Project Report Task 2

Data Storage Paradigms, IV1351

2024-11-21

Project members:
Jonathan Värild, varild@kth.se
Oscar Caddeo, ocaddeo@kth.se
Elias Holm, eliholm@kth.se

Declaration:

By submitting this assignment, it is hereby declared that all group members listed above have contributed to the solution. It is also declared that all project members fully understand all parts of the final solution and can explain it upon request.

It is furthermore declared that the solution below is a contribution by the project members only, and specifically that no part of the solution has been copied from any other source (except for lecture slides at the course IV1351), no part of the solution has been provided by someone not listed as a project member above, and no part of the solution has been generated by a system.

1 Introduction

During this assignment, the goal was to transform the conceptual model which was implemented during the previous assignment to a logical and physical model. While the conceptual model focuses on modeling reality by capturing the entities, relationships, and attributes of the Soundgood Music School's operations, the logical and physical model bridges the gap between abstraction and implementation. This refined model is designed to represent the structure and organization of the database, making it easy to translate the diagram to SQL.

2 Literature Study

3 Method

To create a Logical and Physical Model that accurately represents the requirements of the Soundgood Music School the diagram editor Astah Proffesional was used and the diagram made using IE notation. The Logical and Physical Model was created based on the previous Conceptual model and the approach of converting it consisted of several steps:

1. **Create tables and columns for entities and attributes**: Tables were created for each of the entities and columns within the tables for each attribute. For attributes with cardinality higher than one, separate tables had to be created but this had been considered when creating the Conceptual model so only one change had to be made with an additional table for siblings.

2. **Specify types and add constraints**: After all tables and columns had been created, types where specified for each column. All primary key id:s where set to 'INT ALWAYS GENERATED AS IDENTITY' for example. Constraints such as NOT NULL and UNIQUE where also added and cardinality specified between tables.

3. **Define Keys, relationships, Triggers, Checks and Constraints**: For all tables primary keys where defined as well as foreign keys to form relationships between tables. Checks, constraints and triggers where also added to make sure inserted data does not conflict with the requirements. For example a trigger that rejects rental inserts if a student already has 2 active rentals and a trigger that makes sure one instructor can't be booked on different activities during the same timeslot.

4. **Normalization**: Finally the model was normalized to 3NF since the group decided that was sufficient for the model and any more normalization would only complicate the model. It was firstly normalized to 1NF by making sure the model only contains atomic values, each column only contains one value of a single type and each table has a unique primary key identifier. To normalize it to 2NF it was made sure that the model does not contain any partial dependecies. The final step to normalize it from 2NF to 3NF was to remove any transitive dependencies.

5. **Creating the database**: When the model was complete, a database was created in accordance with the model. Creating all the tables, columns as well as all checks, constraints and triggers. After that some mock data was created and inserted into the database to test and the scripts for both creating the database and inserting mock data where revised and then uploaded. With the database and mock data now in place, the database was tested to make sure all constraints worked as intended and that the data the requirements lists were able to be retrieved from the database.

4 Result

The GitHub repository with Astah files, SQL files, etc., can be found at `https://github.com/Cadde0/IV1351-Project`.

The result of our combined physical and logical diagram of our database can be seen in Figure 1 below. Its structure is very alike to our Conceptual Model Diagram since it already was fairly normalized and compatible with a database structure. The first table that we implemented was the **person** which uses an automatically incremented integer ID **school_id** as primary key. It is automatically incremented using the built-in SQL feature **GENERATED ALWAYS AS IDENTITY** which ensures that we can't insert values to it. The **school_id** is widely used in the database since it is used to identify different persons. Their personal ID number is only used to identify their **Person** row with their real life identity. This ensures that we in special cases can update the personal ID number of a person if it were to change for some reason. The table have two checks using regex to ensure that the **personal_id_number** and **zip_code** are correctly formatted.

The **student** table has a relationship with **contact_details** which store personal or relative contact details for each person. Each contact details have a unique ID that is also **GENERATED ALWAYS AS IDENTITY**. This is also the primary key of each contact details. Since we don't require both a phone number and an email address we have added a check that ensures they are not null at the same time.

Next, we have the tables **instructor** and **student** which uses a foreign key to person which connects them. This allows us to divide each person into roles and also apply additional columns or relations to each role such as **can_teach_ensamble** or the table **sibling**. The **instructor** and **student** row will be automatically deleted if the **person** reference is deleted since it is defined as **ON DELETE CASCADE**.

One of our main issues with the Conceptual Diagram Model was that we didn't think about that 3 students with 3 different parents may not be siblings with everyone. For example, student 1 may be sibling with student 2 which share the same mother and the same father. At the same time, student 3 may also be sibling with student 2 by sharing father. Sibling 1 and 2 may however have different fathers and mothers which don't make them siblings. This made it impossible for them all to share a single sibling ID that groups them together. To fix this, we implemented a new table called **sibling** that takes two foreign keys which both are **student_school_id**s which can link two students together as siblings. To ensure that you cannot insert bidirectional data we created a trigger which automatically ensures the first school ID is the smallest one and the second school ID is the largest one. These two IDs build the primary key which ensures only one sibling relation can be saved for two students. We also added a check to ensure that a student cannot be a sibling with themselves. If any of the students are removed from the database, the sibling relationship is automatically removed since both foreign keys are defined as **ON DELETE CASCADE**.

The **instructor** table has a relationship with **instrument_skill** which defines what instrument skills each instructor has. The **instrument_skill** table in turn has a foreign key to **instructor** which links them together. If the **instructor** row is deleted, so is the

instrument_skill row since the foreign key is defined as **ON DELETE CASCADE**. The **skill_level** column also has a check which enforces that the value is between 1 and 3 indicating the different skill levels beginner, intermediate, and advanced. It also has a foreign key to **instrument_type** which helps us keep a clear definition of what instrument it is. If the instrument type is removed the instrument skill is also removed since it also is defined as **ON DELETE CASCADE**.

We also created a table called **instrument_type** which store information about all instrument types that are available. All instrument types have their own ID that can be referenced from other tables using a foreign key. It can then be connected to the instrument type name which is presented to the user, etc.

Next, we have the **activity** table which store general information about all types of activities like individual lessons, group lessons and ensembles. This table also has a generated integer ID. It has a check to ensure that the start and end time of each activity is in the correct order. The **activity** table also has relationships with the **lesson_individual**, **lesson_group**, and **ensamble** tables. These tables help to group each activity into the different types. This is also a small change from our conceptual model diagram which used Lesson to define both an individual and group lesson but with different values to their min and max student attributes. To ensure that we don't have a table with a lot of NULL values for min and max students, we decided to split the tables up instead which removes this need. Both **lesson_group** and **ensamble** share checks to ensure the min students are above 1 and that max students is larger that the min students. All 3 tables share **ON DELETE CASCADE** on **activity_id** which ensures that their row is deleted if the activity row is deleted. The tables **lesson_individual** and **lesson_group** share the check to ensure that the skill level is between 1 and 3 in the same way as it's defined for the **instrument_skill** table. The **activity** table has **ON DELETE RESTRICT** on all foreign keys to ensure that these are not removed before all their references are replaced or removed. The **activity** table also have a trigger that ensures a instructor isn't booked on overlapping times since the instructor can't be at two places at the same time. The **lesson_individual** and **lesson_group** keep a foreign key to **instrument_type** to keep track of what instrument it is. If you try to remove the instrument type it will be stopped due to **ON DELETE RESTRICT**. Instead, you would need to replace the reference before trying to remove the instrument type.

The **activity** table also has a relationship to the **location** table which is a table of all possible locations where you may host an activity. The **location** table has a check to ensure that not both **room_name** or **video_link** are NULL since a location may be either physical, online, or both.

Next we have the **pricing** table which hold pricing information for each activity type based on its type and skill level. It also holds extra information such as sibling discount percentage, if its active and a unique ID. This fulfills the requirement to be able and update the price of a certain activity but still be able to see the old price after its updated. Each activity store the id to the pricing row that was active at that time. If you wish to update the pricing you simply set the current row's to active column to false and insert a new row matching the same activity and skill level. The **pricing** table also has a number of checks such as that **activity_type** is between 1 and 3 which indicates individual lesson, group lesson, and ensemble. It has a check to ensure skill level is between 1 and 3, that the price isn't negative, and that the sibling discount is a number between 1-100 indicated by a discount percentage.

The **booking** table has a relationship with both the **activity** and the **student** table and is used to store what activities each student is booked for. Both foreign keys are set to **ON DELETE CASCADE** which ensures that a booking is removed if its student or activity is removed.

Looking in the other direction of **person** we have the **rental** table which keeps track of all rentals made by each student. It has foreign keys to the **student school_id** and **instrument_inventory inventory_id** which are defined as **ON DELETE CASCADE** since there is no meaning to keep track of a rental that cannot be linked to a **person** or **instrument_inventory**. The table is linked to person to ensure that both students and instructors should be able to make rentals. To fulfill the requirement on that each student only should be able to have two active rentals we have added a trigger that checks this before a row is inserted or updated. Another check ensures that the rental time does not exceed 12 months.

Finally, we have the **instrument_inventory** table which store information about all instruments that the school has in inventory available for rental. All instruments of the same brand and model are stored in the same row and are instead kept track of using a **quantity** column that defines how many instruments there are. The table has two checks to ensure that the quantity and price is over 0. It also references **instrument_type** using a foreign key with **ON DELETE RESTRICT**. This ensures that you cannot remove an instrument type that we keep track of in our instrument inventory.

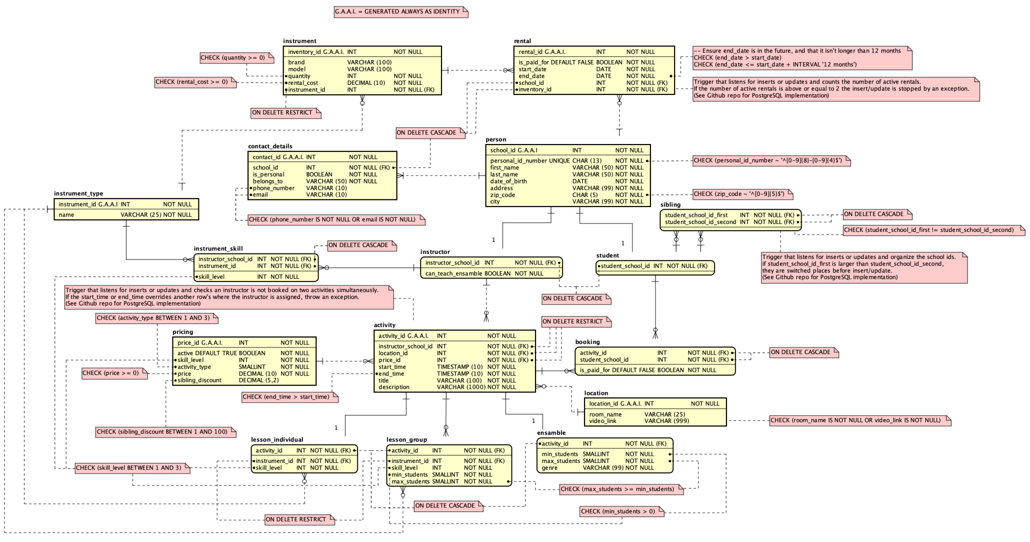
G.A.A.I. = GENERATED ALWAYS AS IDENTITY
instrument
inventory_id G.A.A.I. INT NOT NULL
brand VARCHAR (100)
model VARCHAR (100)
quantity INT NOT NULL
rental_cost DECIMAL (10) NOT NULL
instrument_id INT NOT NULL (FK)
CHECK (quantity >= 0)
CHECK (rental_cost >= 0)
ON DELETE RESTRICT
rental
rental_id G.A.A.I. INT NOT NULL
is_paid_for DEFAULT FALSE BOOLEAN NOT NULL
start_date DATE NOT NULL
end_date DATE NOT NULL
school_id INT NOT NULL (FK)
inventory_id INT NOT NULL (FK)
-- Ensure end_date is in the future, and that it isn't longer than 12 months
CHECK (end_date > start_date)
CHECK (end_date <= start_date + INTERVAL '12 months')
Trigger that listens for inserts or updates and counts the number of active rentals.
If the number of active rentals is above or equal to 2 the insert/update is stopped by an exception.
(See Github repo for PostgreSQL implementation)
ON DELETE CASCADE
contact_details
contact_id G.A.A.I. INT NOT NULL
school_id INT NOT NULL (FK)
is_personal BOOLEAN NOT NULL
belongs_to VARCHAR (50) NOT NULL
phone_number VARCHAR (10)
email VARCHAR (10)
CHECK (phone_number IS NOT NULL OR email IS NOT NULL)
person
school_id G.A.A.I INT NOT NULL
personal_id_number UNIQUE CHAR (13) NOT NULL
first_name VARCHAR (50) NOT NULL
last_name VARCHAR (50) NOT NULL
date_of_birth DATE NOT NULL
address VARCHAR (99) NOT NULL
zip_code CHAR (5) NOT NULL
city VARCHAR (99) NOT NULL
CHECK (personal_id_number ~ '^[0-9]{8}-[0-9]{4}$')
CHECK (zip_code ~ '^[0-9]{5}$')
instrument_type
instrument_id G.A.A.I INT NOT NULL
name VARCHAR (25) NOT NULL
sibling
student_school_id_first INT NOT NULL (FK)
student_school_id_second INT NOT NULL (FK)
ON DELETE CASCADE
CHECK (student_school_id_first != student_school_id_second)
Trigger that listens for inserts or updates and organize the school ids.
if student_school_id_first is larger than student_school_id_second,
they are switched places before insert/update.
(See Github repo for PostgreSQL implementation)
instrument_skill
instructor_school_id INT NOT NULL (FK)
instrument_id INT NOT NULL (FK)
skill_level INT NOT NULL
ON DELETE CASCADE
instructor
instructor_school_id INT NOT NULL (FK)
can_teach_ensamble BOOLEAN NOT NULL
student
student_school_id INT NOT NULL (FK)
1
1
ON DELETE CASCADE
ON DELETE RESTRICT
Trigger that listens for inserts or updates and checks an instructor is not booked on two activities simultaneously.
If the start_time or end_time overrides another row's where the instructor is assigned, throw an exception.
(See Github repo for PostgreSQL implementation)
CHECK (activity_type BETWEEN 1 AND 3)
pricing
price_id G.A.A.I. INT NOT NULL
active DEFAULT TRUE BOOLEAN NOT NULL
skill_level INT NOT NULL
activity_type SMALLINT NOT NULL
price DECIMAL (10) NOT NULL
sibling_discount DECIMAL (5,2)
CHECK (price >= 0)
CHECK (sibling_discount BETWEEN 1 AND 100)
activity
activity_id G.A.A.I. INT NOT NULL
instructor_school_id INT NOT NULL (FK)
location_id INT NOT NULL (FK)
price_id INT NOT NULL (FK)
start_time TIMESTAMP (10) NOT NULL
end_time TIMESTAMP (10) NOT NULL
title VARCHAR (100) NOT NULL
description VARCHAR (1000) NOT NULL
CHECK (end_time > start_time)
booking
activity_id INT NOT NULL (FK)
student_school_id INT NOT NULL (FK)
is_paid_for DEFAULT FALSE BOOLEAN NOT NULL
ON DELETE CASCADE
location
location_id G.A.A.I. INT NOT NULL
room_name VARCHAR (25)
video_link VARCHAR (999)
CHECK (room_name IS NOT NULL OR video_link IS NOT NULL)
1
1
lesson_individual
activity_id INT NOT NULL (FK)
instrument_id INT NOT NULL (FK)
skill_level INT NOT NULL
lesson_group
activity_id INT NOT NULL (FK)
instrument_id INT NOT NULL (FK)
skill_level INT NOT NULL
min_students SMALLINT NOT NULL
max_students SMALLINT NOT NULL
ensamble
activity_id INT NOT NULL (FK)
min_students SMALLINT NOT NULL
max_students SMALLINT NOT NULL
genre VARCHAR (99) NOT NULL
CHECK (skill_level BETWEEN 1 AND 3)
ON DELETE RESTRICT
ON DELETE CASCADE
CHECK (max_students >= min_students)
CHECK (min_students > 0)

Figure 1: The physical/logical diagram that we came up with.

5 Discussion

5.1 Are naming conventions followed? Are all names sufficiently explaining?

All naming conventions are followed strictly. This means that all table names and column names follow snake case. This was the recommended naming conventions presented in the lectures.

5.2 Is the crow foot notation correctly followed?

The crow foot notation is followed correctly since all relationships have the correct cardinality and the table columns reflect the relationships in a correct way.

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

The model has been made to fully follow the 3NF normalization standard. This can be said since all columns hold unique values, each cell contains a single value, each row is uniquely identifiable by its primary key, there's no partial dependency, there's no transitive dependency.

5.4 Are all tables relevant? Is some table missing?

All tables are relevant to store the data needed to fulfill the requirements posed by the assignment. There are no tables that cannot be connected to a certain requirement of the assignment.

5.5 Are there columns for all data that shall be stored? Are all relevant column constraints and foreign key constraints specified? Can all column types be motivated?

All columns are there to fulfill the requirements posed by the assignment. Their data type has been carefully selected to follow the same requirements or general standards for that type of data. All columns that need to be constrained has been done so with checks or triggers. Foreign key constraints are automatically rendered by Astah, modified if needed, and finally verified to be correct.

5.6 Can the choice of primary keys be motivated? Are primary keys unique?

All primary keys or composite primary keys have been carefully selected to ensure that they will be unique. Eventual composite primary keys that would not be unique reflects data that is not wanted. For the sibling table we avoid bidirectional composite primary keys using a trigger that ensure each combination of student IDs only can occur once, regardless of their order.

5.7 Are all relations relevant? Is some relation missing? Is the cardinality correct?

All relations are explicitly needed and there are no relations which are unnecessary. As mentioned earlier, all cardinality have also been carefully selected and is correct. There are no relations missing and this can be verified since the database has been implemented and tested.

5.8 Is it possible to perform all tasks listed in the project description?

The database has been carefully planned to fulfill exactly this requirement. All tables and their relations are chosen based on the actions that we need to perform on the data later on.

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

Notes have been used in the diagram to explain triggers, checks, ON DELETE actions, etc. For the trigger descriptions we refer to our GitHub if an example of a full implementation is requested.

5.10 Are there attributes which are calculated from other attributes and then written back to the database (derived attributes)? If so, why? Records of student fees and instructor payments might be examples of such attributes.

5.11 Are tables (or ENUMs) always used instead of free text for constants such as the skill levels (beginner, intermediate and advanced)?

Initially we used ENUMs for the instrument types but was recommended by a lab assistant to use INT IDs or a lookup table since ENUMs were said to not scale very efficiently. Therefore, IDs like 1 for beginner, 2 for intermediate and 3 for advanced are used for skill level. For instrument types we used the table **instrument_type** which defines an ID and name for all instrument types. Other tables like **instrument_skill**, **instrument_inventory**, **lesson_individual**, and **lesson_group** can then reference these instrument types using a foreign key.

5.12 Is the method and result explained in the report? Is there a discussion? Is the discussion relevant?

5.13 Pros/cons of storing all data in the database compared to application

We have decided to include all data required by Soundgood in the database. This means that rentals are enforced to two active rentals, double bookings are avoided, music instrument types are stored in the database, strict checks are done for all data, etc. One of the main advantages of doing this is that logical errors are a lot less likely to happen. For example, the price for a lesson or rental cannot be negative which efficiently would give the person money instead of charging them. The negative aspect is however that the database has to process more information which causes additional strain on the database. The same functionality can and should be implemented in the application as well but implementing it in both ensures that all data is what we expect it to be. Basically, we are a lot less likely to encounter database faults later when its in use.

5.14 Pros/cons of storing multiple versions of data in the database

The main disadvantage of storing multiple versions of data in the database is of course that more storage is used. However, it can be very useful for looking into historical data like requested by the assignment. For example when you want to look at price history or want to change the price of something without affecting those who have already made a booking at another price. More data can also slow down the database if excessive. If you need to keep a lot of historical data it might be wise to remove old data when it isn't needed. Alternatively you can separate your production database with a history database if the historical data isn't needed as often. This causes less strain on the production database.

6 Comments About the Course