual model is translated to two separate models, a logical and physical. In this task however, we were asked to create a simplified version using a mixture of the two. Just like the conceptual model the logical model describes what data to store, while the physical model describes how to store the data. The translation process can be divided into the following steps:

1. Create a table for each entity in the conceptual model.
2. Create a column for each attribute with at most one value (all attributes with a cardinality of `0..1` and `1..1`).

3. Create a table for the rest of the attributes (all attributes with a cardinality larger than one).

4. Specify the type for each column. In postgreSQL choosing `VARCHAR` over `CHAR` or text does not affect the performance. This means there is really no point in choosing `CHAR` over `VARCHAR`. It can be tempting to use `INT` for attributes such as ids, however, this can lead to problems for ids starting with a zero which is not taken into consideration.

5. Consider column constraints. In the case of `UNIQUE` no two rows can have the same value for a column and in the case of `NOT NULL` means the cardinality starts at 1.

6. Assign primary keys (PK) to all strong entities. A strong entity means, that it can exist on its own. It is preferred, to use surrogate keys for the strong entities because a surrogate key, which is a made up key that only exist in the database, will never change.

7. For all the one-to-one and one-to-many relations use the PK in the strong end and a foreign key (FK) in the weak end. Use a surrogate key in the weak end if the entity has meaning without the strong end. It is also required to consider FK constraints. Lets say we have an entity `lesson` and another entity `lesson _details`, if we were to delete the lesson entity there would be no meaning to keep the lesson details table, therefor we need to specify that it should be deleted.

8. Create a cross-reference-table to resolve all the many-to-many relations to avoid identical rows of data.

9. For all the tables which were created in step 3 create a composite PK with the FK and the multi valued attribute.

After verifying that the model is normalized and that all the necessary operations specified in the detailed description are possible we exported the model to sql and filled it with placeholder data from generatedata.com.

Results

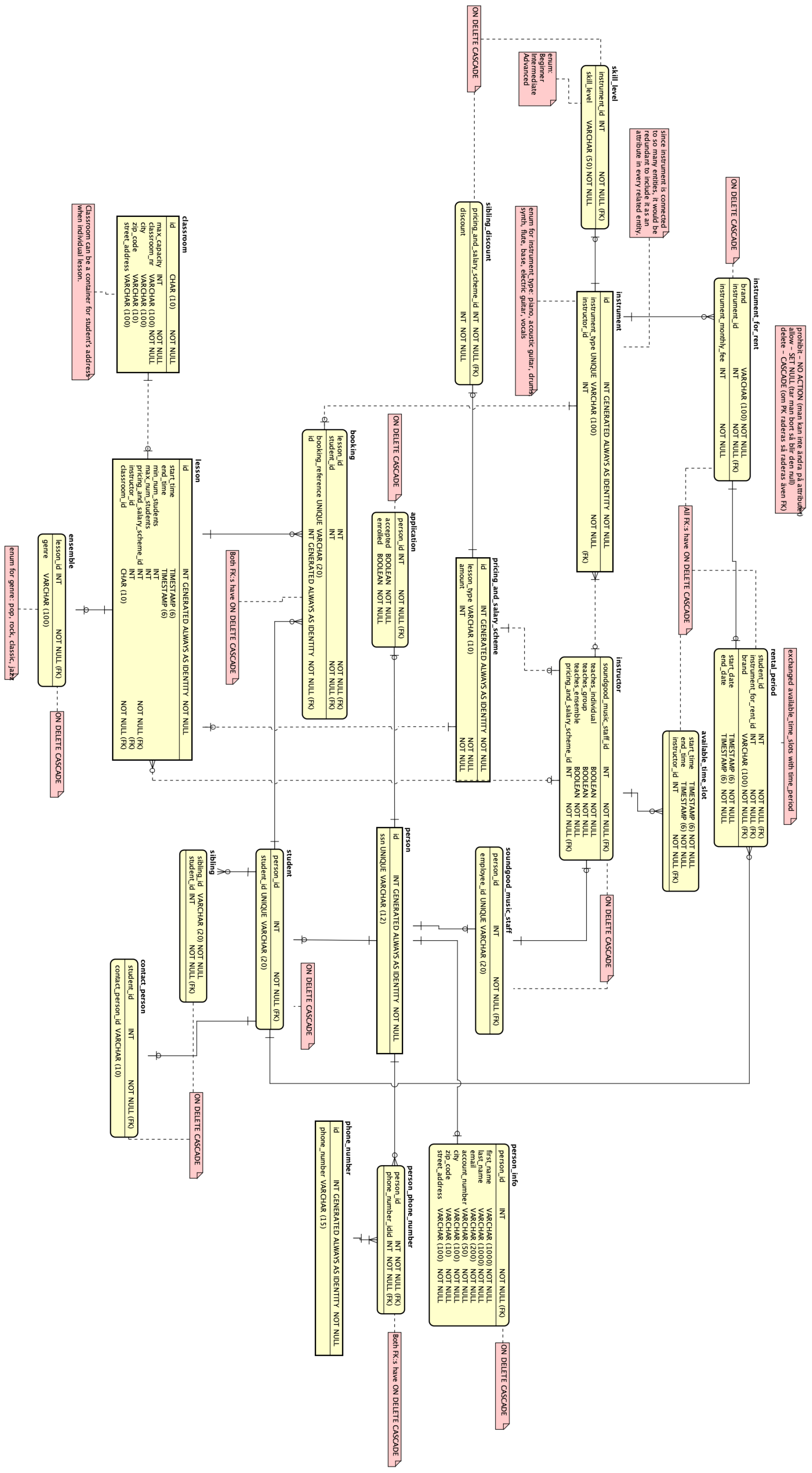

Figure 1: Logical and physical model of the SoundgoodMusic School database

In figure 1. the final translation of the logical/physical diagram can be seen. The code for our sql file can be found in the Git-hub repository section. All code is functional, with no major errors upon creation or insertion of data. The many-to-many relations from the conceptual model has been reworked according to bullet point 8 in the Method section. An example of such a relation can be seen in figure 2, where one person can have several phone numbers and a phone number can be used by several persons. Since the id of a person is used as PK, it must be unique, meaning one person can not appear in more than one row in the table. The solution is to implement a third entity which holds a FK to their respective tables.

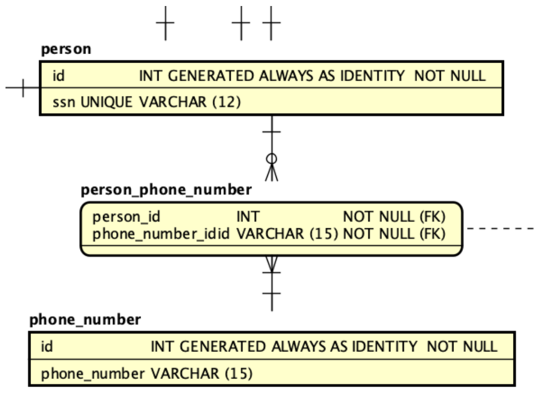
person
id INT GENERATED ALWAYS AS IDENTITY NOT NULL
ssn UNIQUE VARCHAR (12)
person_phone_number
person_id INT NOT NULL (FK)
phone_number_idid VARCHAR (15) NOT NULL (FK)
phone_number
id INT GENERATED ALWAYS AS IDENTITY NOT NULL
phone_number VARCHAR (15)

Figure 2: Solution to the many-to-many relation between the two tables person and phone.

Bullet point 3 specifies that any attribute with a cardinality higher than 1 should be transform into its own entity. An example of such an entity is `available_time_slots` which was previously part of the instructor entity. The problem is that if we were to only use the id as a PK an instructor could at most have one available time slot. To solve this, as described in bullet point 8, we include the multi-valued attribute, in this case `start_time` and `end_time` into our PK.

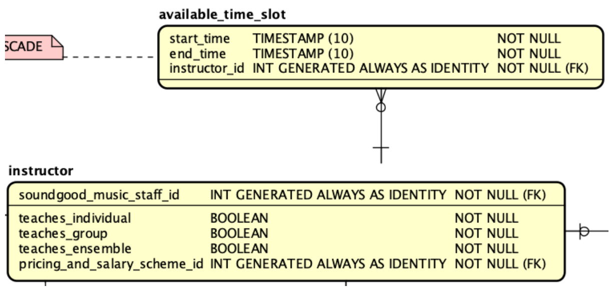
SCADE
available_time_slot
start_time TIMESTAMP (10) NOT NULL
end_time TIMESTAMP (10) NOT NULL
instructor_id INT GENERATED ALWAYS AS IDENTITY NOT NULL (FK)
instructor
soundgood_music_staff_id INT GENERATED ALWAYS AS IDENTITY NOT NULL (FK)
teaches_individual BOOLEAN NOT NULL
teaches_group BOOLEAN NOT NULL
teaches_ensemble BOOLEAN NOT NULL
pricing_and_salary_scheme_id INT GENERATED ALWAYS AS IDENTITY NOT NULL (FK)

Figure 3: Solution to multi-valued attribute in entity.

Git-hub repository

Link to project repository.

Discussion

Initially, both me and my partner were a little confused about the normalisation process of our diagram, even though the lectures were very clear on the subject. Things to consider going into first normal form was atomic values and functional dependencies. It slowly dawned on us that the conceptual model we built in seminar one, was in fact, already in first normal form. The attributes were atomic and all functional dependencies were already in place. Perhaps this had to do with the fact that we built the conceptual model without knowing all the theoretical terms prior to the task. Going into the second normal form we went through all our non-prime attributes to see if any of them could be derived from their corresponding prime-key. An obvious example is `rental_period`. Neither `start_date` nor `end_date` can be derived from its prime-key counterpart, which means that the table is already brought to 2NF. In the end we failed to find any tables that were not already brought to the 2NF. To reach the 3NF any attributes, or columns, that can be derived by another attribute must be resolved. There are several ways to approach the task, but what works best, in most situations, is to move the derived attribute into its own table. We managed to find one attribute which violated the 3NF, namely `ssn` in the table `person`. Several people may have the same name, surname or phone numbers but a social security number will always be unique. Therefor we reasoned that all columns in the table could be derived from the `ssn`. It is debatable whether it is possible to find a persons `email` or `bank_account_number` through their `ssn` but for the sake of simplicity we decided to decompose `person` and all its columns into a separate table which we named `person_info`. However, this implementation unveiled yet another problem. `person` is considered a strong entity, which means it can exist without the existence of a relation to another table. With this logic `person_info` becomes a weak entity, but what use do we, in a database, have of a person without without any data. The alternative would have been to revert our changes a thereby violating the 3NF.