Learning MySQL: How to Retrieve the Last Day of the Previous Month

Authored by

Mohammed looti

November 12, 2025

The Critical Need for Date Boundaries in Database Reporting

Date and time manipulation stands as a **fundamental requirement** for virtually every relational database application, especially when generating reports or performing complex financial aggregation. In MySQL, the need to precisely query for specific calendar boundaries, such as the last day of a given month, is extremely common. Developers frequently face scenarios--like calculating monthly recurring revenue (MRR) or determining billing cycles--that necessitate finding the final day of the month immediately preceding a reference transaction date.

While complex date arithmetic can be cumbersome to implement manually, MySQL provides powerful built-in functions that elegantly streamline this process. These functions eliminate the potential pitfalls associated with manual calculations, such as miscalculating leap years or months with varying lengths (28, 30, or 31 days). The most efficient solution involves a precise and nested combination of specialized date functions, ensuring high accuracy and performance directly within the query engine.

Below, we introduce the concise syntax required to efficiently retrieve the last day of the previous month for any date stored within your database tables, providing a powerful tool for time-series analysis and reporting:

SELECT sales_date, LAST_DAY(sales_date - INTERVAL 1 MONTH)
FROM sales;

This specific structure generates a new column containing the last day of the previous month, cross-referenced against the original **sales_date** column from the **sales** table. This calculation is indispensable for crucial business intelligence tasks, including the calculation of monthly churn rates, generation of accurate period-over-period comparisons, and alignment of transaction data with strict monthly financial cutoffs.

Deconstructing the MySQL Date Functions

To successfully pinpoint the last day of the preceding month, our solution relies on the synergy between two core MySQL date functions: the powerful `INTERVAL` clause and the dedicated `LAST_DAY()` function. Understanding the precise role of each component is essential for mastering advanced date manipulation within MySQL.

The initial step involves shifting the date backward using the `INTERVAL` clause. This clause is specifically designed to perform calendar-aware date arithmetic, allowing us to subtract a defined unit of time--in this case, exactly one calendar month--from the reference date. This action moves the date pointer accurately into the desired previous month, regardless of the starting day or the

month's length. This avoids the inaccuracy of simply subtracting a fixed number of days.

Once the date has been correctly shifted backward, the outer `LAST_DAY()` function is applied. This function is explicitly designed to accept a date argument and return the date corresponding to the final calendar day of that specific month. By nesting the subtraction operation inside `LAST_DAY()`, we force the system to first calculate the date one month prior, and then immediately snap that result to the month's maximum day, ensuring the resulting date is always the final day marker.

Establishing the Demonstration Dataset

To provide a practical illustration of this technique, we will first set up a sample database table. We define a table named `sales`, designed to simulate tracking transactions from a hypothetical grocery store. This table minimally includes the `store_ID` (as the primary key), a description of the `item` sold, and the critical transaction date field, `sales_date`, which utilizes the standard DATE data type.

The following SQL commands detail the table creation and the insertion of five sample records. These records are intentionally selected to cover various temporal boundaries, including dates early in the month (e.g., February 10th) and dates that require rolling back across a year boundary (e.g., January 14th). This diverse data ensures a comprehensive test of the function's reliability.

Suppose we have the following table named **sales** that contains information about sales made at various grocery stores on various dates:

```
-- create table
CREATE TABLE sales (
store_ID INT PRIMARY KEY,
item TEXT NOT NULL,
sales_date DATE NOT NULL
);

-- insert rows into table
INSERT INTO sales VALUES (0001, 'Oranges', '2024-02-10');
INSERT INTO sales VALUES (0002, 'Apples', '2024-11-25');
INSERT INTO sales VALUES (0003, 'Bananas', '2024-06-30');
INSERT INTO sales VALUES (0004, 'Melons', '2024-01-14');
INSERT INTO sales VALUES (0005, 'Grapes', '2024-05-19');

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

Output: The initial table contents confirm the structure and data integrity before we proceed with the date calculation.

+----------+---------+------------+
| store_ID | item | sales_date |
+----------+---------+------------+
| 1 | Oranges | 2024-02-10 |
| 2 | Apples | 2024-11-25 |
| 3 | Bananas | 2024-06-30 |
| 4 | Melons | 2024-01-14 |
| 5 | Grapes | 2024-05-19 |
+----------+---------+------------+

Executing the Core Query and Analyzing Results

Our primary goal is to run the combined function against the `sales` table, extracting the original `sales_date` and simultaneously generating a derived column that precisely identifies the last day of the preceding month. This derived date is frequently utilized in analytical pipelines to align individual transactions with monthly reporting cycles or fiscal cutoffs.

To achieve this, we reuse the combined function syntax introduced earlier: `LAST_DAY(sales_date - INTERVAL 1 MONTH)`. This powerful instruction tells MySQL to process the calculation row-by-row, ensuring meticulous accuracy even when dates span across complex temporal boundaries, such as the crucial transition from January 1st to the previous year's December 31st.

We use the following SQL statement to execute the calculation against our sample data:

SELECT sales_date, LAST_DAY(sales_date - INTERVAL 1 MONTH)
FROM sales;

Output: The result set clearly shows the new calculated date field alongside the original transaction date.

+------------+--+
| sales_date | LAST_DAY(sales_date - INTERVAL 1 MONTH) |
+------------+--+
| 2024-02-10 | 2024-01-31 |
| 2024-11-25 | 2024-10-31 |
| 2024-06-30 | 2024-05-31 |
| 2024-01-14 | 2023-12-31 |

| 2024-05-19 | 2024-04-30 |
+------------+--+

The result set confirms the reliability of the function. For example, the sale recorded on **February 10, 2024**, correctly returns **January 31, 2024**, as the last day of the prior month. Furthermore, the transaction on **January 14, 2024**, accurately rolls back across the year boundary, yielding **December 31, 2023**.

Enhancing Readability with the AS Clause

While the previous query is functionally correct, the automatically generated column header, `LAST_DAY(sales_date - INTERVAL 1 MONTH)`, is overly verbose and impractical for integration into application code, further reporting, or complex joins. A crucial best practice in SQL development is utilizing the `AS` clause to assign a descriptive and meaningful alias to calculated or aggregated fields.

By appending `AS last_previous` to our function call, we provide a clear, concise, and immediately understandable name for the derived column. This refinement dramatically improves the query's readability and maintainability, allowing subsequent developers or analysts to interpret the data output quickly without needing to dissect the underlying nested functions.

We utilize the **AS** statement to give a specific, user-friendly name to this new column:

SELECT sales_date, LAST_DAY(sales_date - INTERVAL 1 MONTH) AS last_previous
FROM sales;

+------------+---------------+
| sales_date | last_previous |
+------------+---------------+
| 2024-02-10 | 2024-01-31 |
| 2024-11-25 | 2024-10-31 |
| 2024-06-30 | 2024-05-31 |
| 2024-01-14 | 2023-12-31 |
| 2024-05-19 | 2024-04-30 |
+------------+---------------+

The new column is now named **last_previous**, which is much easier to read and integrate into application logic. This naming convention is particularly useful when exporting data or using the query as a subquery for more complex reporting structures.

Detailed Breakdown of the Two-Step Logic

To fully appreciate the power of this technique, it is vital to grasp the precise order of operations that guarantees an accurate result every time, regardless of the input date. The formula is structured as two inherently dependent steps: first, calculating the date one full calendar month prior, and second, forcing that derived date to the final day marker of that targeted month.

The critical first stage is handled by the `INTERVAL` operator: `sales_date - INTERVAL 1 MONTH`. This operator is crucial because it is inherently **calendar-aware**. It correctly handles month transitions without simply subtracting a fixed number of days. For instance, if the original date is March 31st, subtracting one month yields February 29th (in a leap year) or February 28th (in a common year). The subtraction operation intelligently adjusts for months with fewer days.

Note: This formula works by executing the following steps in sequence:

The system first performs date arithmetic by subtracting one month using the `INTERVAL` clause. For **2024-02-10**, this results in the date **2024-01-10**.
Then, it uses the `LAST_DAY()` function on the result (2024-01-10) to determine the last day of January, which is **2024-01-31**.

This robust methodology provides a highly efficient and reliable way to execute complex date calculations directly within your SQL queries, minimizing the performance overhead and potential error sources associated with processing date logic in external application layers.

Further MySQL Date Resources

Date manipulation in MySQL is a deep and rewarding topic. Achieving mastery over core functions like `LAST_DAY()` and the `INTERVAL` clause is foundational for writing advanced analytical queries and optimizing data handling processes. These sophisticated functions enable developers to tackle common requirements such as filtering data based on dynamic timeframes, calculating precise durations between two different timestamps, and managing complex time zone manipulations.

To further refine your database management and query optimization skills, consider exploring tutorials and official documentation covering related functionalities. Expanding your knowledge of these core functions will dramatically improve your database management skills, allowing you to write more sophisticated and optimized queries.

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

MySQL: How to Select Rows where Date is Equal to Today