SQL Challenge: Games Played So Far

Problem Statement

You are given a table `Activity` that tracks how many games a player played on specific dates and devices. Your task is to generate a report that, for each player and each event date, shows the **cumulative number of games played up to and including that date**.

Input Table: `Activity`

player_id	device_id	event_date	games_played
1	2	2016-03-01	5
1	2	2016-05-02	6
1	3	2017-06-25	1
3	1	2016-03-02	0
3	4	2018-07-03	5

Expected Output

player_id	event_date	games_played_so_far
1	2016-03-01	5
1	2016-05-02	11
1	2017-06-25	12
3	2016-03-02	0
3	2018-07-03	5

SQL Solution

```sql
SELECT
    t1.player_id,
    t1.event_date,
    SUM(t2.games_played) AS games_played_so_far
FROM Activity t1
JOIN Activity t2
    ON t1.player_id = t2.player_id
    AND t1.event_date >= t2.event_date
GROUP BY
    t1.player_id,
    t1.event_date
ORDER BY
    t1.player_id,
    t1.event_date;
```

Understanding the Problem

We need to:

1. Track the total number of games each player has played up to and including each date
2. For each player and date, we need a running sum of games played
3. Return player ID, event date, and the cumulative sum of games played

Breaking Down the Solution

Let's analyze the SQL query that solves this problem:

```
SELECT
    t1.player_id,
    t1.event_date,
    SUM(t2.games_played) AS games_played_so_far
FROM Activity t1
JOIN Activity t2
    ON t1.player_id = t2.player_id
    AND t1.event_date >= t2.event_date
GROUP BY
    t1.player_id,
    t1.event_date
ORDER BY
    t1.player_id,
    t1.event_date;
```

Step 1: Self-Join Approach for Cumulative Sum

The key technique used here is a self-join of the Activity table. We're joining the table with itself to find all rows that:

1. Have the same player ID
2. Have event dates that are earlier than or equal to each date we're calculating for

```
FROM Activity t1
JOIN Activity t2
    ON t1.player_id = t2.player_id
    AND t1.event_date >= t2.event_date
```

Let's see what this join produces with our sample data:

For player_id = 1 and event_date = 2016-03-01:

- Joins with itself (2016-03-01) only

For player_id = 1 and event_date = 2016-05-02:

- Joins with 2016-03-01 and itself (2016-05-02)

For player_id = 1 and event_date = 2017-06-25:

- Joins with 2016-03-01, 2016-05-02, and itself (2017-06-25)

For player_id = 3 and event_date = 2016-03-02:

- Joins with itself (2016-03-02) only

For player_id = 3 and event_date = 2018-07-03:

- Joins with 2016-03-02 and itself (2018-07-03)

After this join, we'd have:

t1.player_id	t1.event_date	t1.games_played	t2.player_id	t2.event_date	t2
1	2016-03-01	5	1	2016-03-01	5
1	2016-05-02	6	1	2016-03-01	5
1	2016-05-02	6	1	2016-05-02	6
1	2017-06-25	1	1	2016-03-01	5
1	2017-06-25	1	1	2016-05-02	6
1	2017-06-25	1	1	2017-06-25	1
3	2016-03-02	0	3	2016-03-02	0
3	2018-07-03	5	3	2016-03-02	0
3	2018-07-03	5	3	2018-07-03	5

Step 2: Calculating the Running Sum

After joining, we need to sum up all the games played in the matching rows for each player and date combination:

```
SELECT
    t1.player_id,
    t1.event_date,
    SUM(t2.games_played) AS games_played_so_far
```

Let's calculate this sum for each group:

For player_id = 1, event_date = 2016-03-01:

- Sum of games_played: 5 = 5

For player_id = 1, event_date = 2016-05-02:

- Sum of games_played: 5 + 6 = 11

For player_id = 1, event_date = 2017-06-25:

- Sum of games_played: 5 + 6 + 1 = 12

For player_id = 3, event_date = 2016-03-02:

- Sum of games_played: 0 = 0

For player_id = 3, event_date = 2018-07-03:

- Sum of games_played: 0 + 5 = 5

Step 3: Grouping and Ordering

We need to group by player and event date to get one sum per combination:

```
GROUP BY
    t1.player_id,
    t1.event_date
ORDER BY
    t1.player_id,
    t1.event_date;
```

This ensures we have one row per player per date, and that the results are ordered by player ID and then by date (chronologically).

Step 4: Final Result

The final output matches our expected result:

player_id	event_date	games_played_so_far
1	2016-03-01	5
1	2016-05-02	11
1	2017-06-25	12
3	2016-03-02	0
3	2018-07-03	5

Why This Works (Detailed Explanation)

The self-join approach is a classic technique for calculating running totals or cumulative sums in SQL. Here's why it works:

1. **Self-Join with Date Comparison**: By joining a table with itself where dates in the second table are less than or equal to dates in the first, we create rows for all historical data points.
2. **Same Player Filter**: The `t1.player_id = t2.player_id` condition ensures we only sum games for the same player.
3. **SUM Aggregation**: For each player-date combination, we sum all games played up to and including that date.
4. **Conceptual Model**: Think of this as looking at each date and "looking backward" to include all previous activity for that player.

Alternative Modern Approaches

In more recent SQL versions, you could use window functions to achieve the same result more efficiently:

```
SELECT
    player_id,
```

```
    event_date,
    SUM(games_played) OVER (
        PARTITION BY player_id
        ORDER BY event_date
        ROWS UNBOUNDED PRECEDING
    ) AS games_played_so_far
FROM Activity
ORDER BY player_id, event_date;
```

This approach:

1. Partitions the data by player_id (separating each player's records)
2. Orders by event_date within each partition
3. Sums all rows from the beginning of the partition up to the current row

The window function approach is generally more efficient and readable than the self-join method, but both produce the same result. The self-join method works in older SQL versions that don't support window functions.