Learning MySQL: Retrieving the Last N Rows from a Table

Authored by

Mohammed looti

November 12, 2025

Understanding the Challenge of Selecting the Latest Records

When interacting with robust relational database systems such as MySQL, defining the concept of the “last N rows” requires sophisticated handling. Unlike sequential files or basic spreadsheets, database tables do not maintain an inherent physical order that corresponds to insertion time. Therefore, to reliably identify the most recently added or highest-indexed records, developers must utilize a column explicitly designed to track chronological insertion order. This column is typically the Primary Key (like an auto-incrementing `id`) or a dedicated timestamp field. This article provides an expert, standardized SQL solution that efficiently addresses this requirement.

The core difficulty arises because a standard SELECT query paired with the powerful LIMIT clause only retrieves the first N records encountered based on the database engine's current access path. To successfully retrieve the “last” records, we must first impose a reverse sorting mechanism on the entire dataset using our chronological column (e.g., `id`). Once the newest records are positioned at the top of the result set, we apply the LIMIT restriction to capture only that desired subset. Finally, we must re-sort the limited subset back into the logical, ascending chronological order for proper presentation, though this last step is optional depending on the user requirement.

The most reliable and robust method for achieving this specific data retrieval pattern involves using a technique known as a nested query, or derived table. This approach ensures that the limiting operation is performed only after the data has been correctly sorted in reverse, guaranteeing accurate isolation of the most recent records before they are returned to the client application in a readable format. This pattern is essential for applications requiring accurate chronological history retrieval and is highly reliable across various MySQL versions.

The MySQL Solution: Nested Queries and LIMIT

The standard and most effective syntax in MySQL for retrieving the last N rows leverages a subquery structure combined with the crucial application of the ORDER BY and LIMIT clauses. The essential logical flow is straightforward: sort the data in descending order based on the tracking column (typically the primary key or a creation timestamp), retrieve the top N results, and then optionally sort those N results back into ascending order for display.

This two-stage process is necessary because we cannot simply combine `ORDER BY DESC` with `LIMIT` and expect the correct final sort order if ascending presentation is required. The nested query pattern effectively creates a temporary result set containing only the newest records, which the outer query can then safely sort without processing the entire original table again.

Below is the general structure of this powerful technique, demonstrated here selecting the last 10 rows from a hypothetical table named `athletes`:

```
SELECT * FROM
(
SELECT * FROM athletes ORDER BY id DESC LIMIT 10
) AS temp
ORDER BY id ASC;
```

Breaking Down the Nested Query Syntax

To ensure maximum understanding of how this query operates, we must dissect its three primary components. Understanding the role of the inner query versus the outer query is key to mastering this technique.

Inner Query: `SELECT * FROM athletes ORDER BY id DESC LIMIT 10`. This operation is the core mechanism for isolating the desired records. By applying `ORDER BY id DESC`, we effectively place the newest records (those with the largest `id` values) at the beginning of the result set. The subsequent `LIMIT 10` then captures exactly the first 10 rows of this reversed list, which are, by definition, the “last 10 rows” inserted into the table.

Temporary Alias (Derived Table): The result generated by the inner query is treated as a temporary, virtual table. This derived table must be assigned an alias, in this case, `temp`. This aliasing is a mandatory requirement in SELECT statements within MySQL when using subqueries.

Outer Query: `SELECT * FROM temp ORDER BY id ASC`. The outer query selects all columns from the restricted set of 10 rows (the `temp` table). It then applies a final `ORDER BY id ASC`. This step is strongly recommended because it restores the natural, chronological sequence of the records, making the output intuitive and easy to read for the end user or application.

This structure guarantees that the limiting operation happens precisely on the newest data, while the outer query handles the final desired display order. To retrieve a different count of records, you only need to modify the numerical value supplied to the `LIMIT` clause within the inner query.

Setting Up the Sample Data

To effectively demonstrate the functionality of the nested query technique, we will establish a sample table called `athletes`. This table will contain a small, manageable dataset representing basketball player statistics, allowing us to accurately track and verify which rows qualify as the “last N” based on the chronological sequence of the `id` column.

The defined table structure includes three columns: a unique auto-incrementing Primary Key (`id`), the player's `team` (stored as `TEXT`), and their recorded `points` (stored as `INT`). We will populate

this table with 13 distinct records to ensure we have sufficient data volume for effective limiting operations.

```
-- 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, 'Warriors', 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);
INSERT INTO athletes VALUES (0011, 'Jazz', 25);
INSERT INTO athletes VALUES (0012, 'Jazz', 18);
INSERT INTO athletes VALUES (0013, 'Kings', 14);

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

After successfully executing the setup scripts, the `athletes` table contains thirteen rows. The following output confirms the full dataset, establishing our baseline: row 13 is definitively the latest entry based on the auto-incrementing `id` sequence, while row 1 is the oldest.

Output: Full Dataset

```
+----+----------+--------+
| id | team | points |
+----+----------+--------+
| 1 | Mavs | 22 |
| 2 | Mavs | 14 |
| 3 | Lakers | 37 |
```

| 4 | Knicks | 19 |
| 5 | Warriors | 26 |
| 6 | Knicks | 40 |
| 7 | Lakers | 21 |
| 8 | Celtics | 15 |
| 9 | Hawks | 18 |
| 10 | Celtics | 23 |
| 11 | Jazz | 25 |
| 12 | Jazz | 18 |
| 13 | Kings | 14 |
+----+----------+--------+

Demonstration: Selecting the Last Ten Rows

Our primary objective is to select the most recent 10 rows inserted into the `athletes` table. Given the full dataset above, we anticipate retrieving the records ranging from `id = 4` (Knicks) through `id = 13` (Kings).

We apply the nested query structure discussed previously, specifically setting `LIMIT 10` in the inner clause to capture the required subset of data points. This query structure ensures that we accurately isolate the newest records before displaying them in their proper chronological order.

```
SELECT * FROM
(
SELECT * FROM athletes ORDER BY id DESC LIMIT 10
) AS temp
ORDER BY id ASC;
```

The execution of this query yields the following output, perfectly matching our expectation. The results confirm that the inner `ORDER BY DESC` and LIMIT 10 successfully isolated the last ten records. Furthermore, the outer `ORDER BY id ASC` ensured they are correctly displayed chronologically, starting from `id 4` and concluding with the latest entry, `id 13`.

Output: Last 10 Rows

+----+----------+--------+
| id | team | points |
+----+----------+--------+
| 4 | Knicks | 19 |
| 5 | Warriors | 26 |

| 6 | Knicks | 40 |
| 7 | Lakers | 21 |
| 8 | Celtics | 15 |
| 9 | Hawks | 18 |
| 10 | Celtics | 23 |
| 11 | Jazz | 25 |
| 12 | Jazz | 18 |
| 13 | Kings | 14 |
+----+----------+--------+

Adjusting the Query for Specific N Values

The primary advantage of this nested query method is its inherent flexibility. By simply modifying the numerical argument provided to the LIMIT clause within the inner query, we can select any arbitrary number of the most recent rows. For instance, if our application only requires displaying the three latest entries for a summary view, we only need to adjust one value.

We would update the inner SELECT statement to specify `LIMIT 3`. This focuses the retrieval solely on the three rows with the highest `id` values (11, 12, and 13).

```
SELECT * FROM
(
SELECT * FROM athletes ORDER BY id DESC LIMIT 3
) AS temp
ORDER BY id ASC;
```

The resultant output correctly displays only the three newest entries in the table, clearly demonstrating the direct control the `LIMIT` parameter provides over the subset size. This functionality is invaluable for displaying recent activity logs, transaction histories, or time-series data where the chronological order of entry is paramount.

Output: Last 3 Rows

++----+-------+--------+
| id | team | points |
+----+-------+--------+
| 11 | Jazz | 25 |
| 12 | Jazz | 18 |
| 13 | Kings | 14 |
+----+-------+--------+

Performance and Advanced Considerations

While the nested query method is considered robust and standard SQL practice, database administrators and developers must always prioritize performance, especially when dealing with extremely large tables containing millions or billions of rows. The efficiency of this query is highly dependent on the indexing of the column used for sorting.

For optimal performance, the column referenced in the ORDER BY clause (in our case, `id`) must be indexed. If `id` is defined as the primary key, it is automatically indexed, which facilitates extremely fast lookups and sorting, even when reversed. Conversely, if you attempt to sort by a non-indexed timestamp column, MySQL may be forced to perform a resource-intensive full table scan before sorting, which can introduce significant latency.

It is important to note an alternative optimization: if the requirement is simply to retrieve the latest N records and you do not need them sorted in ascending order afterward (i.e., descending order, newest-first, is acceptable), you can completely avoid the complexity and overhead of the nested query. A simpler query provides superior performance in this specific scenario:

SELECT * FROM athletes ORDER BY id DESC LIMIT 10;

For tasks more complex than simply retrieving the absolute latest entries--such as advanced pagination (finding the 10 rows just before the newest 10) or utilizing filtered criteria--more complex offset logic or modern window functions might be necessary. However, for the straightforward requirement of retrieving the absolute latest N records and displaying them chronologically, the nested SELECT structure presented here remains the clearest and most standard implementation.

Additional Resources

The following resources provide essential information on related topics, helping you master common database tasks and optimize performance in MySQL:

How to Use the `LIMIT` Clause Effectively for Pagination
Understanding the Performance Implications of `ORDER BY` with Indexes
A Guide to SELECT Subqueries and Derived Tables in MySQL