Explanation and Model Solution for Example Assignment: SQL–The Structured Query Language

COMPSCI 2DB3: Databases

Jelle Hellings Holly Koponen

Department of Computing and Software
McMaster University

Foreword

Each query in this document consists of three parts. First, there is the original query description (from the assignment). Next, there is the rational of our SQL solution. Finally, there is the resulting SQL query.

Model Solution

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.

   Rational:
   This is a basic **SELECT-FROM-WHERE** query that returns those events from **event** that satisfy the condition "startdate < enddate" (multi-day events). To order on start date and title, we include the clause "**ORDER BY** startdate, title".

   Solution:

   ```
   SELECT *
   FROM event
   WHERE startdate < enddate
   ORDER BY startdate, 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.

   Rational:
   We need to select those events from the **event** table whose postal code matches any of the postal codes in the golden horseshoe area region:

```
SELECT eid, title
FROM event
WHERE postcode IN (postal codes of the golden horseshoe area);
```

To complete this query, we need to find all postal codes that are part of the "golden horseshoe area", for which the example data used the term "Golden Horseshoe". To do so, we simply retrieve that information from the **region** table:

```
SELECT postcode
FROM region
WHERE name = 'Golden Horseshoe';
```

Next, we fill in this postal code query as a subquery. Finally, we add the ordering clause "**ORDER BY** title".

Solution:

```
SELECT eid, title
FROM event
WHERE postcode IN (SELECT postcode
                   FROM region
                   WHERE name = 'Golden Horseshoe')
ORDER BY 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.

   Rational:

   We need to select those users from the **user** table that are *active* and not *organizing events*. Hence, we end up with the following basic structure for our query:

   ```
   SELECT uid, name
   FROM user
   WHERE user is active AND user is not an organizer;
   ```

   A user is *active* if the user has written reviews. Hence, the user identifier of active users can be found in the column "user" of the table **review**:

   user is active = `uid` **IN** (**SELECT** `"USER"` **FROM** `review`).

   Note that we wrote `"USER"` to obtain the user attribute, as DB2 recognizes `user` as a keyword. Finally, a user did not *organizing events* if their user identifier cannot be found in the column "organizer" of the table **event**:

   user is not an organizer = `uid` **NOT IN** (**SELECT** `organizer` **FROM** `event`).

   We combine these ingredients to end up with the solution.

   Solution:

   ```
   SELECT uid, name
   FROM user
   WHERE uid IN (SELECT "USER" FROM review) AND
         uid NOT IN (SELECT organizer FROM event);
   ```

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.

Rational:

To get the number of reviews and average review score (per event), we need information from the **review** table. In specific, we can group this table on the "event" field (which is the event identifier) and the compute the number of reviews in that group and the average review score in that group (hence, the average review score of each event):

```
SELECT event, COUNT(*) AS nrev, AVG(score) AS ascore
FROM review
GROUP BY event;
```

Note that not every event will be in the result of this query: events without reviews will not appear. To get Query 4a, we use the clarification from the Q&A, in which we clarified that `NULL` values are fine for events without reviews. Hence, we can do a **LEFT OUTER JOIN** to combine *all event information* from the **event** table with the event statistics for *those events that have reviews*.

```
SELECT E.*, S.nrev, S.ascore
FROM event E LEFT OUTER JOIN
       (SELECT event, COUNT(*) AS nrev, AVG(score) AS ascore
        FROM review
        GROUP BY event) S ON E.eid = S.event;
```

Note that we perform the aggregation in a subquery and only after computing the aggregation we combine the obtained data with the other attributes. This is always the preferred pattern when dealing with complex aggregates: aggregation after joining can often lead to the wrong results! (See also the slide "The **FROM** clause").

To finish Query 4a, we add the ordering clause "**ORDER BY** ascore **DESC**, title".

Solution:

```
-- Query 4a.
SELECT E.*, S.nrev, S.ascore
FROM event E LEFT OUTER JOIN
       (SELECT event, COUNT(*) AS nrev, AVG(score) AS ascore
        FROM review
        GROUP BY event) S ON E.eid = S.event
ORDER BY ascore DESC, title;
```

Rational:

For Query 4b, we are only interested in events with reviews. Hence, we can replace the **LEFT**

OUTER JOIN by a normal join. Finally, the requirement of at-least-5 reviews can be put at two places. First, we can check that requirement during the aggregate by adding a clause "**HAVING COUNT**(*) >= 5". Alternatively, we can check that requirement after joining with the **event** table in a **WHERE**-clause "**WHERE** nrev >= 5". I personally prefer the first solution, as it hints at an earlier filter step; but a sensible query optimizer should be able to apply this optimization for us.

Solution:

```
--  Query 4b.
SELECT E.*, S.nrev, S.ascore
FROM event E,
       (SELECT event, COUNT(*) AS nrev, AVG(score) AS ascore
        FROM review
        GROUP BY event
        HAVING COUNT(*) >= 5) S
WHERE E.eid = S.event
ORDER BY ascore DESC, title;


--  Query 4b (alternative).
SELECT E.*, S.nrev, S.ascore
FROM event E,
       (SELECT event, COUNT(*) AS nrev, AVG(score) AS ascore
        FROM review
        GROUP BY event) S
WHERE E.eid = S.event AND nrev >= 5
ORDER BY ascore DESC, 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.

   Rational:

   As the first step, we obtain all suspicious events that have reviews that are written before their event starts:

   ```
   SELECT eid, title
   FROM event E
   WHERE eid IN (SELECT event
                 FROM review R
                 WHERE R.event = E.eid AND R.reviewdate < E.startdate);
   ```

   As the second step, we obtain all suspicious events that have reviews written by the organizer of that event:

   ```
   SELECT eid, title
   FROM event E
   WHERE eid IN (SELECT event
   ```

```
                    FROM review R
                    WHERE R.event = E.eid AND R.user = E.organizer);
```

We can easily combine these two queries to get all suspicious events and end up with the solution.

Solution:

```
SELECT eid, title
FROM event E
WHERE eid IN (SELECT event
              FROM review R
              WHERE R.event = E.eid AND
                    (R.reviewdate < E.startdate OR R.user = E.organizer));
```

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.

Rational:

We want all users from the **user** table that only wrote reviews for events in their region. According to the clarification in the Q&A, this includes users that did not write any reviews. Hence, we want all users that have *not* written *non-local reviews* (reviews of events that are outside their region). Such a query would look like:

```
SELECT uid, name
FROM user
WHERE NOT EXISTS (a non-local review);
```

To find the list of users that write non-local reviews, we first have to be able to determine when a review is non-local. We consider a review to be local whenever the location of the user ("postcode" in table **user**) and the location of the event ("postcode" in table **event**) share the same region name in table **region**. Hence, the following query relates *all* postal codes that are in the same region:

```
SELECT R1.postcode, R2.postcode
FROM region R1, region R2
WHERE R1.name = R2.name;
```

Next, we can obtain list of all users that have written *non-local reviews* (reviews of events that are outside their region). To obtain this list, we join the **review** table with the **user** and the **event** tables to get the postal code of the reviewing user and the postal code of the reviewed event. Then, we check whether this pair of postal codes is *not* related to the same region, for which we can adapt the above query:

```
SELECT U.uid
FROM review R, user U, event E
WHERE R.user = U.uid AND R.event = E.eid AND
      NOT EXISTS (SELECT *
                  FROM region R1, region R2
                  WHERE R1.name = R2.name AND
                        R1.postcode = U.postcode AND
                        R2.postcode = E.postcode);
```

Finally, we combine these ingredients to end up with the solution.

Solution:

```
SELECT uid, name
FROM user U
WHERE NOT EXISTS (
    SELECT *
    FROM review R, event E
    WHERE R.user = U.uid AND R.event = E.eid AND
            NOT EXISTS (SELECT *
                        FROM region R1, region R2
                        WHERE R1.name = R2.name AND
                                R1.postcode = U.postcode AND
                                R2.postcode = E.postcode));
```

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.

   Rational:

   Events E_1 and E_2 must be distinct but similar in the sense that they have *exactly the same keywords associated with them.* Hence, there must not be a keyword of E_1 that is *not* a keyword of E_2 and vice versa:

   ```
   SELECT F.eid AS fstid, S.eid AS sndid
   FROM event F, event S
   WHERE F.eid <> S.eid AND
           NOT EXISTS (keyword of F that is not a keyword of S) AND
           NOT EXISTS (keyword of S that is not a keyword of F);
   ```

   Next, we will write a query that returns those keywords of X that are not keywords of Y:

   ```
   SELECT word
   FROM keyword KX
   WHERE KX.event = X.eid AND
           KX.word NOT IN (SELECT word
                           FROM keyword KY
                           WHERE KY.event = Y.eid);
   ```

   We fill in that query twice to end up with the solution.

   Solution:

   ```
   SELECT F.eid AS fstid, S.eid AS sndid
   FROM event F, event S
   WHERE F.eid <> S.eid
     AND NOT EXISTS (
       SELECT *
       FROM keyword KF
       WHERE KF.event = F.eid AND
   ```

```
        KF.word NOT IN (SELECT word
                        FROM keyword KS
                        WHERE KS.event = S.eid))
AND NOT EXISTS (
  SELECT *
  FROM keyword KS
  WHERE KS.event = S.eid AND
        KS.word NOT IN (SELECT word
                        FROM keyword KF
                        WHERE KF.event = F.eid));
```

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.

Rational:

To obtain the popularity score of each event with reviews, we simply group the table **review** on events (the attribute "event") and take the sum aggregation of the review scores score (the attribute "score").

```
SELECT event AS eid, SUM(score) AS pscore
FROM review
GROUP BY event;
```

Next, to return the event information of the most popular event based on its popularity score, we first want to find the event identifier of the most popular event(s). To do so, we require that the popularity score of these most popular event(s) is the maximum popularity score:

```
SELECT event AS eid
FROM review
GROUP BY event
HAVING SUM(score) >= ALL(SELECT SUM(score)
                         FROM review
                         GROUP BY event);
```

Finally, we look up the titles of these most-popular events in the **event** table.

Solution:

```
-- Query 8a.
SELECT event AS eid, SUM(score) AS pscore
FROM review
```

```
GROUP BY event
ORDER by pscore DESC;

--  Query 8b.
SELECT eid, title FROM event
WHERE eid IN (SELECT event AS eid
              FROM review
              GROUP BY event
              HAVING SUM(score) >= ALL(SELECT SUM(score)
                                       FROM review
                                       GROUP BY event))
ORDER BY 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.

Rational:

The keyword badges and the region badges are completely independent of each other. Hence, we are going to write queries for these separately and then simply take the union of these two queries:

```
SELECT uid, 'keyword' AS badge
FROM user
WHERE uid IN (users that get a keyword badge)
UNION
SELECT uid, 'region' AS badge
FROM user
WHERE uid IN (users that get a region badge);
```

Next, we have to determine which users deserve a keyword badge. As the first step, we will construct a table T that lists, for each pair user u and keyword k, how many reviews that u wrote for events with keyword k. This information can be obtained from tables **review** and **keyword** as follows:

```
SELECT R.user, K.word, COUNT(*) number
FROM review R, keyword K
WHERE R.event = K.event
GROUP BY R.user, K.word;
```

Next, we take from this table those users that have the highest number of reviews (for any given keyword):

```
SELECT R.user
FROM review R, keyword K
WHERE R.event = K.event
GROUP BY R.user, K.word
HAVING COUNT(*) >= ALL(SELECT COUNT(*)
                       FROM review S, keyword L
                       WHERE S.event = L.event AND
                             L.word = K.word
                       GROUP BY S.user);
```

Note that both the outer query and the subquery are based on the original query for table T. We have simplified the outer query and the subquery, however. The outer query now only return user identifiers, while the subquery only counts the number of reviews per user for a specific keyword.

The approach to get the users that qualify for the region badge is similar:

```
SELECT R.user
FROM review R, event E, region Rg
WHERE R.event = E.eid AND E.postcode = Rg.postcode
GROUP BY R.user, Rg.name
HAVING COUNT(*) >= ALL(SELECT COUNT(*)
                       FROM review S, event F, region Sg
                       WHERE S.event = F.eid AND
                             F.postcode = Sg.postcode AND
                             Sg.name = Rg.name
                       GROUP BY S.user);
```

Finally, we combine these ingredients to end up with the solution.

Solution:

```
SELECT uid, 'keyword' AS badge
FROM user
WHERE uid IN (
    SELECT R.user
    FROM review R, keyword K
    WHERE R.event = K.event
    GROUP BY R.user, K.word
    HAVING COUNT(*) >= ALL(SELECT COUNT(*)
                           FROM review S, keyword L
                           WHERE S.event = L.event AND
                                 L.word = K.word
                           GROUP BY S.user))
UNION
SELECT uid, 'region' AS badge
FROM user
WHERE uid IN (
    SELECT R.user
    FROM review R, event E, region Rg
    WHERE R.event = E.eid AND E.postcode = Rg.postcode
    GROUP BY R.user, Rg.name
    HAVING COUNT(*) >= ALL(SELECT COUNT(*)
```

```
FROM review S, event F, region Sg
WHERE S.event = F.eid AND
        F.postcode = Sg.postcode AND
        Sg.name = Rg.name
GROUP BY S.user));
```

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.

Rational:

First, we construct an intermediate table T_A that holds the average scores for each event (in the review table):

```
SELECT event, AVG(CAST(score AS DECIMAL)) AS a
FROM review
GROUP BY event;
```

Next, we are going to construct the numerator $(L + H) - S$ by constructing tables that hold, per user, each term in L, H, and S. First, a table T_L that holds all terms in L. Hence, the table T_L will hold, for each user u, the low-factor of every event that user u reviewed with a score below the average (for that event):

```
SELECT R.user, T_A.a - R.score AS v
FROM review R, T_A
WHERE R.event = T_A.event AND R.score < T_A.a;
```

Next, we construct table T_H in a similar manner:

```
SELECT R.user, R.score - T_A.score AS v
FROM review R, T_A
WHERE R.event = T_A.event AND R.score > T_A.a;
```

Then, we construct table T_S. Here, we notice that each review with the average score will count as a term -1 in the numerator. Hence, we have:

```
SELECT R.user, -1 AS v
FROM review R, T_A
WHERE R.event = T_A.event AND R.score = T_A.a;
```

Using these three tables, we can compute the numerator for each user that has written reviews. To do so, we construct table T_N that sums all the numerator-terms for each user as follows:

```
SELECT "USER", SUM(v) AS n
FROM (T_L UNION ALL T_H UNION ALL T_S)
GROUP BY "USER";
```

As the last step, we are ready to construct the denominator D. To do so, we construct table T_D that holds the number of reviews M for each user (in the review table):

```
SELECT "USER", COUNT(*) AS d
FROM review
GROUP BY "USER";
```

Finally, we combine tables T_N and T_D to compute the scoring indicators.

```
SELECT N.user AS uid, N.n / D.d AS si
FROM T_N AS N, T_D AS D
WHERE N.user = D.user;
```

As the query for T_A is repeated three times, we use a **WITH** clause to specify it only once.

Solution:

```
WITH ta(event, a) AS (
       SELECT event, AVG(CAST(score AS DECIMAL))
       FROM review
       GROUP BY event
)
SELECT N.user AS uid, N.n / D.d AS si
FROM (SELECT "USER", SUM(v) AS n
       FROM ((SELECT R.user, ta.a - R.score AS v
                FROM review R, ta
                WHERE R.event = ta.event AND R.score < ta.a) UNION ALL
               (SELECT R.user, R.score - ta.a AS v
                FROM review R, ta
                WHERE R.event = ta.event AND R.score > ta.a) UNION ALL
               (SELECT R.user, -1 AS v
                FROM review R, ta
                WHERE R.event = ta.event AND R.score = ta.a))
       GROUP BY "USER") AS N,
      (SELECT "USER", COUNT(*) AS d
       FROM review
       GROUP BY "USER") AS D
WHERE N.user = D.user;
```