Introduction to Database Design, F2016
IT University of Copenhagen
Carsten Schürmann

Project 3

This project assignment is a group assignment.

Wednesday, 23 November 2016, 23:55

Rules:

- Submit your groups work as a single PDF file with a file name that includes the groups name.
- Upload your work on `learnit.itu.dk`.
- Late hand-ins will not be considered.
- The grade of all project assignments will also be averaged and determine the grade of assignment 4, which also counts with 15% towards your grade.

The goal of this project is to develop a simple video-on-demand application. This is a typical application using a database: the only thing we are missing is a Web interface.

You sell video streams to customers. You have a contract with a content provider that has videos of all the movies in the IMDB database, and you resell this content to your customers. Your first task is to create a database of your customers. Next, your application allows the user to search the IMDB movie database, and to rent movies (which we assume are delivered by the content provider; we don't do this part in the project). Once a customer rents a movie, she can watch it as many times as she wants, until she decides to "return" it to your store. You need to keep track of which customers are currently renting which movies.

There are two important restrictions:

Your contract with the content provider allows you to rent each movie to at most one customer at any one time: the movie needs to be first returned before you may rent it again to another customer (or the same customer). Your own business model imposes a second important restriction: your store is based on subscription (much like Netflix), that allows customers to rent up to a 3 movies. Once they reach that number you will deny them more rentals, until they return a movie.

Task 1:

Design and create the CUSTOMER database in mysql. The database has the following entity sets:

Customer: a customer has an id (integer), a login, a password, a first name and a last name.

Rental: a "rental" entity represents the fact that a movie was rented by a customer with a customer id. The movie is identified by a movie id (from the IMDB database). The rental has a status that can be open, or closed. When a customer first rents a movie, then you create an open entry in Rentals; when he returns it you update it to closed (you don't delete it). Keeping the rental history helps you improve your business by doing data mining.

In addition there is the following relationships:

Rented: Each rental refers to exactly one customer.

Hand-in requirement. Upload a single text file called "setup.sql" with CREATE TABLE and INSERT statements for this database to LearnIT.

Task 2:

The application is a simple command-line Java program. A "real" application will have a Web interface instead, but that requires a lot of programming that is unrelated to the database content of this course. Your Java application needs to connect to two databases: the IMDB database and your own CUSTOMER database.

When your application starts, it reads a customer name and password from the command line. It validates them against the database, then retains the customer id throughout the session. All rentals/returns are on behalf of this single customer: to change the customer you will quit the application and restart it with another customer. Much of this logic is already provided in the starter code. For the most part, all you need to do is uncomment the code that performs the authentication and modify it to match your CUSTOMER schema.

Once the application is started, the user can select one of the following transactions. (We call each action a transaction. You will need to write some of them as SQL transactions. Others are interactions with the database that do not require transactions.)

The "search" transaction: the user types in a string, and you return:

- all movies whose title matches the string
- their director(s)
- their actor(s)
- an indication of whether the movie is available for rental (remember that you can rent the movie to only one customer at a time), or whether the movie is already rented by this customer (some customers forget: be nice to them), or whether it is unavailable (rented by someone else).

The "rent" transaction: The user types in a movie id, and you will "rent" that movie to the customer.

The "return" transaction. The user types in the movie id to be returned. You update your records to mark the return. You will need to use transactions, commits and rollbacks.

In addition you have to provide a minimal amount of user-friendliness: at each iteration of the main loop you will print the current customer's name, and tell them how many additional movies they can rent (given their current plan and the number of movies that they have already rented).

Hints:

- You need to enforce the following constraints:

  1. (C1) At any time a movie can be rented to at most one customer.

2. (C2) At any time a customer can have at most as many movies rented as his/her plan allows.

- What these constraints imply iswhen a customer requests to rent a movie, you may need to deny this request. Thus, you need to worry about C in ACID. Transactions will help.
- You also need to worry about concurrency. A user may try to cheat and coerce your application to violate the constraint C2 above by running two instances of your application in parallel, with the same user id: depending on how you write your application and on race conditions, the malicious user may succeed in renting more movies than he/she is allowed. You need to ensure that each instance of your application runs in isolation, i.e. the I in ACID. You will need transaction to finish this.
- You also need to worry about the A in ACID: what if you lose your CUSTOMER database, or it becomes inconsistent as a result of a systems crash?
- Never include a user interaction inside a transaction! That is, don't begin a transaction then wait for the user to decide what she wants to do (why?). Your transactions should not include any user interactions.
- You need transactions only on the CUSTOMER database: you don't need transactions for IMDB, since this database is never updated.
- Don't try to start a transaction inside another transaction! MySQL doesn't support nesting transactions, and will ignore attempts to run START TRANSACTION while already in a transaction. It will not ignore COMMIT or ROLLBACK, however, which means that if you (accidentally or purposefully) nest transactions, you could end up committing or aborting a transaction earlier than you intended.
- Here is the method to execute multi-statement transactions from Java.

```
Connection _db;
_db.setAutoCommit(false);

[... execute updates and queries.]

 _db.commit();
OR
 _db.rollback();
```