Database Systems, CSCI 4380-01
Homework # 6
Due Monday November 1, 2021 at 11:59:59 PM

Homework Statement. This homework is worth 3% of your total grade. This homework is completely optional. If you skip this homework, midterm #2 will be worth 3 points more.

This homework will concentrate on advanced SQL skills and procedural SQL.

Since this homework requires you to create new tables and functions, you must your own database, already created for you in the course database. It will be named: `db_⟨yourusername⟩`.

I have already created a small database for testing your function in each of your databases (also online as `radiodb_hw6`). It will only contain the relevant radio database tables, but the same content as the previous database. Please do not change the already given tables as it will be costly for me to recreate them. Do all your work by creating new database objects as described below.

Music recommendation is a difficult problem, trying to explain why people like some music more than the others. When getting a recommendation, do you prefer to get some really common song recommended to you that you might already know and like? How about if you get a recommendation for a rare gem, but it is rare because there are not that many people who like that song? What are the odds you are the right person for that song? Needless to say, there has been a lot of research on this topic.

Suppose you frequently tune in to a radio station at a specific hour, clearly you like the songs you hear there. Can I find similar songs to these ones you listened? This is what we will attempt in this homework with a very simple approach in this homework (no machine learning).

Problem Description

Create a single PL/pgSQL function called `recommendation` that takes as input:

- `fromtime, totime`: two time values,
- `inputstation`: a string of station name,
- `topk`: an integer, how many tuples/series to return from the function,
- `w1,w2,w3`: floats to be used as weights.

I have already provided you with a skeleton function description below. It shows you the return type (created as a table) and an example call.

In your hw solution, please keep the function name, input and output types the same as the one I provided you. Keep the create table command as well. Simply replace the function body with your solution.

We will test your code by first running your submission to create the table and the function, and then run your function on different inputs to test. So, make sure that your function can be executed multiple times without a problem once created. You can create other functions if you want to modularize your solution. You can create other objects too, but make sure they do not prevent your code from running multiple times.

A simple way to achieve this is to create your extra objects/tables inside the function and drop them before the function ends. One way to do that automatically is to use temporary tables. You can read about them in Postgresql documentation.

Functionality

Now, let us describe how your function is supposed to work. Given an input
`fromtime, totime, inputstation`, find all songs played by the `inputstation` between the given times. Your will first find songs that are played during these times on this radio station, let's call the set of these songs `songsplayed`. Your job is to find songs that are not in the set `songsplayed` but are similar to these songs.

How can we compute a numerical similarity?

Genre Similarity (`gs`*):* First, a song can be similar to `songsplayed` because it is in the same genre as many of the other songs in this set. So, for each genre, compute how many times they are present in a song in `songsplayed` as `genre_score` (i.e. total number of songs that have this genre, regardless of how many times it is played). Then, for each song, find the total genre score as the sum of `genre_score` for each genre the song has in common with a song in `songsplayed`. The higher this score is, the higher the genre similarity.

Rank similarity (`rs`*):* Secondly, a song can be similar because it was in bilboard rankings in the same average rank as a song from `songsplayed`. For each song, find its average bilboard ranking (`avgrank`) and the average bilboard ranking of songs from `songsplayed` in the same decade as this song (`avgrankplayed`). The rank similarity is then given by `1/abs(avgrank-avgrankplayed)`.

Songs with a null decade will have rank similarity of zero.

Song similarity (`ss`*):* Finally, a song can be similar because it is similar to songs in terms of its features. As there are many features in the songs table, we will concentrate on the following three for simplicity: `energy,liveness,acousticness`. For each song, find the total absolute difference of these features with the average value of the songs in `songsplayed`
(`avgenergy, avgliveness, avgacousticness`) as the song similarity. In other words, for each song `s`, compute:

```
ss=1/(abs(s.energy-avgenergy)+abs(s.liveness-avgliveness)
                        +abs(s.acousticness-avgacousticness))
```

Any song with a null value for any of these attributes will have a song similarity of zero. The songs from `songsplayed` with null values for these attributes will not contribute to the overall score.

Now add all scores for a song using the weights given in the input to the function. Hence, the final score for each song will be given by:

`songscore = w1*gs + w2*rs + w3*ss`.

Your function should sort all the songs by descending order of `songscore` and then by songname ascending and return the first `topk` songs. For each song, return the id, name, artist and `songscore` values. All the returned songs should be different than any song in the `songsplayed`.

Note that your output type is the same structure as the `songsimilarity` table, hence you will return a set of tuples of this table type. We discussed how to do this in class on Monday October 26th. You can refer to that and the pl/pgsql documentation for details. The example file provided with the hw has a skeleton query for this as well. Note that this method simply returns a set of tuples of this type, it does not actually insert the tuples into a table.

SUBMISSION INSTRUCTIONS. You will use SUBMITTY for this homework. Please submit a single file named `hw6.sql` that creates the function and does not run it.

If you want to provide any comments in the file, all SQL comments must be preceded with a dash:

```
-- Example comment.
```

Please double check your submission before submitting. We will not debug functions that do not run. Use a plain text editor for your submission, similar to what you would use for C++ or Python.

Database Schema

This database is merged from multiple datasets, containing data for songs (tracks) from different artists. We have data regarding the songs performance on spotify, bilboard and on various radio stations. As with Homework #3, I dedicate this database to WRPI!

Note that this is real data, scraped from websites, parsed and matched. It is likely very noisy. Make a habit of using simple queries to first explore the data to understand various issues.

```
-- All artists in the database
CREATE TABLE artists (
    id       bigint NOT NULL
    , name   text
    , PRIMARY KEY (id)
);

-- Song (or track) information: name, its artist as well as
-- features based on the audio analysis of the song from 1 million song db
-- decade is which decade the song is from

CREATE TABLE songs (
    id              bigint NOT NULL
    , name          text
    , artistid      bigint
    , uri           text
    , danceability double precision
    , energy        double precision
    , key           double precision
    , loudness      double precision
    , mode          double precision
    , speechiness   double precision
    , acousticness double precision
    , instrumentalness double precision
    , liveness      double precision
    , valence       double precision
    , tempo         double precision
    , duration\_ms  double precision
    , time_signature integer
    , chorus_hit    double precision
    , sections      integer
    , popularity    integer
    , decade        text
    , PRIMARY KEY (id)
    , FOREIGN KEY (artistid) REFERENCES artists(id)
);

-- Genre of the songs in the database, from spotify.
CREATE TABLE song\_genre (
    songid     bigint NOT NULL
    , genre    varchar(100) NOT NULL
    , PRIMARY KEY(songid, genre)
    , FOREIGN KEY (songid) REFERENCES songs(id)
);


-- Bilboard rank (between 1-100) of the songs in the database
-- Each row is for a specific song in a specific date of the Bilboard chart
-- Multiple songs may share a rank in a given chart date.
-- Lastweek, peakrank and weeksonboard are the statistics for a specific
```

```
-- song at the given chartdate.

CREATE TABLE bilboard (
    rank          integer
    , songid      bigint NOT NULL
    , lastweek    integer
    , peakrank    integer
    , weeksonboard integer
    , chartdate   date NOT NULL
    , PRIMARY KEY(songid, chartdate)
    , FOREIGN KEY (songid) REFERENCES songs(id)
);

-- For each song, we store when they were played on radio (based on
-- 8 different radio channels: mai,george,sound,rock,breeze,edge,magic,more
-- All radio stations are based in New Zealand (where I could find data from!)
-- Each time the song is played, we store the timestamp.
-- Note that this is a random sample of tuples from the original 855K tuples.

CREATE TABLE playedonradio (
    id           integer NOT NULL
    , songid     bigint
    , station    varchar(40)
    , playedtime timestamp without time zone
    , PRIMARY KEY (id)
    , FOREIGN KEY (songid) REFERENCES songs(id)
);

-- Based on Rolling Stone Magazine's list of the 500 Greatest Albums of
-- All Time, originally published in 2003, slightly updated in 2012.
```