Assignment 2: SQL Queries (DML)

Jelle Hellings

3DB3: Databases – Fall 2021

Department of Computing and Software
McMaster University

Deadline: October 22, 2021

Cheating and plagiarism. 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. 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 the submitted work.

Cheating and plagiarism are serious academic offenses. All cases of academic dishonesty will be handled in accordance with the Academic Integrity Policy via the Office of Academic Integrity.

Late submission policy. There is a late penalty of 20% on the score per day after the deadline. Submissions five days (or later) after the deadline are not accepted. Do not wait until the deadline to ask questions or raise problems.

Description

A local community leader wants to build an event website for advertising local events. The leader already worked with a consultant to construct a detailed relational schema that describes how all necessary information is stored. Next, the leader wants to know whether it is possible to extract the necessary data for the operation of the event website from this schema. The relational schema consists of the following five relations (SQL tables):

- **user**(uid, name, regdate, postcode).

  The **user** relation contains the user name, registration date (*regdate*), and an optional postal code (*postcode*) of registered users of the website. The postal code will be used to recommend each user with events in its area.

- **event**(eid, title, description, startdate, enddate, organizer, postcode).

  The **event** relation contains the title (or name), description, start and end date, organizer, and postal code (*postcode*) of events. The *organizer* is stored by a user identifier (a value in the column *uid* of the table **user**).

  Events can be organized in other regions than where their organizer is based. Hence, the event postal code does not have to match the postal code of the organizer.

- **keyword**(event, word).

  Each event can be labeled with zero-or-more keywords that describe the event and can be used as search criteria. E.g., a guided tour of a museum with an exhibition on a famous painter could be labeled with: ‘art’, ‘painting’, and ‘exhibition’. The *event* is stored by an event identifier (a value in the column *eid* of the table **event**).

- **region**(postcode, name).

  To simplify searches based on regions (e.g., the golden horseshoe area), postal codes (*postcode*) are mapped to the commonly-used names of regions they are part of. Note that a postal code can be part of many regions and a region can have many associated postal codes.

- **review**(user, event, description, score, reviewdate).

  All users can review events. Such a review consists of a description, a score (0-10), and the date at which the review was written. The *user* is stored by a user identifier (a value in the column *uid* of the table **user**) and the *event* is stored by an event identifier (a value in the column *eid* of the table **event**).

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 community leader is interested in the following queries:

1. High-level query: 'Find all multi-day events'. Detailed description:

   > The community leader wants to include filters on the website to search based on the length of events (e.g., to distinguish between a concert and multi-day festivals).
   >
   > Write a query that returns a copy of the event table that only contains those events that are multi-day events. Order the events on their increasing start date and (if events have the same start date) in alphabetical order on title.

2. High-level query: 'Find all events in a specified region'. Detailed description:

   > Another useful filter for events is filtering based on region. To illustrate these capabilities, write a query that returns all events (columns *eid* and *title* of the **event** table) that are organized in the golden horseshoe area. The community leader specified that the list of events will be presented in alphabetical order on title.

3. High-level query: 'Label users'. Detailed description:

   > Not every users of the website will end up organizing events. To make it easier to distinguish between normal users and event organizers, the community leader wants to make the normal active users easily distinguishable (e.g., user-name printed in a different color).
   >
   > Write a query that returns all normal users (columns *uid* and *name* of the **user** table) that are active (have written reviews), but are not organizers of events. As this list is used internally, it does not have to be ordered in any particular way.

4. High-level query: 'Display event statistics'. Detailed description:

   > When providing an overview of events, the community leader would like to also display the number of reviews and the average review score.
   >
   > Write two queries:
   >
   > (a) a query that returns the event information (the columns from table *event*) augmented with their score statistics consisting of two additional columns *nrev* (number of reviews) and *ascore* (the average review score); and
   >
   > (b) a query that returns the event information augmented with their score statistics, but only includes those events that have a *significant number of reviews* (at-least-5 reviews).

In both cases, order the events on decreasing review score and (if events have the same score) in alphabetical order on title.

5. High-level query: 'Find suspicious events'. Detailed description:

   To keep the website usable, the community leader needs to account for events with *false* reviews (which then can be flagged or removed). Two strong indicators for false reviews are reviews that are written before the event starts or that are written by the organizer of that event.

   Write a query that returns a list of all suspicious events (columns *eid* and *title* of the **event** table) that have such reviews. As this list is used internally, it does not have to be ordered in any particular way.

6. High-level query: 'Find all local-only users'. Detailed description:

   From past experience, the community leader has notices that some users only visit events in their own region. The community leader wants to be able to identify these users (such that these users only get recommendations for events in their own region).

   Write a query that returns all users (columns *uid* and *name* of the **user** table) that have only reviewed events that happened in a region associated with the location of that user. As this list is used internally, it does not have to be ordered in any particular way.

7. High-level query: 'Find similar events'. Detailed description:

   A second important part of the *recommendations* is to recommend events. One way to select recommendations is by selecting similar events.

   Write a query that returns pairs of distinct events (both identified only by their event identifier, first column renamed to *fstid*, second column to *sndid*) such that the events have exactly the same keywords associated with them. Sort the output lexicographically on *fstid* and *sndid* such that the output has, for each event identified by *fstid*, a group of related events.

8. High-level query: 'Find popular events'. Detailed description:

   A second important part of the *recommendations* is to recommend popular events. Popularity can be measured by the amount of reviews and the quality score assigned by each review. In the first iteration, the community leader wants to test whether the sum of all review scores of an event (which takes into account both the amount of visitors and their score) is a good starting indicator for popularity.

   Write two queries:

   (a) a query that returns the event (column *eid* of the **event** table) together with the *popularity score* of that event (column *pscore*); and

   (b) a query that returns the most popular event based on its *popularity score* (columns *eid* and *title* of the **event** table). If several events have the highest popularity score, then the query can return each of these events.

   In both cases, order the events on decreasing popularity score and (if events have the same score) in alphabetical order on title.

9. High-level query: 'Find expert users'. Detailed description:

   The event website is community-driven and will only succeed with enthusiastic participation of its users. To encourage such participation, highly-active users are rewarded with badges. Badges are awarded for the following:

- Users that have written the most reviews for events with a specific keyword k will get a keyword badge.
- Users that have written the most reviews for events in a specific region r will get a region badge.

> Write a query that returns the list of users (column *uid* of the **user** table) that have badges and their badges (the string 'keyword' or 'region'). Each user should only be in the result if it has a badge. If a user has both types of badges, then the user should be twice in the result.

HINT: The query '**SELECT** `'value'` **FROM** `table;`' will return a row with the string value 'value' for every row in table 'table'.

HINT: First write two separate queries: one that only finds the users with a keyword badge, and one that only finds the users with a region badge.

10. High-level query: 'Provide score indicators'. Detailed description:

> Every reviewer uses different ways to determine review scores. E.g., some reviewers always give low scores, while other always give high scores. To help users to interpret reviewers, the community leader wants us to provide a scoring indicator for each reviewer that specifies whether that reviewer scores low or high on average.
>
> Consider a review for event E by a user U with score s. Let a be the average score for event E when considering all reviews. We say that user U scored low with *low-factor* $(a - s)$ if $s < a$, scored standard if $s = a$, and scored high with *high-factor* $(s - a)$ if $s > a$. Let M be the number of reviews, L be the sum of all low-factors for reviews by user U, let H be the sum of all high-factors for reviews by user U, and let S be the number of standard-scored reviews. The scoring indicator of user U is given by
>
> $$\frac{(L + H) - S}{M}.$$
>
> Write a query that returns users (column *uid* of the **user** table) and their scoring indicator (a column named *si*) for all users *that have written reviews*. Order the table such that the users with the highest scoring indicator appear first and the users with the lowest scoring indicator appear last.

HINT: For this query and the score ranges provided (0–10), the precision provided by **DECIMAL** is more than sufficient.

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, 4a, 4b, 5, 6, 7, 8a, 8b, 9, 10). Your submission:

1. must use the constructs presented on the slides or in the book (we do *not* allow any system-specific constructs that are not standard SQL);

2. must work on the provided example dataset when using DB2 (on `db2srv2`): 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 doesn't 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 solve and describe in your clarification what you managed to solve.