How to Select the Row with the Maximum Value in a MySQL Column

Authored by

mohammed looti

January 6, 2026

1. Introduction and Primary Solution Using Subqueries

When working with relational databases like MySQL, a common requirement is to retrieve the entire row associated with the highest value found in a particular column. This task goes beyond simply finding the maximum value itself; it requires fetching all corresponding data fields for that record. The most direct and universally compatible method for accomplishing this involves using a subquery in conjunction with the powerful aggregate MAX() function. Understanding this approach is fundamental for effective data manipulation.

The basic structure of the query utilizes the SELECT statement to pull the desired fields, and then filters the results using the WHERE clause. The subquery is essential because it independently calculates the maximum value, which the outer query then uses as a comparison point. This nested structure ensures that the filtering operation is precise, targeting only those rows where the specified column's value exactly matches the calculated maximum. This allows for the efficient retrieval of the record with the highest score in a given column.

The general syntax is straightforward. You specify the table and column name, and the database engine handles the two-step process: first determining the maximum value, and then applying that constraint to the full dataset. This technique is highly reliable for ensuring you capture the single greatest value, or multiple values if ties exist, which is a critical consideration in robust database programming.

Here is the core structure used in MySQL to select the row containing the maximum value in a designated column, using a hypothetical `athletes` table:

```
SELECT id, team, points
FROM athletes
WHERE points=(SELECT MAX(points) FROM athletes);
```

2. Deconstructing the Subquery Mechanism

The efficiency and accuracy of this method hinge on the proper execution of the inner query, or subquery. The inner query, `(SELECT MAX(column_name) FROM table_name)`, executes first, scanning the entire column to compute the highest numerical value present. This result--a single scalar value--is then passed back to the outer query. It is crucial that the subquery returns only one column and one row, which the aggregate MAX() function inherently ensures.

Once the maximum value is identified (e.g., 37 in our forthcoming example), the outer query takes this value and incorporates it into its filtering logic. The outer query then becomes, conceptually, `SELECT * FROM table_name WHERE clause column_name = 37`. This subsequent filtering

operation scans the table again, but this time only matching rows are returned. This two-phase operation guarantees accuracy in the retrieval process by ensuring the outer query only acts upon the precise maximum value determined by the nested query.

While this technique is robust and highly readable, developers should be aware that subqueries can sometimes be less performant than alternative methods, especially on very large tables, due to the need for multiple table scans. However, for most common use cases and tables of moderate size, the subquery solution provides the clearest, most standard SQL path to achieving the desired result. We will explore performance alternatives, such as the `ORDER BY... LIMIT` method, later in this discussion.

3. Detailed Example Setup: The athletes Table

To illustrate this concept practically, let us establish a sample dataset. We will use a table named **athletes**, which tracks key statistics for various basketball players. This structure allows us to demonstrate how to find the player who achieved the highest score in the **points** column using real-world data points.

The table setup includes essential columns such as `id` (the primary key), `team`, and performance metrics like `points`, `assists`, and `rebounds`. Defining the table structure and inserting sample data is the first step in any practical database demonstration, ensuring our environment is ready for the query execution.

Below is the SQL code used to create the **athletes** table and populate it with six sample rows. Notice the use of appropriate data types like `INT` and `TEXT` to ensure data integrity and proper handling of numerical and string data, adhering to strong database design principles.

```
-- create table
CREATE TABLE athletes (
id INT PRIMARY KEY,
team TEXT NOT NULL,
points INT NOT NULL,
assists INT NOT NULL,
rebounds INT NOT NULL
);

-- insert rows into table
INSERT INTO athletes VALUES (0001, 'Mavs', 22, 4, 3);
INSERT INTO athletes VALUES (0002, 'Kings', 14, 5, 13);
INSERT INTO athletes VALUES (0003, 'Lakers', 37, 6, 10);
INSERT INTO athletes VALUES (0004, 'Nets', 19, 10, 3);
```

```
INSERT INTO athletes VALUES (0005, 'Knicks', 26, 12, 8);
INSERT INTO athletes VALUES (0006, 'Celtics', 15, 1, 2);

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

The resultant table confirms the data population, showing the diverse performance statistics for each team's representative athlete, making it easy to visually identify the maximum points value before running the query:

```
+----+---------+--------+---------+----------+
| id | team | points | assists | rebounds |
+----+---------+--------+---------+----------+
| 1 | Mavs | 22 | 4 | 3 |
| 2 | Kings | 14 | 5 | 13 |
| 3 | Lakers | 37 | 6 | 10 |
| 4 | Nets | 19 | 10 | 3 |
| 5 | Knicks | 26 | 12 | 8 |
| 6 | Celtics | 15 | 1 | 2 |
+----+---------+--------+---------+----------+
```

4. Executing the MAX Value Query in Practice

Now that the table is established, we can apply the primary subquery method described earlier. Our objective is to identify which athlete achieved the maximum number of **points** within this dataset. We are interested in the athlete's ID, team name, and the points themselves, demonstrating how to retrieve specific fields rather than all columns (`*`), which is a recommended practice for production queries.

The inner query first calculates `MAX(points)`, determining that the highest score in the table is 37. The outer query then filters the records to find any row where `points = 37`. This process isolates the specific record(s) corresponding to the record holder efficiently.

We execute the following SELECT query in MySQL:

```
SELECT id, team, points
FROM athletes
WHERE points=(SELECT MAX(points) FROM athletes);
```

The execution yields the following result, confirming that the athlete associated with ID 3 and the

Lakers holds the maximum score:

```
+----+--------+--------+
| id | team | points |
+----+--------+--------+
| 3 | Lakers | 37 |
+----+--------+--------+
```

This output clearly demonstrates that only the row corresponding to the maximum value in the **points** column has been retrieved. The power of this method lies in its simplicity and strict adherence to the defined maximum value, avoiding potential data contamination from other records.

5. Handling Scenarios with Ties and Duplicate Maximums

A critical feature of the subquery method utilizing the equality operator (=) is its native ability to handle ties. If multiple rows share the same maximum value in the specified column, this query structure will successfully return all of them. This behavior is often desired when the objective is to find all record holders, ensuring no top-performing entry is overlooked.

For instance, if we modified the data such that both the Lakers and the Knicks scored 37 points, the MAX() function would still return 37, and the outer query's WHERE clause would select both rows where `points = 37`. This ensures comprehensive data retrieval regardless of duplication in the maximum value field.

However, if the goal is strictly to retrieve only one record, even in the event of a tie, alternative methods such as using `ORDER BY` combined with `LIMIT 1` (as detailed below) are necessary. It is important to remember this distinction when designing database queries: the subquery method prioritizes completeness of the maximum value retrieval, while the `LIMIT` approach prioritizes restricting the result set size.

Note: If there are multiple rows tied with the max value in a particular column, then each of those rows will be returned when using the subquery approach.

6. Alternative Method: Using ORDER BY and LIMIT 1

While the subquery approach is highly readable, an alternative and often more performance-optimized method for selecting the row with the maximum value is to use the SELECT statement combined with `ORDER BY` and `LIMIT`. This method sorts the entire dataset based on the target column in descending order and then restricts the output to the top one row.

The key advantage of this approach, especially within highly optimized database systems, is that indexing on the sorted column can dramatically improve performance compared to the two-pass scan required by the subquery method. By sorting in **DESCENDING** order (`DESC`), the highest value naturally appears at the top of the result set, making the subsequent `LIMIT 1` operation very fast.

The syntax is considerably cleaner and more direct than the nested subquery, simplifying the overall structure of the command:

```
SELECT id, team, points
FROM athletes
ORDER BY points DESC
LIMIT 1;
```

A significant distinction here is how ties are handled. If there are multiple rows with the maximum value, `LIMIT 1` will arbitrarily return only one of those tied rows, based on internal database ordering (which is often unpredictable without a secondary sort key like `id`). If you must retrieve a specific tied record, adding a secondary `ORDER BY` criterion (e.g., `ORDER BY points DESC, id ASC`) is necessary to ensure deterministic results. If the business requirement is to fetch all tied maximums, stick to the subquery method.

7. Performance Considerations for Large Datasets

When dealing with tables containing millions or billions of rows, the choice of query structure can profoundly impact execution time. Both the subquery method and the `ORDER BY... LIMIT 1` method have different performance characteristics that must be considered, particularly in high-traffic production environments.

For the subquery method (using MAX() function in the WHERE clause), the database performs two separate operations: a full scan (or index lookup) to calculate the maximum value, and then another scan/lookup to retrieve the corresponding rows. If the column being queried (e.g., `points`) is properly indexed, the index can be used to quickly find the highest value without scanning the entire table data, significantly optimizing the first step, but the two-pass nature remains a factor.

Conversely, the `ORDER BY... LIMIT 1` method relies on sorting the data. While sorting can be computationally intensive, modern SELECT optimizers are highly efficient at utilizing composite indexes to find the top 1 row quickly. If an index exists on the column (`points`), the database can often perform a "covering index scan" or simply walk the index backwards to find the maximum value and the required row data rapidly, often outperforming the two independent scans of the subquery method, especially when only retrieving the top result.

Best practice dictates that the column being used for finding the maximum value should always be indexed. This dramatically speeds up both the calculation of the maximum value and the sorting operation. Developers should always test both query structures using the `EXPLAIN` tool in MySQL to determine which provides the superior execution plan for their specific table size and indexing strategy.

8. Summary of Approaches and Further Study

Choosing the best approach depends heavily on the specific requirements of the application--namely, whether handling ties is mandatory, or if maximizing performance for single record retrieval is the primary concern.

Comparison of Max Value Retrieval Methods

Method 1: Subquery with MAX() and WHERE: This technique is highly recommended when it is necessary to retrieve **all rows** that share the maximum value (handling ties gracefully). Its logic is clear and aligns well with standard SQL practices.

Method 2: ORDER BY... LIMIT 1: This is generally the preferred approach for maximizing performance on very large tables, especially when you only require a single, top record, and the ambiguity regarding which tied record is returned is acceptable.

For complex scenarios involving grouping (e.g., finding the maximum score for each team within its conference), more advanced techniques like window functions (e.g., `ROW_NUMBER()` or `RANK()`) or correlated subqueries are often required. These advanced methods offer greater flexibility and better logical partitioning capabilities than simple aggregate queries when dealing with partitioned data sets.

Mastering these foundational techniques is essential for any data professional working with database retrieval operations. We encourage exploration of further tutorials explaining how to perform other common tasks in MySQL, such as calculating averages, grouping data using `GROUP BY`, and optimizing complex joins.