University of Toronto
CSC343, Winter 2015

Assignment 2

Due: Tuesday, March 10, at 8:00 pm sharp!

General Instructions

1. Please read this assignment thoroughly before you proceed. Failure to follow instructions can affect your grade.
2. We strongly encourage you to do all your development for this assignment on CDF, either in the labs or via a remote connection.
3. Download the starter files from the course webpage:
   - The database schema, `imdb.ddl`
   - The starter code for the Java portion of the assignment, `Assignment2.java`.

   If you are using a graphical user interface to connect to CDF, you can simply download the files and save them into a directory of your choice. You can also use the command line to download the files. For example:

   ```
   > wget http://www.cdf.toronto.edu/~csc343h/winter/assignments/a2/imdb.ddl
   ```

You are allowed, and in fact encouraged, to work with a partner for this assignment. You must declare your team (whether it is a team of one or of two students) and hand in your work electronically using MarkUs.

Once you have submitted your files, be sure to check that you have submitted the correct version; new or missing files will not be accepted after the due date, unless your group has grace tokens remaining.

Schema

In this assignment, we will work with a movie database that is based on the information stored on IMDb.[1] What follows is a brief description of the meaning of the attributes in the schema; to fully understand this database, read these descriptions in conjunction with the actual schema definition found in `imdb.ddl`.

- **movies**(movie_id, title, year, rating)
  A tuple in this relation represents a movie with a *title*, the *year* it was released, and its *rating* (a float between 0 and 10).
- **people**(person_id, name)
  A tuple in this relation represents a person with a given *name*. Names are stored in a single text attribute in the format 'surname, firstname'; for example, the actor 'Brad Pitt' would be represented as 'Pitt, Brad'.
- **cinematographers**(movie_id, person_id)
  A tuple in this relation represents that the person with id *person_id* worked on the movie with id *movie_id* as a cinematographer.
- **composers**, **directors**, and **writers** link a person with a movie in the same way as **cinematographers**.
- **roles**(movie_id, person_id, role)
  A tuple in this relation represents that person *person_id* acted in movie *movie_id* and played a character named *role*.

[1] Information courtesy of IMDb (http://www.imdb.com). Used with permission.

- **genres**(genre_id, label)
  A tuple in this relation represents a genre called *label*. An example of a genre label is 'Action.'
- **movie_genres**(movie_id, genre_id)
  A tuple in this relation represents that movie *movie_id* belongs to genre *genre_id*.
- **keywords**(keyword_id, keyword)
  A tuple in this relation represents a keyword called *keyword*. An example of a keyword is 'spear-through-chest.'
- **movie_keywords**(movie_id, keyword_id)
  A tuple in this relation represents that movie *movie_id* is associated with keyword *keyword*.

The schema contains a few other tables which we will not use for this assignment. **All of your work in this assignment should only affect the tables listed above, and no other tables.**

The schema definition uses four types of integrity constraints:

- Every table has a primary key ("`PRIMARY KEY`").
- Some tables use foreign key constraints ("`REFERENCES`").
- Some tables define other keys ("`UNIQUE`").
- Some attributes must be present ("`NOT NULL`").

Your work on this assignment must work on *any* database instance (including ones with empty tables) that satisfies these integrity constraints, so make sure you read and understand them.

Warmup: Getting to know the schema

To get familiar with the schema, ask yourself questions like these (but don't hand in your answers):

- Can two different people have the same name in the **people** table?
- Can there be two different movies with the same title released in different years? In the same year?
- Can a person work on a movie as both a writer and a director?
- Is there a lower or upper limit on the number of keywords associated with a movie?

Part 1: SQL Statements

In this section, you will write SQL statements to perform queries and updates to an IMDb database. There are **seven** questions. Write your SQL statement(s) for each question in separate files named `q1.sql`, `q2.sql`, ..., `q7.sql`, and submit each file on MarkUs. You are encouraged to use views to make your queries more readable. However, each file should be entirely self-contained, and not depend on any other files; each will be run separately on a fresh database instance, and so (for example) any views you create in `q1.sql` will not be accessible in `q5.sql`. **Each of your files must begin with the line** `SET search_path TO imdb;` Failure to do so will cause your query to raise an error, leading you to get a 0 for that question.

For questions which ask you to make a query ("Report ..."), your output must exactly match the specifications in the question: attribute names and order, and the order of the tuples.

We will be testing your code in the **CDF environment** using PostgreSQL. It is your responsibility to make sure your code runs in this environment before the deadline! **Code which works on your machine but not on CDF will not receive credit**.

---

1. **Pittsters.** Report every person who has worked as a writer or composer on some movie where Brad Pitt (represented in the database as 'Pitt, Brad') has worked as an actor. Your query should return an empty table if 'Pitt, Brad' is not found.

Attribute	
name	Name of person
bradtimes	Number of *different* movies the person worked with Brad Pitt as writer or composer
Order by	name ascending
Duplicates?	No duplicates

2. **Diversity works?** In the schema, every movie can be associated with some keywords. Use a query to gather evidence about whether there is a correlation between the number of distinct keywords associated with a movie and the rating of that movie.

Attribute	
keywords	Number of keywords
avgrating	Average rating of a movie with exactly that number of keywords
Order by	keywords ascending
Duplicates?	No duplicates

   E.g., a tuple (3, 7.6) in your output table means that the average rating of a movie with exactly three distinct keywords is 7.6.

3. **Job makers.** A **position** in a movie is a job as an actor, cinematograper, composer, director, or writer for that movie. A person is considered to have worked on a movie if he/she is associated with that movie in at least one position. Report the number of positions and distinct people who worked in each movie. If a person worked on the same movie under two different positions, they should be counted **once for each different position** in the *positions* column, but only once in total in the *people* column. (So someone who worked as both a writer and director for a movie would be counted twice in positions, but one in people.) However, an actor who played two or more different roles in one movie should only be counted once in both the positions and people columns.

Attribute	
movie_id	id of the movie
positions	Total number of positions worked on for the movie
people	Number of distinct people who work on the movie
Order by	positions descending, then people descending, then movie_id ascending
Duplicates?	No duplicates

4. **Super writers.** A *decade* is a group of ten years starting with a year divisible by 10: e.g., 1950-1959, 1990-1999, 2010-2019. A movie is *super* if its rating is greater than or equal to the rating of every other movie released in the same decade.

   Report the people who worked as a writer on at least one movie, and all of whose movies were super movies. Your output should have one tuple per writer and super movie (so if "Pitt, Bob" wrote three movies, and they were all super, then each movie would appear in a separate tuple with "Pitt, Bob" in the output). You should ignore other positions a person may have had; if "Pitt, Bob" also directed a movie that was not super, he could still be included in the output.

   Your value for the *decade* attribute **must be a string of the form "YYYYs", e.g., "1950s", "1970s", "2000s".** You may assume all years are at most four digits long, and use the SQL function `CAST` (you'll need to look up what it does).

Attribute	
writer	name of the person
supermovie	title of the super movie
rating	rating of the movie
decade	the movie's decade
Order by	writer ascending, then decade ascending, then title ascending
Duplicates?	No duplicates

5. **Busy bees.** Find the actors who have been in at least one movie per year for three consecutive years. If this has happened multiple times for the same actor (e.g., they acted in movies in 2003-2005 and 2010-2015), choose the *latest* three years (2013-2015). If the actor acted in more than one movie in a year, report the one whose title comes first in alphabetical order.

Attribute	
person_id	id of actor
name	name of the actor
year1	first of the three consecutive years
movie1	title of movie in year1
year2	second of the three consecutive years
movie2	title of movie in year2
year3	first of the three consecutive years
movie3	title of movie in year3
Order by	person_id asc
Duplicates?	No duplicates

6. **Part 2 Coming Soon...** Everyone knows that Hollywood has run out of original ideas. Assume that in the year 2020, a sequel is released for every movie released before 1990 (not including movies released in 1990). The sequel movie has the following properties:

   - The sequel's title is equal to the original movie's title concatenated with ': The Sequel'. For example, the sequel for 'Schindler"s List' would be called 'Schindler"s List: The Sequel'. Capitalization and spaces matter! (Also note that SQL strings use *double-quotes* to represent apostrophes.)
   - All other attributes of the movie, other than the year of release, should be the same.
   - Because robots will be our overlords in 2020, none of these sequel movies have any humans associated with them (actors, directors, writers, etc.).

   Insert the new sequels into the **movies** table according to the above condition; no other tables should be updated. The new sequels should have movie_ids in consecutive order, immediately following the largest movie_id in the existing table. The sequels should be added in order of the movie_id of the corresponding original movies. So if 'Schindler"s List' had movie_id 3, and 'The Godfather' had movie_id 10, then 'Schindler"s List: The Sequel' should have a smaller movie_id than 'The Godfather: The Sequel'.

   You may assume for this question, and only this question, that the movie_ids in the database instance form a consecutive sequence of numbers starting at 1. However, your commands should also work properly if there are no movies in the database.

7. **He was dead all along.** Our robot overlords have decided that M. Night Shyamalan's movies are so bad that they must be purged from the database. In fact, they decreed that the movies are so bad that every person associated with any one of his movies (as an actor, writer, etc.) must *also* be removed from the database. Perform SQL commands to do the following:

   - Remove all movies that M. Night Shyamalan worked on (in any capacity).
   - Remove all people associated with any movie that M. Night Shyamalan worked on.

   You must remove the people and movies from *all* the tables that might reference them, not just **movies** and **people**.

   You must use the string 'Shyamalan, M. Night' to search for this person in the **people** table. Your commands should do nothing if this name is not found in the database.

Embedded SQL

You might have heard of the Six Degrees of Kevin Bacon game: pick an actor, and determine the shortest path between this actor and Kevin Bacon. We define a **path** between two actors P1 and P2 as a sequence of actors [P1, Q1, Q2, ..., Qn, P2] such that P1 and Q1 worked on the same movie, Q1 and Q2 worked on the same movie, ..., Qn and P2 worked on the same movie. Note that for this game, we ignore the other non-actor people who worked on the movies – they cannot be part of the path. We say that the **connectivity** of two people P1 and P2 is the length of the shortest path between them, *not* including P2. So if P1 and P2 worked on the same movie, there is a path [P1, P2] connecting them, and the connectivity of P1 and P2 is 1.

Accomplishing this computation using pure SQL queries is rather difficult, because we lack the ability to use some kind of iteration to try paths of indeterminate length. That is, while we can write a query for "paths of length 3", we can't search for "shortest path" without appealing to rather advanced DBMS techniques. In this part, you will instead implement this Six Degrees game in Java, using JDBC to connect to a database defined by the IMDb schema.

Before you begin, read these general instructions:

1. You may not use standard input or output. Doing so even once will result in the autotester terminating, causing you to receive a **zero** for this part.
2. The database, username, and password must be passed as parameters, never "hard-coded." We will use the `connectDb()` and `disconnectDB()` methods to connect to the database with our own credentials. Our test code will use these to connect to a database. You should **not** call these in the other methods we ask you to implement; you can assume that they will be called before and after, respectively, any other method calls.
3. All of your code must be written in `Assignment2.java`. This is the only file you may submit for this part.
4. You may not change the method signature of any of the core four functions we've asked you to implement. However, you are encouraged to write helper functions to maintain good code quality.
5. As you saw in lecture, to run your code, you will need to include the JDBC driver in your class path. You may wish to review the related JDBC Exercise posted on the course website.

Open the starter code in `Assignment2.java`, and complete the following four functions. You must not change the names and signatures of any of the core four functions.

1. `connectDB`: Connect to a database with the supplied credentials.
2. `disconnectDB`: Disconnect from the database.
3. `findCoStars`: Return a list of actors who worked with a given actor on some movie.
4. `computeConnectivity`: Compute the connectivity between two actors in the database.

Note that neither the SQL queries nor the actual Java code needs to be very complex for this assignment; we really just want to give you an opportunity to put these two pieces together in a single program.

The main built-in Java class you'll be working with is `ArrayList`, for which you can find documentation here: `http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`. You can find some additional material in section 3.2 of here: `http://www.vogella.com/tutorials/JavaCollections/article.html#javacollections_lists`.

We don't want you to spend a lot of time learning Java for this assignment, so feel free to ask lots of Java-specific questions as they come up. Also keep in mind that to accomplish the assignment, you shouldn't need more than a handful of the basic `ArrayList` methods, plus a `for` loop.