SQL Questions Asked in Facebook

— PRAGYA RATHI

---

1. Consecutive Error Count for Users

- **Question**: Write a query to find all users who encountered 3 consecutive errors in a log table.
- **Table**: `logs` with columns `user_id`, `log_time`, `status`.
- **Hint**: Use the `LAG` or `LEAD` function to compare statuses of consecutive logs for each user.

```
SELECT user_id

FROM (

   SELECT user_id,

        status,

        LAG(status, 1) OVER (PARTITION BY user_id ORDER BY log_time) AS prev_status1,

        LAG(status, 2) OVER (PARTITION BY user_id ORDER BY log_time) AS prev_status2

   FROM logs

) AS t

WHERE status = 'error'

AND prev_status1 = 'error'

AND prev_status2 = 'error';
```

---

2. Find Users with Max Revenue Per Category

- **Question**: For each product category, identify the user who generated the maximum total revenue.
- **Table**: `transactions` with columns `user_id`, `product_category`, `revenue`.
- **Hint**: Group by `product_category` and use `ROW_NUMBER()` with `PARTITION BY`.

```
WITH RevenueByUser AS (
```

```
    SELECT user_id, product_category, SUM(revenue) AS total_revenue
    FROM transactions
    GROUP BY user_id, product_category
)
SELECT user_id, product_category, total_revenue
FROM (
    SELECT user_id, product_category, total_revenue,
        ROW_NUMBER() OVER (PARTITION BY product_category ORDER BY total_revenue
DESC) AS rank
    FROM RevenueByUser
) AS ranked
WHERE rank = 1;
```

---

3. Product Popularity Over Time

- **Question**: Calculate the rolling 7-day average of product views per product.
- **Table**: `views` with columns `product_id`, `view_time`, `user_id`.
- **Hint**: Use `WINDOW` functions and rolling averages.

```
SELECT product_id, view_time,
    AVG(view_count) OVER (PARTITION BY product_id ORDER BY view_time ROWS
BETWEEN 6 PRECEDING AND CURRENT ROW) AS rolling_avg
FROM (
    SELECT product_id, view_time, COUNT(user_id) AS view_count
    FROM views
    GROUP BY product_id, view_time
) AS product_views;
```

---

4. Finding Gaps in Order IDs

- **Question**: Given a table of order IDs, identify any gaps in sequential order numbers.
- **Table**: `orders` with columns `order_id`.
- **Hint**: Use `LAG()` to compare the current order with the previous order.

```
SELECT order_id, LAG(order_id, 1) OVER (ORDER BY order_id) AS previous_order
 FROM orders
WHERE order_id - LAG(order_id, 1) OVER (ORDER BY order_id) > 1;
```

---

5. Detect Abandoned Carts

- **Question**: Find all users who added items to their cart but didn't complete a purchase within the last 24 hours.
- **Tables**: `cart_actions` (with `user_id`, `action`, `action_time`), `purchases` (with `user_id`, `purchase_time`).
- **Hint**: Use a `LEFT JOIN` to find users with cart actions but no corresponding purchases.

```
SELECT c.user_id
FROM cart_actions c
LEFT JOIN purchases p
ON c.user_id = p.user_id
AND p.purchase_time > c.action_time
AND p.purchase_time <= c.action_time + INTERVAL '24 hours'
WHERE c.action = 'add_to_cart'
AND p.user_id IS NULL;
```

---

6. Average Session Duration Per User

- **Question**: Calculate the average session duration for each user. A session is defined as a continuous series of events where no event is more than 30 minutes apart.
- **Table**: `events` with columns `user_id`, `event_time`.
- **Hint**: Use `LAG()` to calculate time gaps between events and identify session boundaries.

```
WITH EventGaps AS (
    SELECT user_id, event_time,
        LAG(event_time) OVER (PARTITION BY user_id ORDER BY event_time) AS
prev_event_time
    FROM events
),
SessionData AS (
    SELECT user_id, event_time,
        CASE
            WHEN event_time - prev_event_time > INTERVAL '30 minutes' OR prev_event_time
IS NULL
            THEN 1 ELSE 0
        END AS new_session
    FROM EventGaps
)
SELECT user_id, AVG(session_duration) AS avg_session_duration
FROM (
    SELECT user_id,
        SUM(EXTRACT(EPOCH FROM (LEAD(event_time) OVER (PARTITION BY user_id
ORDER BY event_time) - event_time))) AS session_duration
```

```
    FROM SessionData
    WHERE new_session = 0
    GROUP BY user_id
) AS sessions
GROUP BY user_id;
```

---

7. Find Users with Increasing Purchase Amounts

- **Question**: Find all users whose total purchase amount has increased with each transaction.
- **Table**: `purchases` with columns `user_id`, `purchase_amount`, `purchase_time`.
- **Hint**: Use `LAG()` or `LEAD()` to compare the purchase amounts of consecutive transactions.

```
SELECT user_id
FROM (
    SELECT user_id, purchase_amount,
        LAG(purchase_amount) OVER (PARTITION BY user_id ORDER BY purchase_time) AS prev_purchase
    FROM purchases
) AS t
WHERE purchase_amount > prev_purchase;
```

---

8. Total Watch Time for Each User

- **Question**: Calculate the total watch time for each user based on video start and end events.
- **Tables**: `video_start` and `video_end` with columns `user_id`, `video_id`, `event_time`.
- **Hint**: Use a `JOIN` on `user_id` and `video_id`, and calculate the time difference between start and end events.

```
SELECT vs.user_id, vs.video_id,
    SUM(EXTRACT(EPOCH FROM (ve.event_time - vs.event_time))) AS total_watch_time
FROM video_start vs
JOIN video_end ve
ON vs.user_id = ve.user_id
AND vs.video_id = ve.video_id
GROUP BY vs.user_id, vs.video_id;
```

---

9. Find Products That Have Never Been Purchased

- **Question**: Write a query to find all products that have never been purchased.
- **Tables**: `products` with `product_id`, `purchases` with `product_id`.
- **Hint**: Use a `LEFT JOIN` and filter for `NULL` values in the `purchases` table.

```
SELECT p.product_id
FROM products p
LEFT JOIN purchases pu
ON p.product_id = pu.product_id
WHERE pu.product_id IS NULL;
```

---

10. Returning Users with Multiple Failed Login Attempts

- **Question**: Identify users who had more than 3 failed login attempts within an hour but eventually logged in successfully.

- **Table**: `logins` with columns `user_id`, `login_time`, `status` (either "success" or "fail").
- **Hint**: Use `WINDOW` functions like `COUNT()` with `PARTITION BY` to count failed attempts.

```
WITH FailedAttempts AS (
    SELECT user_id, login_time,
           COUNT(*) OVER (PARTITION BY user_id ORDER BY login_time RANGE INTERVAL '1
hour' PRECEDING) AS fail_count
    FROM logins
    WHERE status = 'fail'
),
SuccessfulLogin AS (
    SELECT DISTINCT user_id
    FROM logins
    WHERE status = 'success'
)
SELECT fa.user_id
FROM FailedAttempts fa
JOIN SuccessfulLogin sl
ON fa.user_id = sl.user_id
WHERE fa.fail_count > 3;
```