Database Systems, CSCI 4380-01
Homework # 5
Due Sunday March 27, 2016 at 11:59:59 PM

Introduction.

This homework is on procedural SQL. You will use the same Yelp database from Homework #3 for this homework. Refer to Homework #3 for the data set and schema for Yelp data. This homework will require you to create database objects which you cannot do in the shared databases. As a result, we have created a database for each student in the class. All objects you create in these databases are only visible to you unless you give specific access to others. To keep your work private, I recommend you refrain from doing so.

Database server is available at: http://csci4380rpi.cloudapp.net/phppgadmin

Note that if you are using your private database in the server, it is highly recommended that you use the folder named `yelp_smallerfiles.zip` posted on Piazza which breaks the Yelp data into smaller files. The larger files do not seem to work over PhpPgAdmin.

You are welcome to discuss the algorithm for solving this homework with others, but do not show your code. It is important that you gain experience writing procedural code.

In this homework you will be implementing a simple social recommender system using pl/pgsql. The idea is that you will be making recommendations for businesses you may like, given a set of users who wrote reviews you like or dislike.

Suppose there is a new table in your database:

```
create table favorites (
      user_id varchar(40) primary key
      , vote int check (vote = 1 or vote = -1)
      , foreign key (user_id) references users(user_id)
) ;
insert into favorites values('1w_QkIakyV7zvNCCuuyqUQ',1) ;
insert into favorites values('4ujDwiek_48n2Q4NzJb6zA',-1) ;
insert into favorites values('x-aUZNvgXj--r0rd6iDMaQ',-1) ;
```

This table lists the users you like (vote=1) or dislike (vote=-1). We will test your code for different sets of favorite users.

You will write a procedure in pl/pgsql that takes as input a single string for a category of interest and top count, and implicitly use the `favorites` table to return a recommendation of businesses.

Problem Description

Assume the favorites table is already given and populated with the users. Write a pl/pgsql function named:

```
recommend(category VARCHAR, topusercount int)
```

that returns a string with name, city of businesses and a score of relevance on each line, formatted as shown below.

```
yelp=> select recommend('Food',3);
                        recommend
-----------------------------------------------------------------
 Karavalli                          Latham        10.40+
 The Fresh Market                   Latham        9.63 +
 DeFazio's Pizzeria                 Troy          9.17 +
 The Placid Baker                   Troy          9.17 +
 Dinosaur Bar-B-Que                 Troy          9.15 +
 Ala Shanghai Chinese Cuisine       Latham        8.68 +
 X's To O's Vegan Bakery            Troy          8.33 +
 Snowman                            Troy          8.08 +
 Sushi X & Lounge                   Latham        7.37 +
 Bella Napoli                       Latham        7.20 +
```

The string must list businesses in the order of descending relevance. The returned list must contain at most 10 businesses (top 10), but there may be fewer businesses based on the category. See below for the description of `topusercount`.

Using the category and favorites to compute the recommendations

Find businesses of interest. First step is to find all businesses of interest. Given input category `C`, we want to find businesses that are classified in `C` or a related category.

To find the related categories, you need to find all categories that co-occur with `C` for at least 5 different businesses. For example, if `C=Food` and at least 5 businesses are classified both as `Food` and `Restaurant`, then `Restaurant` is a related category.

Given `C` and all related categories, find all businesses in this category.

Find users of interest and their importance. Now, we will look at users. We want to find users that are similar to the seed users, i.e. those users from the `favorites` table. We will compute a set of users called `recommenders` which are composed of the seed users, and users who are most similar to our seed users.

Suppose `UI` is an input user and `UO` is a different user in the database. To be able to judge similarity of `UI,UO`, `UO` must have at least 3 businesses rated in common with `UI`.

The similarity of two users is given by the average difference between their scores for each business they rated in common. We want to only add users who are in the top `topusercount` most similar for each seed user to the recommenders list (recall that topusercount is an input to the function). Of course, a single user may be considered as very similar to two different seed users. In this case, we will give that user a weight of two. This is a good time to take advantage of the bag semantics of SQL and list that user twice in the recommenders. The set of users in `Recommenders` will have attributes: `user_id`, `vote` and `similarity`.

- `vote` is +1 or -1, depending on whether the user is similar to (or equal to) a favorite with the same vote.
- `similarity` is the `1-(average distance between reviews)/5`. We divide by 5 to normalize the distance to the 5 star scale.

  For a user in the favorite table, similarity is 1.

What happens if a user is most similar to one of our favorites with vote= +1 and another one with vote=-1? If so, we must remove this user. Their point of view is ambiguous. If a user is both a favorite with vote 1, but is also added as similar to another favorite user with negative vote, we will remove the similar user entry but keep the favorite user entry.

Rank businesses of interest by the user reviews. Our final step is to rank the businesses of interest we computed above by their match to the recommenders. Basically, we want to return businesses that are liked by the users we trust (vote=1) and weigh their recommendations by their similar to the people we trust.

How about users we do not trust (vote=-1)? Well, we do not trust their opinion. If they did not like the business, that does not mean much to us. But, if they really liked the business, may be it lowers our opinion of the business a bit. So, we will give a weight of `0.1` to reviews of 4 or 5 stars by these users. The weight is likely to be adjusted based on user testing or even customizing the user feedback over time (explicit or implicit).

Here is how it all comes together:

For each business of interest, if the business is not rated by any of the recommender users, give it a score of 0. For all other businesses, the score is given by: $A - B$ where

$A = sum(similarity * stars)$ sum of ratings for users we trust
$B = 0.1 * sum(similarity * stars)$ sum of ratings of 4 or 5 for users we do not trust

Rank all businesses by this score (descending) and then alphabetically by business name (ascending), return the top 10 such businesses (or all if there are a total of less than 10 businesses in the list).

Requirements

Submit a single file called `hw5ans.sql`. Your file must contain a single procedure definition and nothing else. All your functionality must be encapsulated within the procedure. Do not define other functions or other structures.

We will test your program by first creating the function, then populating the favorite table with different input and running your function to see the output.

Your function must be named exactly `recommend` and must take as input a VARCHAR and an INTEGER. Your function must not change any of the existing relations in the database. However, you are free to create new tables in your function to store temporary values. When your function finishes execution, all these tables must be dropped, leaving no changes. While it is possible to use `TEMPORARY TABLES` that are dropped at the end of a session for this, this sometimes causes a problem with our testing code.

Instead, you should manually drop the tables yourself at the end of the function execution. Beware though if your function fails to complete, the tables may fail to drop. Test your code thoroughly before submitting. As we want to test your functions automatically, it is very important that you satisfy these requirements, otherwise it will slow out down our testing and you will be penalized.

Also your submission must have the following format, to drop the function first and then recreate it (to allow us to test multiple functions in sequence):

```
drop function recommend(varchar, integer) ;
create function recommend(input_category varchar, topusercount integer) returns varchar
    AS $$
BEGIN
    RETURN 'Hello world!';
END ;
$$ LANGUAGE plpgsql ;
```

For your testing purposes, you can run this function using:

```
select recommend('a', 3) ;
```

Note

There is a lot to unpack in this homework. I will post an example output for you to test with. Write in small pieces and test each piece. I used three helper tables to solve this. My code is around 133 lines, with lots of lines used for formatting. Use tables and queries strategically, then the problem is not that difficult.

Also, if you are using tables to store intermediate computations (and I recommend that you do), first do not drop these tables when the function ends so that you can test their contents. Once you are sure everything works, then write your function in a way that it creates tables as it needs and uses then, and then drops all the tables before the final return.

Also, when you want to create a nicely formatting string, a few pointers:

`E'\n'` is a new line

You can format a string by padding it with spaces, for example:

```
SELECT rpad('Hello world', 25);
```

You can format a float by casting it to a value:

```
SELECT cast(4.213123123 as numeric(4,2));
```

My input formatted the name by 35 characters, city by 12.