How to Find the Row with the Maximum Value in Each Group in MySQL

Authored by

mohammed looti

January 6, 2026

The task of selecting the complete row associated with the maximum value within defined groups is a common and critical challenge in SQL, often referred to as the "greatest-n-per-group" problem. While naive attempts might involve using the aggregate MAX() function alongside the GROUP BY clause, these standard aggregate methods typically only return the maximum value itself, not the related data from the entire row (like the athlete's ID or other descriptive details). To reliably retrieve the complete row corresponding to the highest value in each group within MySQL, the most robust and widely compatible technique involves using a specific type of subquery known as a correlated subquery. This method compares each row in the main table against the maximum value found in its specific group defined by the grouping column, ensuring accuracy and integrity of the full record.

Implementing the Correlated Subquery for Max-Value Selection

The standard approach to solving this problem in MySQL (and many other relational database systems) requires leveraging a subquery within the `WHERE` clause. This subquery calculates the maximum value for the grouping criteria, and the outer query selects only those rows that successfully match that calculated maximum. This structure ensures that we retrieve the complete record that holds the highest metric within its respective category. The key is establishing a correlation between the outer query (aliased as `a1`) and the inner query (aliased as `a2`) based on the grouping column. This mechanism bypasses the inherent limitations of standard aggregation when trying to select non-aggregated fields alongside grouped results.

The following syntax illustrates how to select all columns (`*`) for the row containing the maximum value in the **points** column, grouped dynamically by the **team** column. Note that the inner query executes once for every row processed by the outer query, making it an extremely precise way to filter results based on group aggregates without running into typical GROUP BY clause complexities when selecting non-aggregated columns. This self-referencing pattern is fundamental to accurately solving the greatest-n-per-group problem where n=1.

```
SELECT *
FROM athletes a1
WHERE points=(SELECT MAX(a2.points)
FROM athletes a2
WHERE a1.team = a2.team)
```

In this specific implementation, we are selecting rows from the `athletes` table where the value in the `points` column is equal to the maximum `points` value found among all records sharing the same `team`. The critical line is `WHERE a1.team = a2.team`, which establishes the necessary correlation, ensuring that the inner query only aggregates data relevant to the current team being

evaluated by the outer query. This powerful technique provides an accurate and complete solution to the max-by-group requirement.

Dissecting the Correlated Subquery Logic

Understanding precisely how the correlated subquery operates is essential for mastering advanced SQL filtering techniques. Unlike a standard, non-correlated subquery that executes only once before the outer query runs, a correlated subquery relies on values from the outer query to define its scope. The process can be visualized as an iterative check against the data. For every row (aliased as `a1`) evaluated by the primary `SELECT` statement, the nested subquery (referencing the table as `a2`) must execute again.

During each execution of the inner subquery, it performs an aggregation--in this case, calculating the MAX() function on the `points` column--but this calculation is critically restricted to the subset of records where the `team` column matches the current team being examined in the outer query (enforced by the condition `a1.team = a2.team`). The result of this inner query is a single, scalar maximum value specific to that team.

The outer query then uses this single maximum value to filter the original row (`a1`). If `a1.points` is exactly equal to the maximum value returned by the subquery for that team, the row is included in the final result set. This robust mechanism guarantees that only the records achieving the highest score within their predetermined group are returned, providing the full details of the record, not just the aggregate number. This detail-oriented approach is why the correlated subquery is favored for compatibility and precision.

Practical Demonstration: Setting Up the Dataset

To solidify this concept, let us work with a concrete dataset. Suppose we have a table named **athletes** that meticulously tracks performance data, specifically the points scored by various players across different teams. Our objective is to retrieve the record of the highest-scoring athlete from each team represented in this table. This scenario perfectly highlights the utility of the correlated subquery method in MySQL.

We begin by defining and populating our sample table using standard Data Definition Language (DDL) and Data Manipulation Language (DML) commands. The structure includes an `id` (acting as the primary key), `team` (the essential grouping column), and `points` (the metric we wish to maximize within each defined group). The data represents scores from ten different players across five distinct teams.

```
-- create table
CREATE TABLE athletes (
```

```
id INT PRIMARY KEY,
team TEXT NOT NULL,
points INT NOT NULL
);

-- insert rows into table
INSERT INTO athletes VALUES (0001, 'Mavs', 22);
INSERT INTO athletes VALUES (0002, 'Mavs', 14);
INSERT INTO athletes VALUES (0003, 'Lakers', 37);
INSERT INTO athletes VALUES (0004, 'Knicks', 19);
INSERT INTO athletes VALUES (0005, 'Knicks', 26);
INSERT INTO athletes VALUES (0006, 'Knicks', 40);
INSERT INTO athletes VALUES (0007, 'Lakers', 21);
INSERT INTO athletes VALUES (0008, 'Celtics', 15);
INSERT INTO athletes VALUES (0009, 'Hawks', 18);
INSERT INTO athletes VALUES (0010, 'Celtics', 23);

-- view all rows in table
SELECT * FROM athletes;
```

Reviewing the Initial Dataset

Before proceeding with the filtering query, it is crucial to examine the full content of the **athletes** table. This step helps confirm the groups present and verify the maximum values we expect to retrieve later. Our sample dataset includes groups for Mavs, Lakers, Knicks, Celtics, and Hawks, with varying player counts per team.

The initial output clearly shows the distribution of players and their scores. We can observe that the Knicks have the highest player count (three), while other teams like the Lakers and Mavs have two entries each. Our primary task is to isolate the record corresponding to the highest individual score achieved by a player belonging to each of these unique teams, effectively reducing the table to one row per team.

Output of the full table:

```
+----+---------+--------+
| id | team | points |
+----+---------+--------+
| 1 | Mavs | 22 |
| 2 | Mavs | 14 |
```

| 3 | Lakers | 37 |
| 4 | Knicks | 19 |
| 5 | Knicks | 26 |
| 6 | Knicks | 40 |
| 7 | Lakers | 21 |
| 8 | Celtics | 15 |
| 9 | Hawks | 18 |
| 10 | Celtics | 23 |
+----+---------+--------+

Based on a manual inspection, the expected maximum scores per team are: Mavs (22), Lakers (37), Knicks (40), Celtics (23), and Hawks (18). Our subsequent query must successfully retrieve the entire row--including the player ID--associated with these maximum scores.

Executing the Max-by-Group Query

We now apply the correlated subquery structure developed in the previous sections to solve the requirement: selecting the rows with the maximum **points** value for each **team**. This query efficiently filters the `athletes` table, retaining only those rows that meet the condition of being the top scorer for their respective team.

The query is concise, leveraging table aliases (`a1` and `a2`) to distinguish between the instance of the table used in the outer select and the instance used within the inner aggregate calculation. This use of self-joining and correlation is what separates this robust solution from simpler but often inaccurate standard grouping methods. It ensures that the comparison is made row-by-row against the team's maximum score.

```
SELECT *
FROM athletes a1
WHERE points=(SELECT MAX(a2.points)
FROM athletes a2
WHERE a1.team = a2.team)
```

The successful execution of this statement yields the precise result set we aimed for, confirming that the correlated subquery method effectively isolates the desired records based on the condition that the row's point total matches the maximum point total for its team.

Analyzing the Final Query Results

The resulting output confirms that for every group defined by the **team** column, only the complete

row associated with the highest **points** value is returned. This conclusive result demonstrates the efficiency and accuracy of using the correlated subquery for this specific grouping problem in MySQL, providing the full context of the top performer.

Output of the filtered query:

```
+----+---------+--------+
| id | team | points |
+----+---------+--------+
| 1 | Mavs | 22 |
| 3 | Lakers | 37 |
| 6 | Knicks | 40 |
| 9 | Hawks | 18 |
| 10 | Celtics | 23 |
+----+---------+--------+
```

As demonstrated by the results, the filtering worked perfectly across all groups. For example, the Knicks team originally had scores of 19, 26, and 40. Only the row corresponding to the score of **40** (ID 6) is present in the final result set. Similarly, for the Celtics, where scores were 15 and 23, only the row with **23** (ID 10) is returned. This conclusive output verifies that the correlated subquery effectively achieves the greatest-n-per-group objective for n=1 by comparing individual row values against the dynamic maximum of their respective group.

Summary of Solution Methods

When aiming to select the full row associated with the maximum value per group in SQL, the correlated subquery remains the most reliable and backward-compatible method, particularly for older versions of MySQL. However, modern database platforms, including MySQL 8.0+, offer alternative solutions that can sometimes be more performant and easier to read.

For users on modern systems, the use of Window Functions, such as `ROW_NUMBER()`, provides an excellent declarative alternative. This function partitions the data by the grouping column (e.g., `team`) and then assigns a sequential rank based on the maximizing column (e.g., `points` descending). An outer query simply filters for the row where the assigned rank is 1. While highly efficient, this method is only available in recent versions.

Key takeaways for ensuring success using the compatible correlated subquery method include:

Self-Join Necessity: The query must alias and join the table to itself (using aliases like `a1` and `a2`) to allow simultaneous comparison of individual rows against group aggregates calculated within the inner query.

Correlation: The conditional link (e.g., `WHERE a1.team = a2.team`) is absolutely essential. This correlation ensures that the inner subquery calculation of the MAX() function is restricted precisely to the group currently being evaluated by the outer query, preventing erroneous global maximum calculations.

Performance Consideration: While effective, correlated subqueries can sometimes be slower than alternative methods (like window functions) on very large datasets due to the iterative nature of their execution. However, they remain the standard for maximum compatibility.

By mastering the correlated subquery, developers can reliably solve complex data retrieval problems that go beyond simple aggregation, ensuring that the entire context of the maximal data point is preserved and accurately returned in SQL.