15-415: Database Applications

Project 1: Querying the MovieLens Database

School of Computer Science
Carnegie Mellon University, Qatar
Spring 2015

Assigned date: February 03, 2015
Due date: February 17, 2015 by 11:59PM

I. Project Objectives:

The objectives of this project are to help students in: (a) practicing and applying the constructs of SQL, (b) querying a real dataset such as MovieLens, and (c) appreciating the power of SQL in extracting and analyzing useful information from real datasets.

II. Loading the MovieLens Database:

The MovieLens dataset is composed of information about 10,681 movies and their actors, directors, ratings and tags, all gathered from the online movie recommender service MovieLens. For this project, we will query the MovieLens dataset to extract useful information about movies.

The project archive (posted on the course web-page) contains five files, namely *movies.dat, actors.dat, genres.dat, tags.dat,* and *tag_names.dat*, obtained from the MovieLens Dataset. The files have been preprocessed and are ready to import into your project's database. Create the following five relations under your database and import the data from each file into the corresponding relation: [3 points]

movies(mid: integer, title: varchar, year: date, rating: real, num_ratings: integer)

actors(mid: integer, name: varchar, cast_position: integer)

genres(mid: integer, genre: varchar)

tags(mid: integer, tid: integer)

tag_names(tid: integer, tag: varchar)

In the movie relation, each movie has a unique *mid*, *title*, *year* of release, an overall user *rating* between 0 and 5.0 computed as the average of *num_ratings* ratings. Additional information about a movie is recorded in the remaining relations and is self-explanatory. Note that *cast_position* is the position of an actor in a movie's cast list. For example, in the movie ´Twilight´, the cast positions of Kristen Stewart and Robert Pattinson are 1 and 2 respectively.

III. Querying the MovieLens Database:

For each of the following questions, write and execute an SQL query that achieves the required task using PostgreSQL. You may define and use VIEWS whenever you find them suitable.

1. Print all movie titles starring '*Daniel Craig*', sorted in an ascending alphabetical order. [2 points]

2. Print names of the cast of the movie '*The Dark Knight*' in an ascending alphabetical order. [2 points]

3. Print the distinct genres in the database and their corresponding number of movies N where N is greater than 1000, sorted in the ascending order of N. [2 points]

4. For each year, print the movie title, year, and rating, sorted in the ascending order of year and the descending order of movie rating. [2 points]

5. Critiques say that some words used in tags to convey emotions are very recurrent. To convey positive and negative emotions, the words 'good' and 'bad', respectively, are used predominantly in tags. Print all movie titles whose audience opinion is *split* (i.e., has at least one audience who expresses positive emotion and at least one who expresses negative emotion). [4 points]

6. One would expect that the movie with the highest number of user ratings is either the highest rated movie or perhaps the lowest rated movie. Let's find out if this is the case here. [8 points]

    6.1 Print all information (mid, title, year, num ratings, rating) for the movie(s) with the *highest* number of ratings. [1 point]

    6.2 Print all information (mid, title, year, num ratings, rating) for the movie(s) with the *highest* rating (include tuples that tie), sorted by the ascending order of movie id. [1 point]

    6.3 Is (Are) the movie(s) with the most number of user ratings among these *highest* rated movies? Print the output of the query that will check our conjecture (i.e., your query will print the movie(s) that has (have) the highest number of ratings as well as the highest rating). [2 points]

    6.4 Print all information (mid, title, year, num ratings, rating) for the movie(s) with the *lowest* rating (include tuples that tie), sorted by the ascending order of movie id. [1 point]

    6.5 Is (Are) the movie(s) with the most number of user ratings among these *lowest* rated movies? Print the output of the query that will check our conjecture (i.e., your query will print the movie(s) that has (have) the highest number of ratings as well as the lowest rating). [2 points]

6.6 In conclusion, is our hypothesis or conjecture true for the MovieLens database? [1 point]

7. Print the movie title, year, and rating of the *lowest* and *highest* movies for each year in 2005 – 2011, inclusive, in the ascending order of year. In case of a tie, print the records in the ascending order of title. [10 points]

   For your reference, a sample output for the years 2003 – 2005 is shown below:

year	title	rating
2003	House of the Dead	3.8
2003	Oldeuboi	4.6
2004	Catwoman	1.4
2004	Bin-jip	4.4
2005	Alone in the Dark	2.2
2005	Chinjeolhan	4.7
2005	Star Wars	4.7

8. Let us find out who are the "no flop" actors. A 'no flop' actor can be defines as one who has played only in movies which have a rating greater than or equal to 4. We split this problem into the following steps. [12 points]

   8.1 Create a view called *high ratings* which contains the distinct names of all actors who have played in movies with a rating greater than or equal to 4. Similarly, create a view called *low ratings* which contains the distinct names of all actors who have played in movies with a rating less than 4. Print the number of rows in each view. [3 points]

   8.2 Use the above views to print the number of 'no flop' actors in the database. [2 points]

   8.3 For each 'no flop' actor, print the name of the actor and the number of movies N that he/she played in, sorted in descending order of N. Finally, print the top 10 only. [7 points]

9. Let us find out who is the actor with the highest 'longevity'. Print the name of the actor/ac- tress who has been playing in movies for the longest period of time (i.e., the time interval between their first movie and their last movie is the greatest). [15 points]

10. Let us find the close buddies of Annette Nicole. Print the names of all actors who have starred in (at least) all movies in which Annette Nicole has starred in. Note it is ok

if these actors have starred in more movies than Annette Nicole has played in. Since, PostgreSQL does not provide a relational division operator, we will guide you through the following steps (you might find it useful to consult the slides or the textbook for the alternative "double negation" method of performing relational division). [15 points]

10.1 First, create a view called *co_actors*, which returns the distinct names of actors who played in at least one movie with Annette Nicole. Print the number of rows in this view. [3 points]

10.2 Second, create a view called *all_combinations* which returns all possible combinations of *co_actors* and the movie ids in which Annette Nicole played. Print the number of rows in this view. *Note how that this view contains fake (co_actor, mid) combinations!* [4 points]

10.3 Third, create a view called *non_existent* from the view *all_combinations* by removing all legitimate (co_actor,mid) pairs (i.e., pairs that exist in the *actors* table). Print the number of rows in this view. [4 points]

10.4 Finally, from the view *co_actors*, eliminate the distinct actors that appear in the view *non_extistent*. Print the names of all co_actors except Annette Nicole. [4 points]

11. Let us find out who is the most *social* actor. A *social* actor is the one with the highest number of distinct co-actors. We will break this into two sub-tasks: [15 points]

    11.1 For the actor Tom Cruise, print his name and the number of distinct co-actors. [5 points]

    11.2 For each actor, compute the number of distinct co-actors. For the highest such number, print the name of the actor and the number of distinct co-actors. In case of a tie, print the records sorted in alphabetical order by name. [10 points]

12. We will now write some queries for a *Content-Based Movie Recommendation System* such as NetFlix. In reality, the accuracy of the recommendations is so important that NetFlix, for instance, offered a prize of one million dollars for the first algorithm that could beat its own recommendation algorithm by 10%. The prize was finally won in 2009, by a team of researchers called "Bellkor's Pragmatic Chaos". However, in this project we shall deploy a *simple* algorithm that may or may not produce optimal recommendations.

    Content-based recommendations focus on the properties of items, in our case movies. The similarity of two movies is determined by measuring the similarity of their properties. For a movie item, we shall consider the following five properties: actors, tags, genres, year, and rating.

    Given two movies X and Y, the similarity of Y to X, *sim(X,Y)*, can be computed as:

$$\frac{\text{fraction of } common\ actors\ +\ \text{fraction of } common\ tags\ +\ \text{fraction of } common\ genres\ +\ age\ gap + rating\ gap}{}$$

where *fraction* is the number of common elements between X and Y divided by the number of elements *of X, age gap* is the normalized difference between the production years of X and Y, and *rating gap* is the normalized difference between the ratings of X and Y. Intuitively, the smaller the gaps are, the better (since movies of the same decade and rating are more likely to be similar). Moreover, note that we divide by five because each property is given an equal weight of 1.

Given a user who is known to like the movie *'Mr. & Mrs. Smith'*, write a query that prints the movie title, rating, and similarity percentage (i.e., similarity * 100) for the top 10 movies that are most similar to the *'Mr. & Mrs. Smith'* movie, ordered by the similarity percentage. [10 points]

IV. Getting Help:

You can get help by visiting the professor and the TA during their office hours or by appointment. You can also post your questions on Piazza. The link for the course home page on Piazza is: http://piazza.com/qatar.cmu/spring2015/15415/home

V. Deliverables:

- Write all your SQL queries in a PDF document named ***P1-<your_andrew_id>.pdf***
- For each question and/or sub-question, create two files namely ***Q<question_#>-[<sub-question_#>]-<your_andrew_id>.txt*** *and* ***Q<question_#>-[<sub-question_#>]-<your_andrew_id>.csv*** containing the query and its output result respectively.
- Zip all your files into a single archive file and submit it to **/afs/qatar.cmu.edu/usr10/mhhammou/www/15415-s15/handin/p1/andrew_id/** by February 17, 11:59 PM. In case of any problems, you can email your project archive to the professor and the TA.

VI. Late Policy:

- If you hand in on time, there is no penalty (duh!).
- 0 -24 hours late = 25% penalty.
- 2 4 -48 hours late = 50% penalty.
- More than 48 hours late = you lose all the points for this project.

NOTE: You can use your grace-days quota. For details about the quota, please refer to the syllabus.