🧠 SQL Challenge: Difference Between Latest and Second Latest Event Values

📝 Problem Statement

You are given a table named `events` with the structure below:

```
CREATE TABLE events (
    event_type INTEGER NOT NULL,
    value INTEGER NOT NULL,
    time TIMESTAMP NOT NULL,
    UNIQUE(event_type, time)
);
```

Goal:

For each `event_type` that appears **more than once**, return a result that shows:

- The `event_type`
- The **difference** between the latest and second latest `value`, ordered by `time`.

📊 Example Input:

event_type	value	time
2	5	2015-05-09 12:42:00
4	-42	2015-05-09 13:19:57
2	2	2015-05-09 14:48:30
2	7	2015-05-09 12:54:39
3	16	2015-05-09 13:19:57
3	20	2015-05-09 15:01:09

📈 Expected Output:

event_type	value
2	-5
3	4

💡 SQL Solution:

```
WITH ranked_events AS (
    SELECT
        event_type,
        value,
        ROW_NUMBER() OVER (PARTITION BY event_type ORDER BY time DESC) AS rn
    FROM events
),
filtered_events AS (
```

```
    SELECT * FROM ranked_events
    WHERE rn <= 2
)
SELECT
    event_type,
    MAX(CASE WHEN rn = 1 THEN value END) -
    MAX(CASE WHEN rn = 2 THEN value END) AS value
FROM filtered_events
GROUP BY event_type
HAVING COUNT(*) = 2
ORDER BY event_type;
```

Understanding the Problem

We need to:

1. Find event types that appear more than once in the table
2. Calculate the difference between the latest value and the second latest value for each event type
3. The events are ordered by timestamp (time column)
4. Return the event type and the calculated difference

Breaking Down the SQL Query

The solution uses Common Table Expressions (CTEs) and window functions. Let's break it down:

```
WITH ranked_events AS (
    SELECT
        event_type,
        value,
        ROW_NUMBER() OVER (PARTITION BY event_type ORDER BY time DESC) AS rn
    FROM events
),
filtered_events AS (
    SELECT * FROM ranked_events
    WHERE rn <= 2
)
SELECT
    event_type,
    MAX(CASE WHEN rn = 1 THEN value END) -
    MAX(CASE WHEN rn = 2 THEN value END) AS value
FROM filtered_events
GROUP BY event_type
HAVING COUNT(*) = 2
ORDER BY event_type;
```

1. First CTE: Ranking Events by Time

```
WITH ranked_events AS (
    SELECT
        event_type,
```

```
        value,
        ROW_NUMBER() OVER (PARTITION BY event_type ORDER BY time DESC) AS rn
    FROM events
)
```

This first step:

- Creates a CTE named `ranked_events`
- Uses the window function `ROW_NUMBER()` to assign a rank to each event
- `PARTITION BY event_type`: Groups the rows by event_type
- `ORDER BY time DESC`: Orders each group by time in descending order (newest first)
- The result is that for each event_type, the most recent event gets rank 1, the second most recent gets rank 2, etc.

2. Second CTE: Filtering to Latest Two Events

```
filtered_events AS (
    SELECT * FROM ranked_events
    WHERE rn <= 2
)
```

This step:

- Creates a second CTE named `filtered_events`
- Simply filters the previous CTE to keep only events with rank 1 or 2
- This means we're only keeping the latest and second latest events for each event_type

3. Final Query: Calculating Differences

```
SELECT
    event_type,
    MAX(CASE WHEN rn = 1 THEN value END) -
    MAX(CASE WHEN rn = 2 THEN value END) AS value
FROM filtered_events
GROUP BY event_type
HAVING COUNT(*) = 2
ORDER BY event_type;
```

The final query:

- Groups by `event_type`
- Uses `CASE` statements to identify the latest value (rn = 1) and the second latest value (rn = 2)
- Calculates the difference between these values
- The `HAVING COUNT(*) = 2` ensures we only include event types that have at least two events
- Orders the results by `event_type`

Walking Through an Example

Let's use the example data provided:

event_type	value	time
2	5	2015-05-09 12:42:00
4	-42	2015-05-09 13:19:57
2	2	2015-05-09 14:48:30
2	7	2015-05-09 12:54:39
3	16	2015-05-09 13:19:57
3	20	2015-05-09 15:01:09

Step 1: Apply ranked_events CTE

After the first CTE, we would have:

event_type	value	time	rn
2	2	2015-05-09 14:48:30	1
2	7	2015-05-09 12:54:39	2
2	5	2015-05-09 12:42:00	3
3	20	2015-05-09 15:01:09	1
3	16	2015-05-09 13:19:57	2
4	-42	2015-05-09 13:19:57	1

Note: The rows are ordered by event_type and then by time in descending order, with rn assigned accordingly.

Step 2: Apply filtered_events CTE

After filtering to keep only rows where rn ≤ 2:

event_type	value	time	rn
2	2	2015-05-09 14:48:30	1
2	7	2015-05-09 12:54:39	2
3	20	2015-05-09 15:01:09	1
3	16	2015-05-09 13:19:57	2
4	-42	2015-05-09 13:19:57	1

Step 3: Final calculation

Now we group by event_type and calculate the differences:

For event_type 2:

- Latest value (rn = 1): 2
- Second latest value (rn = 2): 7
- Difference: 2 - 7 = -5

For event_type 3:

- Latest value (rn = 1): 20
- Second latest value (rn = 2): 16
- Difference: 20 - 16 = 4

For event_type 4:

- We only have one record (only rn = 1)
- This gets filtered out by `HAVING COUNT(*) = 2`

Final result:

event_type	value
2	-5
3	4

Key Concepts for Beginners

1. **Window Functions**: `ROW_NUMBER()` assigns a sequential integer to rows within a partition
2. **Common Table Expressions (CTEs)**: Temporary named result sets that exist for a single query
3. **CASE Expressions**: Conditional logic within a SQL query
4. **HAVING Clause**: Filters groups, whereas WHERE filters individual rows
5. **MAX with CASE**: A technique to extract specific values based on conditions

Alternative Approaches

For simpler SQL dialects that don't support window functions, you could use:

```
SELECT
    e1.event_type,
    e1.value - e2.value AS value
FROM events e1
JOIN (
    SELECT
        event_type,
        value,
        time,
        (SELECT COUNT(*) FROM events e3 WHERE e3.event_type = e2.event_type AND
e3.time > e2.time) AS later_count
    FROM events e2
) AS latest_events
ON e1.event_type = latest_events.event_type
WHERE latest_events.later_count = 0
AND (
    SELECT COUNT(*) FROM events e4
    WHERE e4.event_type = e1.event_type
) > 1
ORDER BY e1.event_type;
```

But the window function approach is generally more readable and often more efficient.

This problem demonstrates a common pattern in SQL: finding differences between consecutive records within groups, which is useful for tracking changes over time in various data analysis scenarios.