Part 2 (P2): Transactions and interfacing with applications

This part of the project uses the IMDB database from homework assignment 1, and the CUSTOMER database that you designed in the previous part of your group assignment.

You will learn how to interact with a database through a Java application, and how to implement transactions for your video store. We provide for you some starter code in our download folder, implementing some very basic functionality for your video store: **P2-VideoStoreStarterCode.zip**

Background

You have designed the client database for your store, and now it is time to implement some basic functionality for your application! Your task is to complete a working prototype of your video store application that connects to the database then allows the customer to use a command-line interface to search, rent, and return movies. We have already provided code for a complete UI and partial back end; you will implement the rest of the back end. In real-life, you would develop a Web-based interface instead of a command-line interface, but we use a command-line interface to simplify this assignment.

Remember the two important restrictions that you need to enforce:

1. Because your store is brand-new, your contract with the content provider will only allow 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).
2. Your own business model imposes a second important restriction: your store is based on subscriptions (much like Netflix), allowing customers to rent up to a maximum number of movies for as long as they want. Once they reach that number you will deny them more rentals, until they return a movie. You offer a few different rental plans, each with its own monthly fee and maximum number of movies.

A. Initial Setup

Your system will be a Java application. Download the **P2-VideoStoreStarterCode.zip**. You should see the following files:

- `VideoStore.java` : the command-line interface to your video store; calls into `Query.java` to run customer transactions
- `Query.java` : code to run customer transactions against your database, such as renting and returning movies
- `dbconn.config` : a file containing settings to connect to the customer and IMDB databases. You need to edit it before running the starter code.

- `postgresql-9.2-1002.jdbc4.jar` : the JDBC to the PostgreSQL driver. You may prefer to download the latest version of JDBC at https://jdbc.postgresql.org/. This tells Java how to connect to the postgres database server, and needs to be in your CLASSPATH (see below)
- `setup.sql` : contains a sample setup for the CUSTOMER database. You should not need this file, since you created your own, more elaborate version of the CUSTOMER database for the previous group assignment. Your version **should** differ from ours; in fact, we expect that each team's version of the customer database will be different. We expect you to adapt your customer database and Query.java so that they follow your design from part 1.

At this point, we assume that you have Java installed on your machine, and the IMDB and CUSTOMER databases already set up. Modify `dbconn.config` with the username and password that you use for PostgreSQL. This will allow your java program to connect to your postgres databases.

You are now ready to try the starter code. Please follow the instructions for your platform as shown in the table below. The last command launches the starter code. Normally, you run it like this:

```
java VideoStore wolfgang wolfgangpassword
```

In the command above, you provide the username and password of the **video store user**. Later, your code will check the name and password of the customer, but the starter code ignores both `user` and `password` at this point (you can put any strings you like).

Windows

```
cd \where\you\unzipped\the\starter\code
    [replace the directory below with your JDK's bin\ directory]
path C:\Program Files\Java\jdk1.6.0_25\bin;%path%
set CLASSPATH=.;postgresql-9.2-1002.jdbc4.jar
javac -g VideoStore.java Query.java
java VideoStore user password
```

Linux or Mac

```
cd /where/you/unzipped/the/starter/code

export CLASSPATH=.:postgresql-9.2-1002.jdbc4.jar
javac -g VideoStore.java Query.java
java VideoStore user password
```

If you are getting an error about the JDBC driver when running the starter code, try different versions of the driver, which you can download at http://jdbc.postgresql.org/download.html .

Now you should see the command-line prompt for your video store:

```
 *** Please enter one of the following commands ***
> search <movie title>
> plan [<plan id>]
> rent <movie id>
> return [<movie id>]
> fastsearch <movie title>
> quit
>
```

The `search` command works (sort of). Try typing:

```
search Nixon
```

After a few seconds, you should start getting movie titles containing the word 'Nixon', and their directors. (You don't yet get the actors: one of your jobs is to list the actors.)

Objective:
Your task is to write the Java application that your customers will use, by completing the starter code. You need to modify only `Query.java`. Do not modify `VideoStore.java`, because we will test your homework using the current version of `VideoStore.java`. The application is a simple command-line Java program. A "real" application will have a Web interface instead, but this is beyond the scope of this assignment. (You will have the chance to implement additional features, such as a nicer front-end in part 3 of the project.) Your Java application will connect to both the IMDB and the CUSTOMER databases on postgres.

When your application starts, it reads a customer username and password from the command line. It validates them against the database, then retains the customer id throughout the session. All actions (rentals/returns etc) are on behalf of this single customer: to change the customer you must quit the application and restart it with another customer. The authentication logic is not yet wired up in the starter code; one of your tasks will be to make it work.

Once the application is started, the customer can select one of the following commands:

- Search for movies by words or strings in the title name.
- View a list of rental subscription plans, and change his/her plan.
- Rent a movie by IMDB ID number.
- Return a rented movie, again by IMDB ID number.

To complete your application, you will do the following:

1. Complete the provided IMDB movie search function.
2. Write a new, faster version of the search function.
3. Implement the remaining functions in `Query.java` to read and write customer data from your database, taking care to ensure atomic transaction semantics.

User Authentication:
In its initial state, the application does not check the user's login information, and ignores the provided username and password. In order to enable user authentication, search for and uncomment the relevant lines of code in `Query.java`. There are three such locations in the starter code, relating to the user authentication functionality.

NOTE:
You should build your application around the database schema that you designed in part 1 of the project. However, it is possible that you will recognize some shortcomings of your schema, through the application development. It is ok to make revisions to your schema in that case, such as adding additional fields to tables. In that case, please submit also an updated "**... - ER.pptx**" file with your submission. Again, we provide you with the code in setup.sql only as an example, and to help you get the video application working faster, but we expect you to adapt the code to work with your own DB design.

B. Complete the search function [15 points]

In the search command, the user types in a string, and you return:

- all movies whose title matches the string. Can you make the search case-insensitive?
- the director(s) of each movie
- the actor(s) that appear in each movie
- 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 starter code already returns the movies and directors. Your task is to return all actors, and also to indicate whether the movie is available for rental.

C. Improve the search function (fastsearch) [25 points]

When writing programs that talk to a back-end DBMS it is possible to implement some of the data processing either in the Java code or in the SQL statements. In the first case we issue many SQL queries and perform some of the query processing in Java; in the latter case we issue only one (or a small number) of SQL queries and push most of the work to the database engine. We used the former in task B; we will use the latter in task C. In this task you will reimplement the search function from B but instead of dependent joins you will push the joins to the database engine. In principle, this should be faster, however, you may not necessarily notice that: on our small database the speed is affected much more dramatically by whether you are running with a cold cache, or a hot cache.

You will implement the fastsearch functionality, by using joins computed in the database engine, instead of the dependent joins computed in Java by the search function. As you see in `VideoStore.java`, the fastsearch command in the interface invokes `transaction_fast_search` from `Query.java`, so that is the method that you should modify. Your fastsearch should return only (1) the movie information (id, title, year), (2) its actors, and (3) its director. It does not need to return the rental status. Notice that search issues O(n) SQL queries, because for each movie it runs a separate SQL query to find all its directors (and its actors). Instead, you will write fastsearch to issue only O(1) SQL queries, say two or three.

Hint: One query finds all movies matching the keyword; one query finds all directors of all movies matching the keyword; on query finds all the actors of all the movies matching the keyword. Execute each of these three queries separately. You then need to merge the results of the three queries *in* the Java code. The merge will be easier if your SQL queries sort the answers by the movie id. (There is also a way to write fastsearch with only two, or even only one single SQL query, but don't worry about that because it gets more messy with questionable benefits.)

Compare search and fastsearch with different movie titles, e.g., Nixon, nowhere, Harry Potter and the Chamber of Secrets, etc. When do you observe the most benefits?

D. Customer database transactions [60 points]

Now, complete the application by implementing each 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.)

1. The "login" transaction, which is run implicitly when you start your command line program, authenticating the user by his/her username and password. Much of the authentication 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.

2. The "personal data" transaction: To provide a minimum amount of user-friendliness, at each iteration of the program's main loop, you need to 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).

3. The "plan" transaction. Here, the customer types the command `plan PLAN_ID` and you set their new plan to that plan id. How does the customer know which plan id's are available? They type in `plan` without any plan id, and then you will list all available plans, their names, and their terms (maximum number of movies available for rental and monthly fees).

4. The "rent" transaction. The user types in `rent MOVIE_ID`, and you will "rent" that movie to the customer.

5. The "return" transaction. The user types in `return MOVIE_ID`. You update your records to mark the return of that movie. How does the customer know which movies they are currently renting (and thus they can return)? They type in `return` without any movie id, and the system will list for them all the movies that they are currently renting.

Comments on transactions

You must use SQL transactions in order to guarantee ACID properties: you must define begin- and end-transaction statements, and insert them in appropriate places in `Query.java`. In particular, you must ensure that the following two constraints are always satisfied, even if multiple instances of your application talk to the database.

C1. at any time a movie can be rented to at most one customer.

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

Concretely: (a) when a customer requests to rent a movie, you may need to deny this request and (b) when a customer selects a "lower" plan (with fewer allowed movies), you may also need to deny this request (why?). You can implement denying in many ways, but we strongly recommend using the SQL ROLLBACK statement.

You must use transactions correctly such that users cannot cheat, nor can race conditions introduced by concurrent execution lead to an inconsistent state of the database. For example, 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. Your properly designed transactions should prevent that.

Design transactions correctly. Avoid including user interaction inside a SQL transaction: that is, don't begin a transaction then wait for the user to decide what she wants to do (why?). The rule of thumb is that transactions need to be as short as possible.

When one uses a DBMS, by default **each statement executes in its own transaction**. To group multiple statements into a transaction, we use

```
BEGIN TRANSACTION
```

....

```
COMMIT or ROLLBACK
```

This is the same when executing transactions from Java, by default each SQL statement will be executed as its own transaction. To group multiple statements into one transaction in java, you can do the following:

Method 1:

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

[... execute updates and queries.]

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

Method 2:

```
_begin_transaction_read_write_statement.executeUpdate();

[... execute updates and queries. You may want to consider making
a different method for read-only transaction.]

_commit_transaction_statement.executeUpdate();
OR
_rollback_transaction_statement.executeUpdate();
```

When auto comit is set to true, each statement executes in its own transaction. With auto-commit set to false, you can execute many statements within a single transaction. By default, on any new connection to a DB auto-commit is set to true.

To test that your transactions work correctly, we recommend the following (and this is how we will test your homework). Place a break in the middle of your transaction, by reading and throwing away a line of the user's input. Run two (or more?) instances of VideoStore.java, say A and B. Let both reach the point when they read from the standard input; then you decide which one you allow to proceed, and thus control the order in which the transactions are interleaved.

Deliverables

You need to turn in 1 file for this assignment: Submit your Query.java that implements all the requested functionality (tasks B, C, D). Please don't modify VideoStore.java, and don't introduce additional files for your code. In testing your code, we will assume that you will be using the customer database design that you submitting in the previous deliverable. Completed version of the starter code file, that implements the requested functionality.

If you have changed your design, please also submit new **ER.pptx** and **setup.sql** files for your database.

Submit your solutions through Blackboard by the due date. Each team only needs to submit one solution.