Learning MySQL: Retrieving Rows with Maximum Values by Group

Authored by

Mohammed looti

November 12, 2025

Understanding the "Max Value Per Group" Challenge in SQL

One of the most essential and frequently encountered analytical problems when managing data in a relational database environment, such as MySQL, is efficiently retrieving the complete row associated with the maximum or minimum value within a defined group. This common scenario is often termed the **"Greatest-N-Per-Group"** challenge. For instance, a data analyst might need to identify the highest recorded transaction for every store location, or, as we will explore in detail, determine the single highest score achieved by a player on each distinct team. Attempting to solve this only using the standard SQL aggregate function `MAX()` proves insufficient, as it only returns the maximum value itself, failing to retrieve the crucial associated metadata, such as the athlete's name, ID, or other descriptive fields, necessary for a complete record.

Effectively tackling this challenge requires specialized querying techniques that go beyond simple aggregation. The goal is to compare the value of a specific column in a row against the maximum value of that same column, but only within the subset of rows belonging to the same group. This dynamic comparison allows us to filter out all records that are not the "winner" of their respective group. Historically, the most traditional, reliable, and widely supported method across various SQL implementations has been the utilization of **correlated subqueries**. This approach establishes a tight link between the inner and outer queries, enabling precise, row-by-row maximum calculation based on the grouping criteria defined in the outer statement.

Method 1: Solving with Correlated Subqueries

The mechanism of a correlated subquery is remarkably powerful for solving grouped maximum problems because it relies on values retrieved from the outer query to execute the inner query. In our standard implementation, the outer query examines the main table (aliased as `a1`), while the inner query runs against the same table (aliased as `a2`). This inner query calculates the maximum value for a specific column, but crucially, it is filtered by the `WHERE` clause to match the current group being processed by the outer query (e.g., matching `a1.team = a2.team`). The outer query then uses this calculated group maximum in its own `WHERE` clause to filter the original records, retaining only the rows where the score (`points`) matches the highest score calculated for that specific group.

The following syntax illustrates the highly portable and standard method in MySQL for selecting the entire row associated with the maximum value in a designated column, grouped by another categorical column. This pattern remains a cornerstone of database querying for extracting summarized yet detailed record information, ensuring compatibility even with older database versions.

SELECT *

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

Analyzing this query, the core logic is designed to pinpoint the record where a player's `points` score is precisely equal to the maximum score achieved by any player affiliated with their respective `team`. The outer query selects all available columns (`SELECT *`) from the `athletes` table (aliased as `a1`). The crucial filtering mechanism resides in the `WHERE` clause: for every row processed by `a1`, the inner subquery dynamically calculates the `MAX(points)` from the `athletes` table (aliased as `a2`). The correlation condition, `a1.team = a2.team`, guarantees that this maximum calculation is restricted solely to the records belonging to the team currently being evaluated by `a1`. This mechanism ensures we successfully isolate the single top performer for each distinct group defined by the `team` column.

Practical Implementation: Setting Up the Sample Data

To provide a concrete example and allow for rigorous testing of our correlated subquery solution, we must first establish a functional sample table. We will create a table named `athletes` that simulates tracking individual player performance across multiple teams. This table structure includes a unique identifier (ID), the team name (which serves as our grouping column), and the points scored (the value column we intend to maximize within each group). Setting up realistic and varied data is absolutely fundamental for confirming the validity and robustness of any complex SQL solution.

The `athletes` table incorporates three essential columns: `id` (the primary key), `team`, and `points`. The following sequence of Data Definition Language (DDL) and Data Manipulation Language (DML) commands executes the table creation and populates it with diverse performance data. We have deliberately included multiple entries for teams like 'Mavs' and 'Knicks' to ensure our grouping logic is thoroughly tested and verified against competing scores within the same category.

```
-- 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;
```

Executing the final `SELECT *` command allows us to view the complete, unfiltered dataset, confirming that we have established the necessary groups and varying point values across the 10 records. This initial view is an essential baseline against which we can compare and verify the results generated by our maximum-per-group query, ensuring that the filtering process works exactly as intended.

Output:

```
+----+---------+--------+
| 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 |
+----+---------+--------+
```

Executing the Correlated Subquery Solution

With the sample data successfully loaded, our primary task is to execute the query that selects

only the single row corresponding to the maximum **points** value achieved for each distinct **team**. This involves applying the robust correlated subquery pattern introduced earlier directly to the populated `athletes` table. The reliability and universal compatibility of this method, particularly for users operating on older versions of MySQL that may lack newer analytical capabilities, make it a fundamental skill for any professional working with relational data.

We utilize the following SQL syntax, which systematically checks every athlete's score against the highest score recorded for their specific team through the inner query. If the athlete's individual score perfectly matches the team's maximum score, that entire row is retained and included in the final result set. This process effectively and definitively solves the greatest-per-group problem using a standard, cross-compatible technique.

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

Upon successful execution of the query against our sample data, the output confirms the effectiveness of the correlated subquery in accurately solving the greatest-per-group challenge. The result set is dramatically filtered, returning exactly one row per team, and critically, that row is guaranteed to contain the highest score recorded for that specific team, along with all its associated metadata (ID, etc.).

Output:

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

By reviewing the filtered results, we can clearly observe the successful isolation of the maximum value. For example, the 'Knicks' team originally contained three scores: **19**, **26**, and **40**. The query successfully discarded the two lower scores, returning only the single record associated with the maximum value of **40**. Likewise, the 'Mavs' team had scores of **22** and **14**, yet only the row

corresponding to the highest score of **22** was retained. This outcome verifies that the logic correctly identified the row with the maximum value in the **points** column for every distinct **team** in the dataset.

Alternative Approaches: Leveraging Window Functions

While the correlated subquery method provides excellent compatibility, modern SQL standards, particularly those implemented in MySQL 8.0 and subsequent versions, advocate for the use of Window Functions. Functions such as `ROW_NUMBER()`, `RANK()`, or `DENSE_RANK()` offer a syntactically cleaner and often significantly more performant way to partition data and apply analytical calculations. Unlike aggregate functions, Window Functions calculate values without collapsing the rows, making subsequent filtering based on the calculated rank straightforward.

Implementing a Window Function solution typically involves defining a partition (which is equivalent to our grouping column, `team`) and specifying the ordering within that partition (e.g., ordering by `points` in descending order). The function then assigns a unique rank or row number to each record within its respective partition. Finally, an outer query or a Common Table Expression (CTE) is used to select only those rows where the assigned rank (e.g., `rn`) is 1. This method avoids the performance drawback associated with forcing the database engine to execute a separate, correlated subquery for every single row in the main table, resulting in notable speed improvements on very large datasets.

The conceptual query below demonstrates how `ROW_NUMBER()` is used to achieve the same result as the correlated subquery, but using the modern analytical syntax:

```
SELECT id, team, points
FROM (
SELECT *,
ROW_NUMBER() OVER (PARTITION BY team ORDER BY points DESC) AS rn
FROM athletes
) ranked_athletes
WHERE rn = 1;
```

Considerations and Performance Implications

The choice between utilizing a correlated subquery and a Window Function hinges primarily on two factors: database version compatibility and query performance at scale. The correlated subquery is highly compatible, making it the only viable solution for environments running older versions of MySQL or other legacy database systems that do not support modern analytic functions. However, the execution model, which necessitates recalculating the maximum value repeatedly (once for

every row processed by the outer query), can introduce significant performance bottlenecks when querying tables that contain millions of records.

In contrast, the Window Function approach is engineered for efficiency by the database engine. It generally requires only a single pass (or a minimal number of passes) over the data to simultaneously calculate the necessary ranking across all groups. This optimization translates directly to faster execution times. Therefore, if your operating environment supports MySQL version 8.0 or newer, using a ranking function like `ROW_NUMBER()` is the strongly recommended strategy for both superior performance and enhanced code clarity. Regardless of the method selected, optimizing query speed requires ensuring that the grouping column (the `team` column in our example) is properly indexed to minimize disk I/O operations and maximize the efficiency of the lookup process.

Additional Resources for Advanced SQL Techniques

Successfully mastering the greatest-N-per-group problem is a vital step in developing proficiency in complex analytical SQL operations. For developers and administrators seeking to further enhance their data manipulation skills, exploring alternative methods, such as using advanced `JOIN` operations (like a `LEFT JOIN` combined with `GROUP BY`) or delving deeper into the extensive capabilities of Window Functions, will prove highly advantageous.

The following topics represent essential next steps for those looking to deepen their expertise in MySQL and analytical SQL:

Tutorial on deleting duplicate rows while keeping the latest record.

Guide to optimizing query performance using different types of indexing.

Detailed explanation of using Common Table Expressions (CTEs) for multi-step data processing and improved query organization.