CSE 344 Homework 7

Objectives:
To get experience with database application development and transaction management.
Assignment tools:
SQL Server through SQL Azure.
starter code files
Additional files (you normally don't need these):
SQL Server JDBC Jar files (the JDBC driver for older versions of Java),
IMDB dataset ascii version (in case you want to test your homework on your own database, before using SQL Azure)
Due date:
March 6, 2014 at 11:59 pm in the dropbox
What to turn in:
Customer database schema in `setup.sql`, completed version of the `Query.java` starter code file, and the dbconn.properties file.

Read this whole document before starting this project. There is a lot of valuable information here, including the Final Comments section at the bottom.

Congratulations, you are opening your own video rental store!

You signed a contract with a content provider that has videos of all the movies in the IMDB database, and you will resell these videos to your customers. Your store allos customers to search the IMDB movie database, then rent movies (which we assume are delivered by the content provider; we don't do this part in the project). After renting a movie, the customer may 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 what movies.

There are two important restrictions:

1. Your contract with the content provider allows you to rent each movie to at most one customer at any 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 restriction: your store is based on subscriptions (much like Netflix), allowing customers to rent up to a maximum number of movies, depending on their retnal plan. 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.

In this homework, you have two main tasks. The first is to design a database of your customers. The second 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 homework.

Task 0: Running the starter code (0 points)

Your system will be a Java application. Download the starter code files; you will 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.properties`: a file containing settings to connect to the customer and IMDB databases. You need to edit it before running the starter code.
- `sqljdbc4.jar`: the JDBC to SQL Server driver. This tells Java how to connect to a SQL Azure database server, and needs to be in your `CLASSPATH` (see below)
- `sqljdbc.jar`: the JDBC to SQL Server driver for older versions of Java. Use this driver only if the other one does not work for you.

To run the starter code, use either the Java JDK or run javac and java from the command line.

You also need to access the `IMDB` database on SQL Azure from Homework 3. Modify `dbconn.properties` with your username and password for SQL Azure. This allows your java program to connect to the database on SQL Azur..

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 Joe jopsswd
```

That is, you provide the username and password of the **video store user**. Later, your code will check the name and password of Joe-the-customer, but the starter code ignores both `user` and `password` (you can put any strings you like).

Using the command-line

Windows	Linux or Mac

`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=.;sqljdbc4.jar`
`javac -g VideoStore.java Query.java`
`java VideoStore user password`
	`cd /where/you/unzipped/the/starter/code`
	
	`export CLASSPATH=.:sqljdbc4.jar`
	`javac -g VideoStore.java Query.java`
	`java VideoStore user password`

If you got an error message about the JDBC driver when running the above, try to use the older driver *sqljdbc.jar* instead of sqljdbc4.jar.

An alternative way to run the starter code from Eclipse

So the instructions seem to emphasize running this application out of a terminal. Since my primary box is Windows, and the terminal is a pain in Windows, I set it up to run entirely in Eclipse so I didn't have to deal with it. Here are the steps you need to do:

1. Create a workspace with a project. I'll refer to this project as hw7 from now on.
2. Copy the .java files into the project src/ folder.
3. Copy the .jar and .properties files into the hw7 folder. Make sure you add the necessary values to the .properties file now.
4. Right click on the project, choose Build Path -> Configure Build Path..., then Add Jars..., then expand hw7 and select the sqljdbc[4].jar file. Click Ok and then Ok again to close the menus.
5. Click the Run button. You should get a message in the console about incorrect usage.
6. Click the drop down next to the Run button and click Run Configurations... Under Java Applications there should be one called VideoStore (It should open to this by default).
7. Click the Arguments tab. In the Program Arguments box, enter two space-delimited arguments for the username and password.
8. Click Run.

This will cause the application to run with those two arguments. It will store them so the next time you hit the run button it will use those same two arguments again. If you wish to change the arguments, you will have to open the Run Configurations... menu again and change the arguments.

Alternatively, you can open Run Configurations..., right click on VideoStore and select Duplicate. This should create one called VidoeStore (1). You can change the arguments in this one without affecting the original VideoStore (or change the name field above the tabs so you know which configuration it is). Then, when you click the arrow next to Run, it should give you an option to choose which configuration to run. This makes it easier to switch users without having to constantly change the arguments.

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.)

Task 1: Customer database design (20 points)

Your first task is to design and create your `customer` database in SQL Azure. We created a database for you on SQL Azure called *yourloginCustomer* where *yourlogin* is your login: create there all tables for your `customer` database.

Note that your Java application will connect to **two** databases: IMDB and yourloginCustomer and tehrefore your application needs to establish **two** JDBC connections. Modify the file *dbconn.properties* to indicate the name of yourloginCustomer database; use the same username and password as for IMDB.

What to turn in: a single text file called *setup.sql* with CREATE TABLE and INSERT statements for your customer database.

Customer information

Your customer information database shoudl have the following entity sets:

E1. Customer: a customer has an **id** (integer), a **login**, a **password**, a **first name** and a **last name**.

E2. Subscription. Each plan has a **Subscription id** (integer), a **name** (say: "Basic", "Rental Plus", "Super Access" -- you can invent your own), the **maximum number** of rentals allowed (e.g. "basic" allows one movie; "rental plus" allows three; "super access" allows five; again, these are your choices), and the **monthly fee**. For this project, you are asked to insert four different rental plans.

E3. 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, and the **date and time** the movie was checked out, to distinguish multiple rentals of the same movie by the same customer. 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 never delete it). Keeping the rental history helps you improve your business by doing data mining (but we don't do this in this class.)

In addition there are the following relationships:

R1. Each customer has exactly one rental subscription.

R2. Each rental refers to exactly one customer. (It also refers to a single movie, but that's in a different database, so we don't model that as a relationship.)

Create a text file called *setup.sql* with CREATE TABLE statements and INSERT statements that populate each table with a few tuples (minimum 8 tuples): you will turn in this file. This file should be runnable on SQL Azure through SQL Server Management Studio. Write a separate script file with DROP TABLE statements; it's useful to run it whenever you find a bug in your schema or data (don't turn in this file).

IMPORTANT NOTE: SQL Azure requires that you **create a clustered index on each table before you can insert values**, see this link for details and an example. You will get an error message if you try to insert into a table without a clustered index!

To create your tables and insert your data do the following:

- From SQL Server Management Studio, connect to `m01rrgdwg2.database.windows.net` using SQL Server Authentication.
- Once you are connected:

- In the Object Explorer on the left, select Databases -> yourloginCustomer
- Click on New Query (at the top)
- Copy and paste the content of *setup.sql* and press "Execute". We recommend that you execute one statement at a time and examine its effects before moving on to the next statement.

Task 2: Java customer application (80 points)

Your second 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`.

What to turn in: the Java file `Query.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 our class. Your Java application will connect to both the `IMDB` and the `CUSTOMER` databases on SQL Azure.

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; as mentioned above, one of your tasks will be to make it work.

In the welcome screen, please at least print out the customer name and the number or movies the user can still rent.

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, fixing a security flaw in it along the way.
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.

Task 2A: Completing the search function (20 points)

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

- all movies whose title matches the string, case-insensitively
- 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 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.

Task 2B: Push the join to the database engine (20 points)

Goal: 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 perfrom 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 2A; we will use the latter in 2B. In this task you will reimplement the search function from 2A 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, and by the netwrok traffic.

Write a function called `fastsearch`, by using joins (or outer joins? you need to determine that!) computed in the database engine, instead of the dependent joins computed in Java iby the `search` function. 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.)

How much better should you expect `fastsearch` to be? We tried it on both a local installation of SQL Server and on SQL Azure. In the former case, fastsearch typically increases the speed very little: perhaps from 2-3 seconds to 1 second or so; with a cold cache the performance increase may be larger, from minutes, to several seconds. On SQL Azure, the performance bottleneck is typically the data transfer from the server to the client, hence you may not see any visible performance improvement. For some queries, fastsearch may actually be slower than search! For example, we found that "search nowhere" took 1min12sec while "fastsearch nowhere" took only 34sec. In contrast, "search Nixon" took 21sec while "fastsearch Nixon" took 30sec. Your results may vary.

Task 2C: Customer database transactions (30 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. However, you must also establish a connection to your Customer database. You need to add code to do this. Look at how the starter code establishes a connection to the IMDB database.

2. The "print customer info" 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 her 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.

Be sure to use SQL transactions when appropriate to implement these "transactions". See more on this below.

Task 2D: Stop SQL Injection (10 points)

Recall the `search` for example type:

```
search Nixon
```

Now type this at the prompt:

```
search ' and year=1899 --
```

What happens? You get all movies in 1899! Now type this:

```
search '; drop movie; drop casts; drop actor; --
```

Actually, don't try it, you get the idea...

This is SQL injection: hackers like to do it on Website interfaces to databases. Your task here is to fix the search and `fastsearch` function to prevent SQL injection attacks. Fix the security issue by changing the code in Query.java (only). *Hint*: when fixing the issue, look at other parts of the starter code that execute SQL to see what they do differently from the broken search code.

Transaction management (for Task 2C)

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, but not shorter.

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 one of three things:

Approach 1:

We provide you with three helper methods. So before your first statement in the transaction, simply execute

```
beginTransaction();
```

When you are done with the transaction, then call:

```
commitTransaction();
```

OR

```
rollbackTransaction();
```

Approach 2:

You can execute the SQL code for START TRANSACTION and friends directly, using the SQL we have provided in the starter code (also check out [SQL Azure's transactions documentation)]():

```
// When you start the database up
Connection conn = [...]
conn.setAutoCommit(true); // This is the default setting, actually
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

// In each operation that is to be a multi-statement SQL transaction:
conn.setAutoCommit(false); // You MUST do this in order to tell JDBC that you are starting a multi-statement transaction

beginTransactionStatement.executeUpdate();

[... execute updates and queries.]

commitTransactionStatement.executeUpdate();
[OR]
rollbackTransactionStatement.executeUpdate();

conn.setAutoCommit(true);  // To make sure that future statements execute as their own transactions.
```

Approach 3:

```
// When you start the database up
Connection conn = [...]
conn.setAutoCommit(true); // This is the default setting, actually
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);


// In each operation that is to be a multi-statement SQL transaction:

conn.setAutoCommit(false);
[... execute updates and queries.]
conn.commit();
[OR]
conn.rollback();
conn.setAutoCommit(true);
```

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.

FINAL COMMENTS:

- The starter code is designed to give you a gentle introduction to embedding SQL into Java. Start by running the starter code, examine it and make sure you understand the part that works. You will need to create (and connect to) your new customer database before uncommenting and running the query statements. Then you should insert 1-2 customers and uncomment the few lines of code in the starter code that do the user authentication and that greet the user with her/his name and plan details. This should all work flawlessly (if not, talk to the instructor or TA). Then, complete the first (the "search") transaction, i.e. return the actors. Then continue with the other functions.

- The completed project has about 20 quite simple SQL queries embedded in the Java code. Some queries are *parameterized*: a parameter is a constant that is known only at runtime, and therefore appears as a '?' in the SQL code in Java: you already have examples in the started code.

- We won't use recovery in this homework. Instead you will rely on the script file that you need to prepare for Task 1. This file contains all the CREATE TABLE and INSERT statements that are needed to start your project.