TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Dr. Thomas Neumann

Exercises for *Foundations in Data Engineering*, WiSe 23/24

Alexander Beischl, Maximilian Reif (i3fde@in.tum.de)
http://db.in.tum.de/teaching/ws2324/foundationsde

Sheet Nr. 07

Exercise 1 In the last lectures you have learned about window functions. Now, we want to solve sheet 5's `exercise 3.2` & `3.3` using *window functions*. Again, use this WebInterface for the task. By clicking on the button `UniSchema` you can see the different relations. Use the expanded `examination` relation and the solution from `task 3.1`:

```
WITH examination(MatrNr,CourseNr,PersNr,Grade) as (
    SELECT * FROM pruefen
    UNION
    VALUES (29120,0,0,3.0), (29555,0,0,2.0),
           (29555,0,0,1.3), (29555,0,0,1.0)
),
grades(Name,MatrNr,Semester,Grade) as (
    SELECT s.name, s.matrnr, semester, avg(Grade)
    FROM studenten s, examination p
    WHERE s.matrnr = p.matrnr
    GROUP BY s.name, s.matrnr, semester
)
```

1. Based on the individual average grade, determine each student's rank within the student's cohort (students in the same semester) using *window functions*.

   Solution:

```
WITH examination(MatrNr,CourseNr,PersNr,Grade) as (
    SELECT * FROM pruefen
    UNION
    VALUES (29120,0,0,3.0), (29555,0,0,2.0),
           (29555,0,0,1.3), (29555,0,0,1.0)
),
grades(Name,MatrNr,Semester,Grade) as (
    SELECT s.name, s.matrnr, semester, avg(Grade)
    FROM studenten s, examination p
    WHERE s.matrnr = p.matrnr
    GROUP BY s.name, s.matrnr, semester
)
SELECT *, rank() over (partition by Semester
          order by Grade asc) as Rank
FROM grades n
ORDER BY n.Semester, Rank;
```

2. Additionally, for each student calculate the difference between the student's average grade and the cohort's average using *window functions*. (The cohort's average is the average of individual averages.)

Solution:

```
WITH examination(MatrNr,CourseNr,PersNr,Grade) as (
    SELECT * FROM pruefen
    UNION
    VALUES (29120,0,0,3.0), (29555,0,0,2.0),
           (29555,0,0,1.3), (29555,0,0,1.0)
),
grades(Name,MatrNr,Semester,Grade) as (
    SELECT s.name, s.matrnr, semester, avg(Grade)
    FROM studenten s, examination p
    WHERE s.matrnr = p.matrnr
    GROUP BY s.name, s.matrnr, semester
)
SELECT *, rank() over (partition by Semester
          order by Grade asc) as Rank,
       avg(Grade) over (partition by Semester) as GPA,
       avg(Grade) over (partition by Semester) - Grade as
          difference
FROM grades n
ORDER BY n.Semester, Rank;
```

Exercise 2 Window Functions in SQL

Analyze the salaries of all professors using *window functions* on the university schema. You may test your queries in the WebInterface. Use the example relation *Professors*:

```
WITH Professors (persnr, name, paygrade, room, salary, taxclass) as
(
  VALUES (2125,'Sokrates','C4',226,85000,1),
         (2126,'Russel','C4',232,80000,3),
         (2127,'Kopernikus','C3',310,65000,5),
         (2128,'Aristoteles','C4',250,85000,1),
         (2133,'Popper','C3',52,68000,1),
         (2134,'Augustinus','C3',309,55000,5),
         (2136,'Curie','C4',36,95000,3),
         (2137,'Kant','C4',7,98000,1)
)
```

1. Assign a rank to each professor, depending on their salary. Professors with the same salary should receive the same rank.

   Solution:

   ```
   SELECT *, rank() over (order by salary desc)
   FROM Professors;
   ```

2. Assign a rank within their pay grade to each professor, depending on their salary. Professors with the same salary should receive the same rank.

   Solution:

   ```
   SELECT *, rank() over (partition by paygrade
             order by salary desc)
   FROM Professors;
   ```

3. Calculate the running sum for each professor so that each professor's sum includes the salaries of those earning less or equal than they within their pay grade.

Solution:

```
SELECT *, sum(salary) over (partition by paygrade
          order by salary asc)
FROM Professors;

-- equivalent to

SELECT *, sum(salary) over (partition by paygrade
          order by salary asc
          range between unbounded preceding and current row)
FROM Professors;
```

4. Calculate the running average of each professor and the 2 above and below sorted by salary and partitioned by grade.

   Solution:

```
SELECT *, avg(salary) over (partition by paygrade
          order by salary desc
          rows between 2 preceding and 2 following)
FROM Professors;
```

5. Calculate the running average of each professor and the professors within a range of 5000€ more and less sorted by salary and partitioned by grade.

   Solution:

```
SELECT *, avg(salary) over (partition by paygrade
          order by salary desc
          range between 5000 preceding and 5000 following)
FROM Professors;
```

6. For each professor return the name of the one professor before and after him/her in the salary ranking. Professors with equal salary should be ordered by their name.

   Solution:

```
SELECT *, lag(name) over (order by salary desc, name asc),
          lead(name) over (order by salary desc, name asc)
FROM Professors;
```

7. Calculate the top 3 with and without window functions.

   Solution:

```
-- With window function
SELECT *
FROM (SELECT *, rank() over (order by salary desc)
      FROM Professors)
WHERE rank < 4

-- Without window function
SELECT *
FROM Professors p
WHERE 3 > (SELECT count(*)
           FROM Professors c
           WHERE p.salary < c.salary)
```

Exercise 3 The Hiking Quest

Charlie is quite fond of the idea of completing challenges. One of her current adventures includes completing very long hiking trails. Being a very organized person, but always short on spare time, she divides the trails into day-hikes. Whenever the sun is shining and time allows, she sets of to complete a section. On the Munich-Venice trail, she already completed section 1-3, 8-9 and 22-23. Furthermore, she completed some sections on a mediterranean island.

Her achievements can be given to a database like this:

```
WITH trails (id, leg) as (
   SELECT 1, leg FROM generate_series(1,28) leg
  UNION ALL
   SELECT 2, leg FROM generate_series(1,15) leg ),
completed (trail_id, leg) as (
   VALUES (1,1),(1,2),(1,3),(1,8),(1,9),(1,22),(1,23),
          (2,1), (2,2), (2,11), (2,12)
)
SELECT * FROM completed;
```

Support her cause by providing some SQL-queries. (Hint: Use window functions)

1. Create some motivating statistics:

   a) How many sections did she complete? (A section is a series of consecutive day-hikes)

      Solution:

      ```
      SELECT count(*)
      FROM (SELECT distinct trail_id,
                   leg - row_number() over
                   (partition by trail_id order by leg) section
            FROM completed);
      ```

   b) What is the average, minimum, maximum length of all completed sections?

      Solution:

      ```
      SELECT avg(cast(len as float)), min(len), max(len)
      FROM (SELECT count(*) as len
            FROM (SELECT trail_id,
                         leg - row_number() over
                         (partition by trail_id order by leg)
                            section
                  FROM completed)
            GROUP BY trail_id, section);
      ```

2. Help her plan the next trip: A long weekend is coming up, so Charlie can spend 3 days hiking. List uncompleted sections that consist of 3 or more day-hikes.

Solution:

```
SELECT id,
       min(leg) firstLeg,
       max(leg) LastLeg,
       max(leg) - min(leg) + 1 length,
       section
FROM (SELECT id, leg, leg - row_number() over w as section
      FROM trails
      WHERE not exists(SELECT *
                       FROM completed c
                       WHERE trail_id = trails.id
                         and trails.leg = c.leg
                      )
      window w as (partition by id order by leg)
     )
GROUP BY id, section
HAVING max(leg) - min(leg) + 1 >= 3
ORDER BY length;

-- Alternative solution using except all:
SELECT id,
       min(leg) firstLeg,
       max(leg) LastLeg,
       max(leg) - min(leg) + 1 length,
       section
FROM (SELECT id, leg, leg - row_number() over w as section
      FROM (SELECT *
            FROM trails
              EXCEPT ALL
            SELECT  trail_id as id, leg
            FROM completed
           )
      window w as (partition by id order by leg)
     )
GROUP BY id, section
HAVING max(leg) - min(leg) + 1 >= 3
ORDER BY length;
```

Exercise 4 For the graph in Figure 1, state SQL queries that answer these questions. You can use the following `with-statements` to query the graph in SQL:

```
WITH recursive singleDirection (a,b) as (
   SELECT *
   FROM (VALUES (1,2), (2,4), (1,3), (3,4), (2,5), (5,6),
                (4,6)) as graph ),
undirectedGraph as (
   SELECT *
   FROM singleDirection
  UNION ALL
   SELECT b, a
   FROM singleDirection )
```

1. Is 6 reachable from 1?

Solution:

```
WITH recursive singleDirection (a,b) as (
   SELECT *
   FROM (VALUES (1,2), (2,4), (1,3), (3,4), (2,5), (5,6),
                (4,6)) as graph ),
undirectedGraph as (
   SELECT *
   FROM singleDirection
  UNION ALL
   SELECT b, a
   FROM singleDirection ),
hull (a,b) as (
   SELECT * FROM undirectedGraph
  UNION
   SELECT fst.a, snd.b
   FROM hull fst, undirectedGraph snd
   WHERE fst.b = snd.a )
SELECT *
FROM hull
WHERE a = 6
  and b = 1;
```

2. How long is the shortest path from 1 to 6? (To end recursion, use this information: The diameter of the graph is 4.)

   Solution:

```
WITH recursive singleDirection (a, b) as (
   SELECT *
   FROM (VALUES (1,2), (2,4), (1,3), (3,4), (2,5), (5,6),
                (4,6), (6,7)) as graph ),
undirectedGraph (a,b) as (
   SELECT *
   FROM singleDirection
  UNION ALL
   SELECT b, a
   FROM singleDirection ),
hull (a, b, dist) as (
   SELECT a, b, 1
   FROM undirectedGraph
  UNION ALL
   SELECT fst.a, snd.b, dist + 1
   FROM hull fst, undirectedGraph snd
   WHERE fst.b = snd.a
     and dist <= 4 )
SELECT min(dist)
FROM hull
WHERE a = 6
  and b = 1;
```

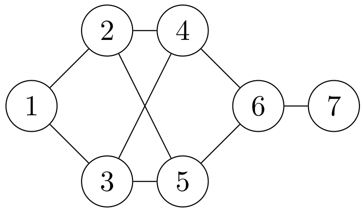

Figure 1: Example graph

Exercise 5 The Raft Consensus Protocol

To get a feeling for the *Raft consensus protocol*, you can play with the *RaftScope* at https://raft.github.io/. You can trigger events by right-clicking on servers.

1. Using the Raft visualization, simulate the following events:
   a) Wait for the first leader election.
   b) Send a request to the leader. What do you observe?
   c) Stop the leader and newly elected leaders until only two servers are left. What happens?
   d) How many servers have to fail to kill a cluster of n servers?

   Solution:

   a) Wait until the first election (e.g., of S1) is done, and all nodes reach the second term and turn blue in the visualization.

   b) You send the request by right-clicking the current leader. The leader will add the request to its log and forward it to all followers. All reachable followers will store the request in their log and send an acknowledgment message back to the leader. If the majority consents (=sends an acknowledgment), the leader will update its state machine. In the leader's next heartbeat message, the leader will convey the update to its followers. You can see this visualized in the table, where the dot and the solid lines represent the current persistent state.

   c) When the leader does not send heartbeat messages anymore, nodes will time out, and timed-out nodes will become candidates. This triggers a new leader election. Once the majority of ALL servers (also offline ones) in the system vote for the candidate, the candidate becomes the new leader. A new leader can't be elected anymore if the majority of the nodes are offline, in our example, three offline nodes.

   d) At least $\lceil n/2 \rceil$.

2. How can we be sure that a newly elected leader is holding the newest committed log entry?

Solution:

- S4 is candidate but is not holding the latest committed log entry
- S3 does not vote for S4 because its log is more ”up to date ”
- S4 cannot become a leader because the majority is more ”up to date” and thus won’t vote for S4
- Eventually, another node, e.g., S3 with the newest committed log entry, times out and becomes candidate.
- Other nodes will vote for S3 because it contains an ”up to date” version and make it the leader.
- This always works because by definition:
  - The majority of the servers are holding the last committed log entry
  - The majority of the servers are required to elect the new leader

  Two majorities of the same cluster must intersect. Eventually, a new leader with the newest committed log entry will be elected.