CIS560 – Database Systems Concepts **Name:______________**

SQL Assignment 5 [40 points] – due October 18th at 11:59PM

In this assignment, we will be working with the **movie** database that we have used before, and a new **customer** database that you will create as part of this assignment.

Note 1: In principle, the **movie** and **customer** databases could be of different types (e.g., MySQL and PostgreSQL) and on different servers, and you would need to connect to each of them with the right driver. However, to keep things simple, we will assume that both **movie** and **customer** databases are MySQL databases, hosted on the same server (specifically, the CIS server). [They will actually be part of your CIS username database.]

Note 2: You will need to be within the KSU domain to be able to connect to your MySQL database on the CIS server. If you want to work from home, you will have to install the KSU VPN Client: http://www.k-state.edu/its/security/vpn/.

Note 3: This is an individual assignment. You are allowed to discuss the assignment with your colleagues, but your submission should reflect your own work. Sharing or copying code is not permitted. We reserve the right to divide the grade between any students who violate this policy. In addition, you must identify all those that you collaborate with on your assignment cover sheet. Consult your instructor if you need further clarification.

Note 4: Upload the file into the Dropbox on KSOL.

You will practice JDBC, Updates and Transactions.

Remember that the **movie** database consists of 6 tables:

```
movie_info (movie_id, movie_name, year, rating)
actor_ids (actor_id, actor_name, gender)
actor_movies (actor_id, movie_id)
producer_ids (producer_id, producer_name)
producer_movies (producer_id, movie_id)
genre(movie_id, genre)

actor_movies.actor_id refers to actor_ids.actor_id
actor_movies.movie_id refers to movie_info.movie_id

producer_movies.producer_id refers to producer_ids.producer_id
producer_movies.movie_id refers to movie_info.movie_id

genre.movie_id refers to movie_info.movie_id
```

JDBC drivers: In order to connect to a database from Java, you need a database-specific JDBC driver. As we work with MySQL databases, you will need to download the MySql JDBC connector, e.g. `mysql-connector-java-5.1.18-bin.jar` (other versions might also work) and add it to your `classpath`. You can download this jar file from http://dev.mysql.com/downloads/connector/j/.

If you use Eclipse to develop your program (which we strongly recommend) and don't have it on your computer already, you can download it from this link http://www.eclipse.org/downloads/ (Eclipse IDE for Java Developers). To be able to use Eclipse, you will also need a Java JDK. You can download it from the following link, if you have not needed it before: http://www.oracle.com/technetwork/java/javase/downloads/index.html.

If you use Eclipse, you can use `Configure Build Path`… to `Add External Jar` to the `Libraries`, specifically you will need to add the mysql-connector-java jar. Read more about how you can do this at: http://www.wikihow.com/Add-JARs-to-Project-Build-Paths-in-Eclipse-(Java)

Starter code: code.zip – available on KSOL - download and unzip into your local directory. You will only need to modify `Query.java`.

Running the starter code: To run the starter code make sure you have set the classpath as described above. Then, change **dbconn.config** to edit the username and password for MySql. Then:

```
javac SQLassign.java
javac Query.java
java SQLassign uid passwd
```

(The starter code does not authenticate the user, so uid and passwd are ignored: one of your jobs is to modify the starter code to do the authentication.)

At this point you should see this:

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

The command 'search' is partially implemented. Try typing:

```
search agent
```

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

If you use Eclipse, this is how you could send command line arguments in Eclipse:

http://www.javaprogrammingforums.com/java-jdk-ide-tutorials/362-how-send-command-line-arguments-eclipse.html

In Eclipse, your dbconn.config file should be in the main project folder (not in the src folder).

Project description

The goal of this assignment 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 (simply called **movie** 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 movie database (the starter code already does that), 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, he/she can watch it as many times as they want, until they decide to "return" it to your store. You need to keep track of which customers are currently renting which 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 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 subscription (much like Netflix), which allows customers to rent up to a maximum number of movies. 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.

Task 1

Design and create the **customer** database (as part of your CIS database). The database has the following entity sets:

E1. customer: a customer has a **cid** (integer), a **login**, a **password**, a **first name** and a **last name** and a **rental plan**.

E2. plan: each plan has a **plan 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 assignment, 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 **cid**. The movie is identified by a **movie id** (from the **movie** database). The rental has a **status** that can be *open*, or *closed*. When a customer 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). We record the rental history using an attribute called **time,** which is meant to store the time when a movie was rented. To keep things simple, we will represent time as an integer, specifically a counter that shows the number of times a particular movie was rented by the same customer. For example, we could have the following entries in the customer table for a customer cid_5, movie M_166 combination:

(cid_5, M_166, closed, 1)
(cid_5, M_166, closed, 2)
(cid_5, M_166, open, 3)

When the customer cid_5 returns the movie M_166, the last tuple will be updated to

(cid_5, M_166, closed, 3)

If the customer rents again that movie, we will have a new entry in the table

(cid_5, M_166, open, 4)

Keeping the rental "history" could help you improve your business by doing data mining (although we don't do that in this assignment).

In addition, there are the following relationships:

R1. Each customer has exactly one rental plan (which means we don't need a separate relation for cid,plan_id pairs).

R2. Each rental refers to exactly one customer. (It also refers to a single movie, but that's in a different database, i.e., in the **movie** 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 (say 2-8 tuples): you will turn in this file. This file should load into MySQL. 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.

What to turn in (part 1) [10 points]: a single text file called "setup.sql" with CREATE TABLE and INSERT statements for this database. Test this file by loading it into your MySQL **customer** database. Your Java program should then be ready to run.

Task 2

Write the Java application, by completing the starter code. Remember, you only need to modify the Query.java class.

The application is a simple command-line java program. A "real" application will have a real “Web interface” instead, but we will not worry about that in this assignment. Your Java application needs to connect to your CIS database, which contains the **movie** and the **customer** databases.

When your application starts, it reads a `customer name` and `password` from the command line. It validates them against the database and 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.

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.)

1. The "search" transaction [5 points]: the user types in a string, and you return:

- all movies whose title matches the string
- their producer(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 producers: you still need to retrieve and print the actors and the availability status. In the starter code, the producers are obtained using a "dependent join" (sometimes called "nested loop join"). You should use the same technique for retrieving the actors: there is a more efficient way, but we'll implement that in "fastsearch".

What you will learn in 1:

a. how to run SQL queries from Java
b. what a dependent join is, and how easy it is to embed it in an application
c. how slow the dependent joins are

2. The "plan" transaction [5 points]: Here, the customer types in a plan id and you set his/her new plan to this plan id. How do the customers know what plan ids 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).

What you will learn in 2: simple database updates from Java.

3. The "rent" transaction [10 points]: The user types in a movie id, and you will "rent" that movie to the customer.

4. The "return" transaction [10 points]: The user types in the movie id to be returned. You update your records to mark the return.

What you will learn in 2, 3, 4: the typical interaction between the application and the database; basic data integration (you integrate **customer** with **movie** databases); the need for transactions and rollback.

5. The "fastsearch" function [10 points]. Here, you will fix the performance problem in the "search" function: "fastsearch" has exactly the same functionality as "search" (except that it doesn't have to return the availability status). However, in “fastsearch” all the dependent joins are replaced with regular joins, executed in the database engine.

Hint: Add ORDER BY to each of the three queries so you can merge-join the three streams efficiently.

Note 5.1: This task requires a little bit more creativity (and more work) than the others, but shouldn't be too difficult!

Note 5.2: Depending on your setting, "fastsearch" may actually run slower than regular "search": if that happens for the right reason, then you are OK. In "fastsearch", you are running upfront three SQL queries, which are more complex than the single SQL query executed in "search": this initial step will take longer. Subsequently however, "fastsearch" no longer issues queries, and iterates over all movies, actors, and producers exactly once, while search needs to issue two new SQL queries for each movie (not counting the computation of the availability status). Thus, the time it takes to the *first answer* may be longer in "fastsearch", but subsequent answers should be returned "fast". If you don’t actually see a difference between “search” and “fastsearch” could be because the databases you are working with are not too large.

What you will learn in 5: the database system is much better at performing the join than the java application.

In addition to implementing the *transactions* above, 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).

What you will learn: give the user as much data as they need, but not more.

What to turn in (part 2): the java file Query.java.

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. Then create your **customer** database, insert 1-2 customers, and uncomment the few lines of code in the starter code that do the user authentication. Also, write the `transaction_personal_data` function, which is used to greet the user with her/his name and plan details. This should all work flawlessly (if not, talk to me or to the GTA, Rohit Parimi). Then, complete the first (the "search") transaction, i.e. return the actors. Then continue with the other function.
- 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: we have talked about this in class and you already have examples in the started code.
- You need to enforce the following constraints:

  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.

  What this means is that

  a. when a customer requests to rent a movie you may deny this request and
  b. when a customer selects a "lower" plan (with fewer allowed movies) you may deny this request (why?). Thus, you need to worry about C in ACID.

  Hint: Transactions and ROLLBACK are your friends.
- 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.

  Hint: Transactions are your friends again.
- As a student concerned about your grade in CIS560, 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. You won't use recovery, however. 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.
- Never include a user interaction inside a transaction! That is, don't begin a transaction then wait for the user to decide what he/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 **movie**, since this database is never updated.

Hint about transactions in Java: There are two ways to execute multi-statement transactions from Java. Feel free to use the method that you prefer.

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.]

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

Acknowledgments: The assignment has been adapted from a similar assignment at University of Washington.