CSE 344 Homework 2: SQL (basic)

Objectives:
To create and import databases, to practice simple SQL queries, and to create and use database indexes.
Assignment tools:
`sqlite3`, IMDB dataset (.tar.gz archive)
(Reminder: To extract the content of a tar file, try the following command: `tar zxvf imdb2010.tar.gz`)
Due date:
Monday, January 27, 2014, at 11:59 pm Turn in your answers here.
What to turn in:
`create-tables.sql`, `create-indexes.sql`, and `hw2-queries.sql` (see below)

*NOTE: after you create and import the table CASTS as described below, run the query select count(*) from CASTS: the answer should be 11445843. If you get **significantly** fewer items, then replace casts.txt with this file: http://www.cs.washington.edu/education/courses/cse344/casts.txt.gz The reason is that some verions of sqlite (on OS X) read quotations marks incorrectly and drop about 1/2 of the records in casts (we have seen cases when only 5634257 records were read).*

In this homework, you will write several SQL queries on a relational movie database. The data in this database is from the IMDB website. The database consists of six tables:

ACTOR (id, fname, lname, gender)
MOVIE (id, name, year)
DIRECTORS (id, fname, lname)
CASTS (pid, mid, role)
MOVIE_DIRECTORS (did, mid)
GENRE (mid, genre)

All `id` fields are integers. Movie.year is also an integer. All other fields are character strings. You can use either `text` or `varchar(xx)` for the character strings (people's names have maximum 30 characters, roles and genres have maximum 50, and movie titles have maximum 150). You should define the `gender` field to be one single character long.

In addition, impose the following constraints:
ACTOR.id**, MOVIE.**id**, DIRECTORS.**id = primary keys of the corresponding tables
CASTS.pid = foreign key to **ACTOR**.id
CASTS.mid, **MOVIE_DIRECTORS**.mid= foreign keys to **MOVIE**.id
MOVIE_DIRECTORS.did = foreign key to **DIRECTORS**.id
Interestingly, the IMDB dataset is not perfectly clean and some entries in **GENRE**.mid refer to non-existing movies. While we could clean-up the dataset, we chose to leave it unchanged to point-out that dirty data is a huge problem in practice.
In this assignment, simply do not specify that GENRE.mid is a foreign key. If you do, the data will fail to load.

We provide the movie database as a set of plain-text data files in the linked .tar.gz archive. Each file in this archive contains all the rows for the named table, one row per line.

In this homework, you need to do three things: (A) import the movie dataset into SQLite, (B) create indexes on the data to make the queries run fast, and (C) run SQL queries to answer a set of questions about the data.

A. IMPORTING THE IMDB DATABASE (10 points):

To import the movie database into SQLite, you will need to run `sqlite3` with a new database file: for example "**sqlite3 myhw2**". Then you can use the `CREATE TABLE` SQL statement to create the tables, choosing appropriate types for each column and specifying all key constraints as described above:

```
CREATE TABLE table_name (  . . . );
```

Currently, SQLite does not enforce foreign keys by default. To enable foreign keys use the following command. The command will have no effect if your version of SQLite was not compiled with foreign keys enabled. Do not worry about it.

```
PRAGMA foreign_keys=ON;
```

Then, you can use the SQLite `.import` command to read data from each text file into its table:

```
.import filename tablename
```

See examples of `.import` statements in the lecture notes, and also look at the SQLite documentation or `sqlite3`'s `.help` for details.

Put all the code for part A (six `create table` statements and six `.import` statements) into a file called `create-tables.sql`.

B. CHOOSING INDEXES (40 points):

NOTE: you may want to start working on C first, then return to B.

Once you have imported the movie database, your main task is to write SQL queries (task C below). However, you will find that even simple queries will take a long time; this is because the movie database you created lacks indexes on frequently accessed columns of its tables.

Hence, you will need to choose indexes for the movie database and create them using SQLite's `CREATE INDEX` SQL statement:

```
CREATE [UNIQUE] INDEX index_name ON table_name(col_1, col_2, ...);
```

If given, the optional keyword `UNIQUE` says that no row can exist in the table with duplicate values of all the index columns. If you did not mark those constraints already in the `CREATE TABLE` statements, telling SQLite about them here may let it choose a faster way to create and maintain the index, or to read through the indexed table when running a query.

Your goal is to choose up to about 10 indexes, such that once the indexes are created, your queries for the questions below should not take more than 1 minute on a modern computer. You have some flexibility about which indexes to choose, there is no absolutely perfect solution; our solution takes only about 15 seconds to run on a desktop machine from 2009, using 11 indexes. Be sure to create only indexes that matter.

Before each CREATE INDEX statement write a one-line comment with the reason for choosing each index (e.g. "in order to speed up the selection in query 7" or "in order to speed up the join in query 11").

Note that index creation can be a time consuming operation.

Put all your code for part B (`CREATE INDEX` code) in a file called `create-indexes.sql`.

C. SQL QUERIES (50 points; 10 points per question):

HINT: You should be able to answer all the questions below with SQL queries that do NOT contain any subqueries!

For each question below, write a single SQL query to answer that question. Put your queries in a file called `hw2-queries.sql`. Add a comment to each query indicating with the question it answers and the number of rows in the

query result.

1. List the first and last names of all the actors who played in the movie 'Officer 444'. *[~13 rows expected]*

2. List all the directors who directed a *'Film-Noir'* movie in a leap year. (You need to check that the genre is *'Film-Noir'* and simply assume that every year divisible by 4 is a leap year.) Your query should return director name, the movie name, and the year. *[~113 rows]*

3. List all the actors who acted in a film before 1900 and also in a film after 2000. (That is: < 1900 and > 2000.) *[~53 rows]*

   How can this be? Actors can't live more than 100 years, right? Please find the explanation. For that you need to investigate a bit, perhaps run 1-2 additional queries (include them in your `hw2-queries.sql` file). Once you identify one logical explanation why some actors appear in movies more than 100 years apart, write it in your SQL code, as a comment to the SQL query; keep your answer below 1-2 sentences.

4. List all directors who directed 500 movies or more, in descending order of the number of movies they directed. Return the directors' names and the number of movies each of them directed. *[~47 rows]*

5. We want to find actors that played five or more roles in the same movie during the year 2010. Notice that **CASTS** may have occasional duplicates, but we are not interested in these: we want actors that had five or more *distinct* roles in the same movie in the year 2010. Write a query that returns the actors' names, the movie name, and the number of distinct roles that they played in that movie (which will be ≥ 5). *[~24 rows]*

Put all your code for part C (`SELECT-FROM-WHERE` code) in a file called `hw2-queries.sql`.

NOTES:

- Unless otherwise specified, in each question that asks you to return all actors, or all directors, or all movies, you should return their names (first and last for people).
- Each question lists a *suggested* number of rows returned for a correct query. If your query produces a slightly different number of rows, that does not necessarily mean that you made a mistake.
- Save all files you create during this homework: you will need them in later homeworks.