How to Select the Last N Rows from a MySQL Table Easily

Authored by

mohammed looti

January 5, 2026

When working with relational databases, a common requirement is accessing the most recent data entries. In MySQL, defining the "last N rows" is not as simple as merely counting backward, as the inherent structure of a table is unordered unless explicitly defined. To successfully select the most recently inserted or updated records--which typically correspond to the highest values in an auto-incrementing identifier--we must enforce a specific ordering. This technique is invaluable for applications requiring real-time dashboards, viewing recent user activity, or paginating through the newest additions to a dataset. The fundamental approach involves a strategic combination of the powerful ORDER BY clause and the restrictive LIMIT clause within a carefully constructed nested SQL query.

The challenge arises because simply using `LIMIT N` without preceding it with an `ORDER BY` clause can return arbitrary rows, depending on how the database engine chooses to store or retrieve the data internally. To guarantee that we are truly selecting the "last" rows, we must rely on a sequential indicator, most often the table's auto-incrementing **Primary Key** or a reliable timestamp column. By first sorting the entire dataset in descending order based on this indicator, the most recent rows bubble up to the top. Only then can the `LIMIT` clause effectively restrict the result set to the top N records, which represent our target "last N rows."

However, retrieving the data in descending order might not be the desired final presentation format. Most users expect to see data, even recent data, presented in chronological or sequential order (ascending ID). This necessitates an additional step: wrapping the initial selection (the descending sort and limit) within a subquery. This subquery acts as a temporary table, allowing us to perform a second, final sort on the result set, ensuring the output is clean, readable, and sequentially correct, while still containing only the latest N records. This nested structure ensures both accuracy in row selection and clarity in presentation.

The Core Solution: Combining ORDER BY and LIMIT

To select the last N rows from any MySQL table, the standard and most efficient technique involves a two-stage process encapsulated within a nested query. The first stage, executed by the inner query, is responsible for identifying and isolating the latest records. This stage leverages the `ORDER BY` clause to sort the data using a monotonically increasing column--ideally the primary key (like an `id` column) or a creation timestamp--in **descending** (`DESC`) order. Sorting in descending order places the newest entries at the beginning of the result set. Immediately following this sorting, the LIMIT clause is applied, specifying the exact number N of rows we wish to retrieve.

The key innovation of this method is the use of a subquery. The result of the inner query (the N latest, but currently reverse-sorted, records) is treated as a temporary derived table. This derived table, often given an alias like `temp`, contains only the necessary subset of data. Without this temporary wrapper, attempting to apply a second `ORDER BY` clause to re-sort the results into

ascending order would confuse the SQL interpreter or fail to execute correctly because the `LIMIT` application often occurs late in the execution plan, sometimes making the final order unpredictable without the subquery encapsulation.

The second stage, executed by the outer query, then selects all columns from this derived table (`SELECT * FROM temp`) and applies a final `ORDER BY` clause, this time using **ascending** (`ASC`) order on the same identifier column. This action reverses the order of the N selected rows back to their original sequence, ensuring the output is presented from the oldest of the N records to the newest. This robust two-step methodology guarantees both the accurate selection of the latest data and a logical presentation format.

Detailed Breakdown of the SQL Syntax

The general syntax used in MySQL to select the last N rows from a table, ensuring correct chronological presentation, is shown below:

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

This particular example selects the last 10 rows from the table named **athletes** and orders the results in ascending order by the values in the **id** column of the table.

Let us dissect the components of this crucial SQL query structure. The inner query, `(SELECT * FROM athletes ORDER BY id DESC LIMIT 10)`, is the selection engine. By applying `ORDER BY id DESC`, MySQL scans the table and sorts all records such that the row with the highest `id` value is at the top. The subsequent `LIMIT 10` then truncates the result set, isolating only the 10 rows with the highest ID values. This derived set of 10 rows is then assigned the alias `AS temp`, transforming it into a temporary dataset accessible by the outer query.

The outer query, `SELECT * FROM temp ORDER BY id ASC`, retrieves all columns from the temporary table `temp`. Because these 10 rows were originally retrieved in descending order (highest ID first), the final step applies `ORDER BY id ASC`. This final sort reorders the 10 rows back into their natural chronological sequence, from the smallest ID within that set to the largest. This nested approach ensures that while the selection process prioritizes the newest records efficiently, the output maintains logical ascending order, which is generally preferred for display purposes. To select a different number of rows, developers simply need to modify the integer value immediately

following the `LIMIT` keyword within the inner query.

To select a different number of rows, simply change the number after **LIMIT** in the query.

The following example shows how to use this syntax in practice.

Prerequisites for Reliable Selection: The Importance of Primary Keys

Defining what constitutes the "last N rows" fundamentally depends on the existence and integrity of a sequential column. In the context of database design, the use of a reliable, auto-incrementing primary key is absolutely critical for this methodology. A primary key (often named `id`) ensures that every row in the table is uniquely identified, and when configured for auto-incrementation, it naturally provides a monotonic sequence that perfectly correlates with the insertion order of records. This sequence is what allows the `ORDER BY id DESC` clause to accurately pinpoint the most recently added data.

If a table lacks an auto-incrementing primary key, alternative methods must be used, which are often less reliable or less performant. For instance, one might rely on a timestamp column (e.g., `created_at`) which must be meticulously managed to ensure it is populated precisely at the moment of insertion. While a timestamp column can work, it introduces potential risks, such as conflicts arising from concurrent inserts or variations in system clock time, which might lead to an imperfect sequence. However, if a timestamp column is indexed and used, the principle remains the same: `ORDER BY created_at DESC LIMIT N`.

The efficiency of this entire operation is heavily dependent on indexing. For the nested query approach to perform well, particularly on large tables, the column used in the ORDER BY clause (whether it is the primary key or a timestamp) must be indexed. Without an index, the MySQL engine would be forced to perform a full table scan and sort operation on the entire dataset before applying the LIMIT clause, which severely impacts performance. With an index, the engine can quickly locate the highest N values, drastically reducing execution time.

Practical Application: Setting up the Sample Data

To demonstrate the functionality of selecting the last N rows, we will utilize a sample table named `athletes`. This table models basic information about various basketball players and is structured to include an auto-incrementing primary key, making it ideal for testing our sorting logic. The structure includes an `id` column, a `team` identifier, and a record of `points` scored.

Suppose we have the following table named **athletes** that contains information about various basketball players:

```
-- 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;
```

Output:

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

Notice that the table has a total of 13 rows. The row with `id=1` is the oldest entry, and the row with `id=13` is the newest entry, based on the sequential nature of the primary key. This complete view serves as our baseline against which we will test the efficiency of our filtering query.

Case Study 1: Selecting the Last Ten Records

Suppose that we would like to select the last 10 rows from the table. Since the table contains 13 total rows, selecting the last 10 rows means we should retrieve IDs 4 through 13. We apply the nested query structure, specifying `LIMIT 10` in the inner clause. This structure ensures that only the 10 rows corresponding to the highest ID values are selected, and then re-sorted for proper display.

We can use the following syntax to do so:

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

The execution of this statement first sorts all 13 rows by ID in descending order (13 down to 1). It then applies the LIMIT clause, retaining only rows 13 down to 4. Finally, the outer query takes this result set and uses the `ORDER BY id ASC` clause to present the data sequentially, starting from ID 4 and ending at ID 13. This successful execution demonstrates the reliability of the nested query pattern for precise data retrieval based on insertion order.

Output:

+----+----------+--------+
| 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 |
+----+----------+--------+

This query selects the last 10 rows of the table, in ascending order based on the values in the **id** column.

Case Study 2: Retrieving the Most Recent Three Entries

To further illustrate the flexibility of this technique, let's adjust the query to retrieve a smaller subset--specifically, the last three rows inserted into the table. This task is often necessary for widgets or summarized views where only the absolute latest activity is relevant. By simply changing the parameter passed to the LIMIT clause in the inner query from 10 to 3, we target rows 11, 12, and 13.

For example, we can use the following syntax to select the last 3 rows in the table:

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

The inner query rapidly identifies the rows with the top three highest ID values (13, 12, and 11). This restricted set is then passed to the outer query. The outer query applies the final ORDER BY clause, ensuring that the final presentation lists the results chronologically (11, 12, 13). This confirms the adaptability of the query structure to any specified N value, providing a highly scalable solution for fetching recent data segments.

Output:

++----+-------+--------+
| id | team | points |
+----+-------+--------+
| 11 | Jazz | 25 |
| 12 | Jazz | 18 |

```
| 13 | Kings | 14 |
+----+-------+--------+
```

Notice that only the last 3 rows in the table are selected.

Performance Considerations and Alternatives

While the nested `ORDER BY...DESC LIMIT N` query is the conventional and highly effective method for retrieving the last N rows in MySQL, developers must remain cognizant of performance implications, especially when dealing with tables containing millions of records. As previously mentioned, indexing the column used for ordering is not optional; it is fundamental to maintaining fast query times. An index allows the database engine to quickly traverse the B-tree structure, locate the maximal values, and satisfy the `LIMIT` condition without performing a resource-intensive full table sort.

For very large datasets where even indexed sorting might become slow due to high concurrency or I/O constraints, developers should consider optimization techniques. If the data retrieval is part of a high-traffic API endpoint, caching the most recent results at the application layer can reduce database load. Furthermore, if retrieving the "last N" rows is always relative to the last known ID (e.g., for continuous polling or pagination), using a `WHERE id >` clause combined with `ORDER BY id DESC LIMIT N` can be significantly faster, as it allows the database to avoid scanning older sections of the table entirely.

In newer versions of MySQL (8.0+), advanced users might consider the use of window functions like `ROW_NUMBER()`. While window functions offer powerful analytical capabilities and cleaner syntax for ranking, they often require the entire partition to be processed before the ranking can be applied. In the specific case of simply fetching the last N rows based on an indexed column, the classic nested `ORDER BY/LIMIT` approach remains generally more performant and straightforward than complex window function implementations because the database can optimize the indexed `LIMIT` operation extremely well. Therefore, for this specific task, the established nested SQL query remains the best practice.

Expanding Your MySQL Skills

Mastering the retrieval of ordered data subsets is just one step in becoming proficient with relational databases. Understanding how clauses like `ORDER BY` and `LIMIT` interact within subqueries is key to writing efficient and reliable code. For those looking to expand their knowledge beyond simple row selection, there are many related concepts that build upon this foundation, such as advanced pagination techniques, using offset correctly, and handling composite keys for ordering.

The methodology demonstrated here forms the basis for constructing complex reports and transactional queries. The ability to isolate the latest data reliably is essential for logging, auditing, and high-frequency data applications. We strongly recommend continuing your exploration of MySQL's extensive documentation to understand how indexing and query optimization can maximize the performance gains achieved by correctly using the `ORDER BY` and `LIMIT` clauses.

The following tutorials explain how to perform other common tasks in MySQL:

How to Paginate Large Datasets
Using Window Functions for Ranking
Optimizing Query Performance with Indexing