Assignment: SQL–The Structured Query Language

COMPSCI 2DB3: Databases–Winter 2023

Deadline: February 8, 2023

Department of Computing and Software
McMaster University

Please read the *Course Outline* for the general policies related to assignments.

Plagiarism is a *serious academic offense* and will be handled accordingly.
All suspicions will be reported to the *Office of Academic Integrity*
(in accordance with the Academic Integrity Policy).

This assignment is an *individual* assignment: do not submit work of others. All parts of your submission *must* be your own work and be based on your own ideas and conclusions. Only *discuss or share* any parts of your submissions with your TA or instructor. You are *responsible for protecting* your work: you are strongly advised to password-protect and lock your electronic devices (e.g., laptop) and to not share your logins with partners or friends! If you *submit* work, then you are certifying that you have completed the work for that assignment by yourself. By submitting work, you agree to automated and manual plagiarism checking of all submitted work.

Late submission policy. Late submissions will receive a late penalty of 20% on the score per day late (with a five hour grace period on the first day, e.g., to deal with technical issues) and submissions five days (or more) past the due date are not accepted. In case of technical issues while submitting, contact the instructor *before* the deadline.

Description

A local group of movie fanatics want to start a review website on which they can place movie reviews and visitors can partake in the discussion for these reviews. As the first step toward this website, the group has asked a programmer to make a quick sketch of what kind of information the website database needs to collect and store. The movie fanatics wondered whether the database schema produced by the consulted programmer was flexible enough to support the ideas they have for the website. Following is the high-level description of the database schema proposed by the programmer:

- **user**($\underline{id}$, *name*, *pwdhash*, *rname*).

  The **user** relation contains a unique user id, the username (which should be unique), and a password hash (*pwdhash*) used to verify login attempts. For authors (that can write articles on the website), the user table also stores their real name (*rname*) that is used to credit the author of an article.

- **article**($\underline{id}$, *aid*, *title*, *body*, *rtime*).

  The **article** relation contains the articles written for the website (e.g., reviews). Each article has a unique identifier, the author identifier (*aid*, which refers to an author in the **user** relation), and the title of the article, the body of the article, and the date and time of the review (*rtime*).

- **comment**($\underline{id}$, *aid*, *uid*, *body*, *on_id*).

  The **comment** relation contains comments on articles. Each comment has a unique identifier, the article identifier of the article the comment belongs to (*aid*, which refers to an article in the **article** relation), the user identifier of the user that wrote the comment (*uid*, which refers to a user in the **user** relation), and the body of the comment. In addition, the website will support *threaded* comments: a comment can react directly on an article (*on_id* is **NULL**) or can react on another comment (*on_id* is the identifier of that other comment).

- **film**(*id*, *title*, *year*).

  The **film** relation contains the films discussed in articles on the website. Each film has a unique identifier, a title, and the year of release.

- **fcategory**(*fid*, *category*).

  The **fcategory** relation relates films (*fid*, which refers to a film in the **film** relation) to the categories of this film (e.g., romcom, drama, thriller, horror, action).

- **fmention**(*aid*, *fid*, *score*).

  The **fmention** relation relates articles (*aid*, which refers to an article in the **article** relation) to the possible-multiple films the article discusses (*fid*, which refers to a film in the **film** relation). Each discussed film receives a reviewer-score *score*.

An example database with these tables and with some example data is provided in the file `sql_example.txt`. Read the comments in that file to learn how to use the example data.

The requested queries

The movie fanatics are interested in the following queries:

1. *Find all authors*

   The movie website will need a dedicated page that lists all authors.

   Query: Write a query that returns a copy of the **user** table that only contains those users that are *authors*. Order the result on the real name of authors. If two authors have the same real name, then order them on their username (which should be unique).

2. *Connect reviews with their categories*

   The movie website will also get a category tab where all categories are listed. After opening a category, the user will see all reviews based on that category.

   Query: Write a query that returns pairs (*aid*, *category*) that relates article identifiers *aid* of reviews with the categories of the films mentioned in the review. Only mention each category once per review, even if that review mentions several films in the same category.

3. *Find interesting reviews.*

   One way to keep visitor retention high is by recommending visitors films they might be interested in. To do so, the website can recommend users reviews they have not yet commented on.

   Query: Write a query that returns pairs (*uid*, *aid*) of user identifiers (*uid*) of users u and article identifiers (*aid*) of reviews r such that user u did not comment on r.

4. *Related films.*

   Another way to keep visitor retention high is by linking related films under a review. One way to do so is by finding films that have the same categories as the films mentioned in the reviews.

Query: Write a query that returns pairs (*aid*, *fid*) that relates article identifiers (*aid*) of reviews r to film identifiers (*fid*) of films f such that f is *not* mentioned in r, but f is has *exactly the same categories* as a film f' that is mentioned in a.

5. *Compute review overviews.*

   The main page of the website will display the list of recent reviews. This list will provide the review title, author information, and a summary of the discussion on the article.

   Query: Write a query that returns tuples (*aid*, *title*, *name*, *tcount*, *rcount*) such that *aid* is the article identifier of the review, *title* is the title of that review, *name* is the real name of the author of the review, *tcount* is the number of distinct discussion threads that comment on the review (the number of reactions placed directly on the review), and *rcount* is the total number of reactions placed on the review. Order the table such that the most-recent review comes first.

6. *Category interest scores.*

   To further improve recommendations, one can recommend new reviews based on the category of the films they review: if a user mainly is interested in *romcom*s, then recommend new reviews of *romcom*s.

   To find the favorite categories of users, the movie critics propose an *interest score.* The interest score of a user u for a category c is the number of reviews user u commented on that review a film associated with category c.

   Query: Write a query that computes the interest scores of the category the user is most-interested in: the query must return triples (*uid*, *category*, *score*) such that *score* is the interest score of the user u with id *uid* in the category *category* such that *score* is the highest of all the interest scores for user u.

7. *Find films in related categories*

   As a last option for recommendations, the system can recommend reviews on films that are similar to categories the user is interested in. To find *most similar categories* without manually programming them into the system, we want to relate to each category the category it shares the most films with. Hence, a category c is most similar to another category d if the number of films that are related to both c and d is the maximum of the number of films that are related to c and any other category.

   Query: Write a query that computes triples (*category*, *fcategory*, *fid*) that relates category (*category*) to film identifiers (*fid*) that *do not belong* to *category*, but do belong to the category *fcategory* that is most similar to *category.*

8. *Find discussion starters.*

   Another way to maximize visitor retention is by increasing engagement, e.g., by stimulating discussions on reviews. One way to do so is by providing influential users with rewards (thereby further incentivizing their active participation on the website). The first attempt at doing so will aim at identifying *discussion starters*: those users whose reactions often elicit many responses. To reward these discussion starters, these users will receive a badge on the website.

   Query: Write a query that returns a list of user identifiers of users that write comments that, on average, receive at-least twice the average number of responses on a comment (a response on a comment c is any comment that reacts on comment c).

9. *Scoring trends.*

   The appreciation of some films increases over time, whereas opinions on other films decreases over time. To provide insight in the trend of a film score, the website will indicate either an *upward* (scores go up), a *stable*, or a *downward* (scores go down) trend for each film. Unfortunately, all authors utilize their own scoring systems, making straight-up comparison between distinct authors unfair. Hence, the score trend of film f will be based on the scores given by authors that wrote multiple reviews for the film f.

Specifically, the trend score of a film is given by $m - n$ in which n is the total number of reviews f received that are *follow-up reviews* (reviews written by an author that already wrote an article about f) and m is the total number of follow-up reviews that film f received in which the author a gave film f a higher score than author a did in the preceding review for film f. Hence, films without follow-up reviews have a trend score of 0 (stable).

Query: Write a query that returns pairs (*fid*, *ts*) that relates film identifiers *fid* of films f with the trending score of f.

10. *Rank films on normalized film scores.*

    Visitors are always interested in the ranking of films (e.g., which film is the best, which one is the worst). As mentioned previously, all authors utilize their own scoring systems, making straight-up comparison between distinct authors unfair. Alternatively, we can assign an *normalized score* to each film.

    The normalized base-score for a film f is the fraction $\frac{x}{y}$ in which y is the number of initial reviews of film f, and x is the sum of the normalized reviewer-scores for film f.

    A review of film f by author a is the *initial review* of film f by author a if it is the earliest review by author a that mentions f. The normalized reviewer-score for film f by author a is *one* if, upon the initial review r of film f by author a, author a assigned a score that was *at-least* as high as the average score author a gave to films upon that point in time (taking into account only the scores assigned in reviews the author wrote before the initial review r). If the assigned score is *lower* than the average, then the normalized reviewer-score for film f by author a is *zero*.

    Query: Write a query that returns pairs (*fid*, *nbc*) that relates film identifiers *fid* of films f with the normalized base-score of *fid*. Order the result on decreasing normalized base-score.

Assignment

Write the above queries. Hand in a single plain-text file (txt) with each of the requested queries in the following format:

```
-- Query [number]
[the query]

-- Clarifications: [any description].


[next query]
```

In the above, `[number]` is the query number (1, 2, 3, 4, 5, 6, 7, 8, 9, 10). Your submission:

1. must use the constructs presented on the slides or in the book (we do *not* allow any SQL constructs that are not on the slides or in the book);
2. must work on the provided example dataset when using IBM Db2 (on `cs2db3.cas.mcmaster.ca`): test this yourself!

Submissions that do not follow the above requirements will get a grade of zero.

Grading

All ten queries get the same mark. You get the full marks on a query if it solves the described problem exactly. We take the following into account when grading:

1. Is the query correct (does it solve the stated problem)?

   Hint: Your queries must not only work on the provided example dataset in file (`sql_example.txt`), but also on all other valid datasets: think about edge cases (e.g., empty tables).

2. Are all required columns present?
3. Are no superfluous columns present?
4. Does the result satisfy the ordering requirements provided?
5. Does the result have unnecessary duplicate values (none of the queries should have duplicates in their output)?
6. Are no superfluous statements present (e.g., **DISTINCT** when the query cannot produce duplicates, **ORDER BY** if the query does not need to be ordered, **GROUP BY** a column that is already unique)?

If you are stuck and cannot solve a query, then provide a query showing the parts that you managed to solve and describe in your clarification what you managed to solve. We will give partial grades for solutions that are not complete, but do solve a *core component* correctly.