SQL Round

Format: Onsite
Duration: 40 Minutes
Difficulty: Medium
Domain: Analytics

Problem

Facebook's analytics team wants to understand how users stay connected among friends on their platform. The team believes that understanding patterns could help improve an algorithm that matches potential friends. Use the friends table below to address questions below. A user can perform the following sequence of actions: (1) request or receive, (2) connect, and (3) block.

Date	User_ID	Receiver_ID	Action
2020-01-01	123	234	Request
2020-01-02	123	234	Connected
2020-01-05	123	333	Request
2020-01-01	234	333	Request
...	...	...	
2020-07-03	444	234	Blocked
2020-07-04	444	123	Connected
2020-07-05	444	333	Connected

#1 - Return a list of users who blocked another user after connecting for at least 90 days. Show the user_id and receiver_id.

#2 - For each user, what is the proportion of each action? Note that the receiver_id can appear in multiple actions per user, only regard the latest status when calculating the distribution.

Files

> **Data File:** friends_connections.csv
> **Solution File:** friends_connections.txt

Solution

> **#1 -** Return a list of users who blocked another user after connecting for at least 90 days. Show the user_id and receiver_id.

[Candidate] Based on the description, can I presume that a user must first receive or request a connection and connect before he could block another user?

[Interviewer] Yes.

[Candidate] I see. If a user '123' connects with another user '234' on 01/01/2020 then blocks the user on 05/01/2020, then should the output return the following?

User_ID	Receiver_ID
123	234
....	....
....	....

[Interviewer] Yes, the count of days between connection and block dates is at least 90 so your assumption is correct.

[Candidate] Understood. Let me write the query.

```
SELECT u.user_id,
          o.receiver_id
FROM
(
        (SELECT *
                FROM connections
                WHERE action = 'Blocked') u
        LEFT JOIN
        (SELECT *
                FROM connections
                WHERE action = 'Connected') o
        ON u.user_id = o.user_id AND u.receiver_id = o.receiver_id
)
WHERE (u.dates - o.dates) >= 90;
```

[Interviewer] Can you explain your query?

[Candidate] I first create a subquery that selects blocked events. Then, I create another subquery that selects only connected events. Finally, I use LEFT JOIN on the connected table to the blocked table to create the following subquery output:

u.Date	u.User_ID	u.Receiver_ID	u.Action	o.Date	o.User_ID	o.Receiver_ID	o.Action
2020-10-01	**443**	**234**	Blocked	**2020-02-01**	**443**	**234**	Connected
2020-06-01	**554**	**234**	Blocked	**2020-02-14**	**554**	**234**	Connected
2020-05-01	**554**	**333**	Blocked	**2020-02-01**	**554**	**333**	Connected
2020-05-01	**999**	**333**	Blocked	**2020-01-01**	**999**	**333**	Connected
...	**...**	**...**	**...**	**...**	**...**	**...**	**...**

With the sub-query above, I can use WHERE clause to return pairs of users and receiver ID's, who had connected for at least 90 days.

[Interviewer] Question, is there a reason you used LEFT JOIN instead of INNER JOIN?

[Candidate] Well, in this problem, either one of the JOIN-clause works.

[Interviewer] Can you elaborate?

[Candidate] Blocking can only happen if users have connected. Given this condition, blocked events are essentially subsets of connection events. Hence, whether LEFT JOIN or INNER JOIN clause is applied, only connections that connected first then blocked are output.

Interviewer Feedback: The candidate's solution is on-point. The query is efficient and optimal. In addition, the candidate explained his solution clearly. Finally, when asked about the difference between LEFT JOIN vs. INNER JOIN, the candidate did not hesitate. So far, the candidate has demonstrated strong SQL skills.

```
# Return a pair of users who once connected for at least 90 days then blocked.
SELECT u.user_id,
        o.receiver_id
FROM
(
    # Before a user can block another user, the users must be connected. Create two
    # sub-queries - one with blocked events and another with connected events.
    # Use inner join on user_ids and receiver_ids.
    (SELECT *
        FROM connections
        WHERE action = 'Blocked') u
    JOIN
    (SELECT *
        FROM connections
        WHERE action = 'Connected') o
    ON u.user_id = o.user_id AND u.receiver_id = o.receiver_id
)
WHERE (u.dates - o.dates) >= 90;
```

> **#2 -** What is the average number of requests still pending per user? Note that the receiver_id can appear in multiple actions per user, only regard the latest status when calculating the distribution.

[Candidate] If a user accepted a connection request, then connected, the only status that matters is the latest one, right?

[Interviewer] Yes, you only need to count the most recent connection status per user.

[Candidate] I see. Suppose that a user named “123” has the following activity:

Date	User_ID	Receiver_ID	Action
2020-01-01	**123**	**234**	Request
2020-01-04	**123**	**234**	Connected
2020-01-02	**123**	**333**	Request
2020-01-10	**123**	**333**	Connected
2020-02-01	**123**	**444**	Request
2020-03-15	**123**	**444**	Connected
2020-03-29	**123**	**444**	Blocked
2020-07-01	**123**	**555**	Sent

Should I assume that the user will have the following output?

User_ID	Sent	Request	Connected	Blocked
123	0.25	0.00	0.50	0.25

[Interviewer] Precisely.

[Candidate] Okay, let me write my solution. I know that for each user_id and receiver_id pair, the connection status follows a sequence from request/sent, connected and blocked. Given this sequence, for each pair, I can order the events based on dates and assign row_number.

Date	User_ID	Receiver_ID	Action	Row_Number
2020-02-01	**123**	**444**	Request	3
2020-03-15	**123**	**444**	Connected	2
2020-03-29	**123**	**444**	Blocked	1

Slicing the row_number when the value is 1 will return the latest event per user_id and receiver_id pair.

```
WITH friendship_status AS (
        SELECT *,
                ROW_NUMBER() OVER(PARTITION BY user_id, receiver_id ORDER BY
                                                                dates DESC) AS event_order
        FROM connections
),
latest_friendship_status AS (
        SELECT *
        FROM friendship_status
        WHERE event_order = 1
)
SELECT * FROM latest_friendship_status;
```

[Interviewer] That works. Now, how would you get the proportions?

[Candidate] Well, I can create columns of dummy values for each action type. I can then apply GROUP BY on each user_id to retrieve the counts.

```
WITH friendship_status AS (
        SELECT *,
                ROW_NUMBER() OVER(PARTITION BY user_id, receiver_id ORDER BY
                                                                dates DESC) AS event_order
        FROM connections
),
latest_friendship_status AS (
        SELECT *
        FROM friendship_status
        WHERE event_order = 1
),
status_dummy_variables AS (
        SELECT *,
                CASE WHEN action = 'Sent' THEN 1 ELSE 0 END AS sent,
                CASE WHEN action = 'Received' THEN 1 ELSE 0 END AS received,
                CASE WHEN action = 'Connected' THEN 1 ELSE 0 END AS connected,
                CASE WHEN action = 'Blocked' THEN 1 ELSE 0 END AS blocked
        FROM latest_friendship_status
),
distribution AS (
        SELECT user_id,
                        SUM(sent)
                        SUM(received)
```

```
                    SUM(connected)
                    SUM(blocked)
        FROM status_dummy_variables
        GROUP BY user_id
)
SELECT * FROM distribution;
```

[Interviewer] Can you explain what you have so far?

[Candidate] The “latest_friendship_status” table generates the following:

Date	User_ID	Receiver_ID	Action	Row_Number
2020-02-01	**222**	**444**	Connected	1
2020-03-15	**123**	**333**	Connected	1
2020-03-29	**123**	**444**	Blocked	1

I then create a set of indicator columns for each action type in the action column:

Date	User_ID	Receiver_ID	Sent	Received	Connected	Blocked
2020-02-01	**222**	**444**	0	0	1	0
2020-03-15	**123**	**333**	0	0	1	0
2020-03-29	**123**	**444**	0	0	0	1

Finally, I roll-up the table at user_id then apply SUM on the indicator columns to generate the following:

User_ID	Sent	Received	Connected	Blocked
222	0	0	1	0
123	0	0	1	1

[Interviewer] Okay, now how would you get the proportion?

[Candidate] Hmm. Let me think...I need to get the total count of events somehow then, divide the count of each event from the total per user.

[Interviewer] That’s right. I just want to let you know there’s only a minute left on this round.

[Candidate] Gotcha.

[Interviewer] Time’s up.

Interviewer Feedback: In the beginning**,** the candidate asked concrete questions to ensure that he understands the problem. He also explained and illustrated his thought process clearly, which demonstrates strong technical communication skills. His solution was partially correct. His steps leading up the count distributions were correct. However, he failed to generate the proportions.

```
WITH friendship_status AS (
        SELECT *,
                ROW_NUMBER() OVER(PARTITION BY user_id, receiver_id ORDER BY
                                                                dates DESC) AS event_order
        FROM connections
),
latest_friendship_status AS (
        SELECT *
        FROM friendship_status
        WHERE event_order = 1
),
status_dummy_variables AS (
        SELECT *,
                CASE WHEN action = 'Sent' THEN 1 ELSE 0 END AS sent,
                CASE WHEN action = 'Received' THEN 1 ELSE 0 END AS received,
                CASE WHEN action = 'Connected' THEN 1 ELSE 0 END AS connected,
                CASE WHEN action = 'Blocked' THEN 1 ELSE 0 END AS blocked
        FROM latest_friendship_status
),
distribution AS (
        SELECT user_id,
                        SUM(sent) / SUM(event_order) AS sent,
                        SUM(received) / SUM(event_order) AS sent,
                        SUM(connected) / SUM(event_order) AS sent,
                        SUM(blocked) / SUM(event_order) AS sent,
        FROM status_dummy_variables
        GROUP BY user_id
)
SELECT * FROM distribution;
```

Final Assessment

In the SQL section, the candidate is assessed based on correctness, SQL efficiency, and communication. For each dimension the candidate is rated in the following scale: (5) superior, (4) good, (3) adequate, (2) marginal, (1) not competent.

Assessments	Rating	Comments
SQL Correctness	4	The candidate aced the first problem. Each step of his solution was logical and ultimately produced the correct solution. In the second problem, however, the candidate was partially correct. All the steps excluding the last part, computing the distributions, was correct. The candidate was able to create the count distribution but failed to generate the proportions. Hence, the candidate receives a score of 4.
SQL Efficiency	5	Both queries were efficient. The candidate did not use unnecessary subqueries or JOINS, let alone CROSS JOINS which is computationally intensive.
Communication	5	The candidate demonstrated strong communication skills. He asked questions to clarify the problems before diving into solutions. During the brainstorming step, he elaborated his thought process. After writing the final solution, he illustrated clearly how each step of the query works.