CS 122A: Introduction to Data Management – Fall 2021

Homework 6: More SQL (Hands-on) (100 points)

Due Date: **Fri, Nov 12 (6:00 PM Pacific)**

Submission

All HW assignments must be submitted online via the HW6 dropbox on Gradescope. See the table below for the HW 6 submission opportunities. Note that after 6 PM on Saturday the 13th no further HW 6 submissions will be accepted. (We will be releasing the solution at that time.) Please strive to get all your work in on time! If possible, try to save the one dropped assignment for the end of the term when you are most likely to want/need it.

Date / Time	**Grade Implications**
Fri, Nov 12 (6:00 PM Pacific)	Full credit will be available
Sat, Nov 13 (6:00 PM Pacific)	10 points will be deducted

More Structured Query Language (SQL)

Please continue using the HW 5 dataset in this assignment. You are going to use PostgreSQL for all of the queries in the assignment and turn in the queries and results according to the provided template.

Write the following queries in SQL against the Swoosh test database. Show the result of each query where requested to do so. Please note that you will not get points for providing the result of a query on this assignment if your SQL query is syntactically incorrect (i.e., if it doesn't execute). Since you have a "live" system at your disposal, this should not be an issue – you can easily run all of your queries that way. **Also, make sure that the result of your queries do not contain duplicate records as you will lose points for that.** Note that some of the problems' solutions will have side effects on the database; you can use other SQL statements to undo those effects while debugging, and in the worst case you can always reload the data if you inadvertently mess it up while getting everything working. When you run your answers to turn them in, start with a clean load and run them all in sequence.

NOTE: For questions requiring you to write DDLs (e.g., views, triggers, stored procedures, alter table), you must write the required DDL statements **by hand**. You will **not** get the points for a problem if you let a GUI tool generate the DDL for you!

ALSO NOTE: Read the **NOTE** above one more time. We're dead serious about this...!

1. [10pts] For all students who enrolled in at least 3 courses after the date 2020-09-24, list the students' ids, first_name and last_name.

a) [7pts] SQL Query:

b) [3pts] Result:

2. [10pts] Find the longest meeting duration for each course taught by the instructor with user_id '486'. Show only the course_id, course_name, and the maximum duration. Rank the results by the durations from highest to lowest.

a) [7pts] SQL Query:

b) [3pts] Result:

3. [10pts] For all posts that are liked by at least 0.4% of all users, list the post_id and the number of users who thumbed up the post. Rank your results by the number of users (likers) from the highest to the lowest.

a) [7pts] SQL Query:

b) [3pts] Result:

4. **Views** [20 pts]

It's finally time to complete the last piece of the puzzle - the "good_posts" attribute from the E-R diagram that we designed earlier. Recall that "good_posts" is a derived attribute, and it's an accounting of the total number of "thumbs up"s that each student has **received**. Note that while each post can get multiple "thumbs up"s, we want to consider each such good post as **only one good post**. In addition, if a student does not have any good posts, their number of good posts should be zero. To implement this, we need to create a view. The view should include each students' user_id and their number of good posts.

a) [15 pts] Create the desired view StudentView by writing an appropriate CREATE VIEW statement.

CREATE VIEW swoosh.StudentView (user_id, occupation, good_posts) AS SELECT …;

b) [5 pts] Show the usefulness of your view by writing a SELECT query against the view that prints the user_id, first_name, and last_name of the student who received the largest number of good posts. Rank the results by the number of good posts from the highest to the lowest.

5. **Stored Procedures** [20 pts]

a) [15 pts] Create and exercise a SQL stored procedure called RegisterInstructor(…) that the application developer can then use to add a new instructor with one piece of initial optional education info to the database.

CREATE PROCEDURE RegisterInstructor(

IN user_id text,
IN email text,
IN first_name text,
IN last_name text,
IN title text,
IN “degree” text,
IN graduation_year integer,
IN major text,
IN school text,
IN education_id text
)
LANGUAGE SQL AS … ;

b) [5pts] Verify that your new stored procedure works properly by calling it as follows to add a new instructor and then running a SELECT query to show the stored procedure’s after-effects:

Result:

6. **Alter Table** [10 pts]

a) [5 pts] Write and execute the ALTER TABLE statement(s) needed to modify the recording table so that when the meeting associated with a recording is deleted, the recording will ***not*** also be deleted. It should be retained instead. (*Note:* The name of the existing foreign key constraint for the meeting_id field is *recording_meeting_id_fkey*).

b) [5 pts] Execute the following DELETE and SELECT statements to show the effect of your change. Report the COUNT query’s result (just the number) returned by the SELECT statement both before and after your DELETE.

```
SELECT COUNT(*)
FROM recording r
WHERE r.meeting_id = '50';

DELETE FROM meeting
WHERE meeting_id = '50';

SELECT COUNT(*)
FROM recording r
WHERE r.meeting_id = '50';
```

Result:

7. **Triggers** [20 pts]

a) [15 pt] Create a table watchedvideo(recording_id, user_id, PRIMARY KEY(recording_id, user_id)) that stores the recordings watched by students as well as the students who watched them. Then write a CREATE TRIGGER statement (**by hand** of course!) to define a trigger that will do the following job: after a student has watched a segment of a recording -- indicated by an insert into the watchedsegment table -- if the sum of the student's watched segment durations of a recording add up to at least the recording length, we say the student has (in fact) watched the recording, and we therefore insert the recording_id and the student's user_id into the watchedvideo table. The new table is only responsible for keeping the watching records after the trigger is created. Use the CREATE FUNCTION statement as well as needed. Your function should ignore duplicate inserts to the new table. HINT: use "...ON CONFLICT..." to handle insertion conflicts.

b) [5 pts] Execute the following INSERT and SELECT statements to show the effect of your trigger. Report the results.

```
SELECT *
FROM watchedvideo
WHERE recording_id = '2252';
Result:
```

```
INSERT INTO watchedsegment(recording_id, user_id, segment_id, watched_from,
watched_to)
VALUES ('2252', '0', '10', '17:00:00', '17:20:00');
SELECT *
FROM watchedvideo
WHERE recording_id = '2252';
Result:
```

```
INSERT INTO watchedsegment(recording_id, user_id, segment_id, watched_from,
watched_to)
VALUES ('2252', '0', '11', '17:00:00', '19:00:00');
```

```
SELECT *
FROM watchedvideo
WHERE recording_id = '2252';
Result:
```

```
INSERT INTO watchedsegment(recording_id, user_id, segment_id, watched_from,
watched_to)
VALUES ('2252', '0', '12', '17:00:00', '18:09:00');
SELECT *
FROM watchedvideo
WHERE recording_id = '2252';
Result:
```