Learning MySQL: How to Calculate the First Day of the Previous Month

Authored by

Mohammed looti

November 12, 2025

The Complexities of Date Manipulation in SQL Environments

In virtually every relational database environment, mastery of date manipulation is a foundational skill. Whether constructing comprehensive financial reports, analyzing time-series data, or generating periodic summaries, developers and database administrators frequently encounter scenarios requiring precise handling of chronological boundaries. A seemingly straightforward requirement--calculating the **first day of the previous month**--often proves surprisingly intricate in practice, necessitating a robust, carefully constructed solution within the SQL framework.

The inherent difficulty stems from the variable nature of calendars. Simply subtracting a fixed number of days is unreliable; months have different lengths (28, 29, 30, or 31 days), and the calculation must correctly account for leap years and year-end transitions. Therefore, reliance on standard arithmetic alone is insufficient. Instead, we must strategically combine specific MySQL Date and Time Functions to engineer a formula that guarantees accuracy regardless of the starting date input.

The primary hurdle is that MySQL lacks a single, dedicated function (like `FIRST_DAY_OF_PREVIOUS_MONTH()`) to achieve this specific calculation instantly. Consequently, we must devise a multi-step approach involving backward date shifting, finding a hard boundary point, and then advancing one step forward. This method ensures that the calculation is robust, highly performant, and consistently returns the correct first day of the preceding month relative to the original date input.

Constructing the Essential MySQL Formula for Monthly Boundaries

To calculate the first day of the previous month efficiently across large datasets within a database table, we employ a highly condensed and powerful combination of built-in **MySQL** functions. This particular formula is favored by professionals because it flawlessly manages all calendar edge cases, including transitions between years (e.g., January 1st to the previous December 1st).

The syntax below provides the most reliable method for this specific date query. The solution strategically utilizes the LAST_DAY() function to pinpoint the end of a month and the INTERVAL keyword for precise temporal arithmetic, allowing us to manipulate the date components accurately:

SELECT sales_date, LAST_DAY(sales_date - INTERVAL 2 MONTH) + INTERVAL 1 DAY FROM sales;

This query generates a calculated column alongside the original **sales_date** field from the **sales** table. The underlying logic is elegant yet critical: by subtracting two months, we intentionally

overshoot the target, landing two months prior. We then use `LAST_DAY()` to find the absolute end of that distant month, and finally, adding one day steps us forward exactly to the first day of the subsequent month--which is precisely the start of the previous month we sought. Understanding this mechanism is vital for advanced date manipulation in enterprise database environments.

Preparing the Demonstration Environment with Sample Data

To effectively illustrate the practical application and verify the accuracy of this formula, we must first establish a sample environment. We will create a simple **database table** named **sales**, designed to mimic real-world transaction records. This setup is crucial for executing the complex calculation query and observing the transformation of the dates directly.

The structure of the **sales** table is minimal but effective, featuring a primary key (store identifier), a descriptive text field (item sold), and the essential date field, **sales_date**, which serves as the input for our calculation. The following SQL commands define this structure and subsequently populate it with five distinct records. These records are intentionally varied, spanning different months and crossing the calendar year boundary, ensuring a thorough test of the date calculation logic.

```
-- 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;
```

Once these setup commands are executed, the resulting **sales** table provides the foundational data for our analysis. The variety of dates confirms that our formula will be tested against different monthly starting points and ensures proper handling of the critical year transition from January to December.

Output of Sales Table:

```
+----------+---------+------------+
| 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 Query and Analyzing the Results

With the sample data prepared, our next step is to execute the calculation query. Our primary objective is to enrich this dataset by incorporating a new, derived column that accurately pinpoints the first day of the month immediately preceding the date stored in **sales_date**. We apply the previously established formula directly against the **sales** table using a simple SELECT statement.

This query efficiently retrieves the original date alongside the calculated previous month's start date, demonstrating **MySQL**'s capability to perform complex date arithmetic with minimal overhead. The internal handling of month and year boundaries ensures swift and reliable results, which is essential for high-volume database operations.

```
SELECT sales_date, LAST_DAY(sales_date - INTERVAL 2 MONTH) + INTERVAL 1 DAY
FROM sales;
```

The resulting output below confirms the success of the function across all test cases. Crucially, observe the entry for January 14, 2024. The calculation correctly determines that the first day of the previous month must roll back into the preceding calendar year, accurately returning December 1, 2023. This outcome solidifies the reliability of combining `LAST_DAY()` and INTERVAL arithmetic for precise monthly boundary calculations in **MySQL**.

Output of Date Calculation:

```
+------------+-----------------------------------------------------+
| sales_date | LAST_DAY(sales_date - INTERVAL 2 MONTH) + INTERVAL 1 DAY |
+------------+-----------------------------------------------------+
| 2024-02-10 | 2024-01-01 |
| 2024-11-25 | 2024-10-01 |
| 2024-06-30 | 2024-05-01 |
| 2024-01-14 | 2023-12-01 |
```

| 2024-05-19 | 2024-04-01 |
+------------+--+

Improving Data Readability with Column Aliases

Although the query successfully calculates the necessary date, the automatically generated column header, `LAST_DAY(sales_date - INTERVAL 2 MONTH) + INTERVAL 1 DAY`, is extremely verbose and unsuitable for production use, whether in application interfaces or complex reporting systems. To significantly enhance the clarity and maintainability of the result set, it is standard practice in professional SQL development to assign a descriptive alias using the **AS** keyword.

Using aliases provides a clean, meaningful identifier for the output data, effectively shielding users and downstream processes from the complexity of the underlying calculation logic. This transformation is particularly vital when the result set feeds into an ETL (Extract, Transform, Load) process or a front-end application that demands easily referenceable field names.

We modify the query by appending the alias `AS first_previous` directly after the date calculation formula. This simple refinement improves the query's readability and integration potential dramatically, aligning with best practices for data presentation in **MySQL** environments.

SELECT sales_date, LAST_DAY(sales_date - INTERVAL 2 MONTH) + INTERVAL 1 DAY AS first_previous
FROM sales;

The resulting output now features the concise, user-friendly column header, **first_previous**, confirming the successful application of the alias:

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

Detailed Breakdown: The Three-Step Logic

Understanding the internal mechanism of this date formula is essential for appreciating its robustness. The key to its accuracy lies in the deliberate use of three sequential operations, which bypass the complexities of calculating day offsets and fluctuating month lengths. We can illustrate this logic using our example date, **2024-02-10**, aiming for the result **2024-01-01**.

Step 1: Shift Back Two Months (`sales_date - INTERVAL 2 MONTH`):
The initial operation involves subtracting two full calendar months from the original date. For 2024-02-10, this calculation yields **2023-12-10**. We choose to subtract two months because our goal is not simply to land in the previous month, but to land in the month *before* that. This preparatory step ensures that when we apply the boundary finding function next, we target the correct endpoint.

Step 2: Find the Absolute Month End (`LAST_DAY(...)`):
Next, we apply the LAST_DAY() function to the result of Step 1 (2023-12-10). The function ignores the day component (10th) and returns the fixed end date of that month, which is **2023-12-31**. This action successfully establishes a hard, reliable boundary point--the very last day of the month preceding our target month.

Step 3: Advance to the First Day (`... + INTERVAL 1 DAY`):
In the final step, we add one day to the result of Step 2 (2023-12-31). Adding one day to the last day of December 2023 automatically rolls the date over to **2024-01-01**. This perfectly aligns with the first day of January 2024--the desired start date of the previous month relative to our original February input.

This systematic, three-part mechanism guarantees that the calculation is consistent and accurate, regardless of whether the input date is the 1st, the 15th, or the 31st of the starting month, successfully navigating all month length differences and year transitions seamlessly.

Alternative Methods and Concluding Recommendations

While the combination of the LAST_DAY() function and INTERVAL arithmetic is widely considered the most robust and elegant solution in **MySQL** for finding the start of the previous month, developers occasionally utilize alternative methods. For instance, one could use the `DATE_SUB()` function in conjunction with date formatting functions like `DATE_FORMAT()`. However, these alternatives often result in queries that are less intuitive and potentially harder to debug or maintain.

Another common approach involves calculating the first day of the current month first, then subtracting one day to find the end of the previous month, and finally calculating the start of that previous month. Despite these options, the method detailed in this guide--`LAST_DAY(date - INTERVAL 2 MONTH) + INTERVAL 1 DAY`--remains the recommended standard. It is favored by

experienced developers for its conceptual clarity, high efficiency, and guaranteed reliability when handling complex boundary conditions in large-scale database operations.

For any professional involved in business intelligence reporting or sophisticated database management, mastering this specific date manipulation technique is essential. By integrating this formula into your workflow, you ensure that all monthly rollups, financial summaries, and temporal analyses are anchored to precise, accurate calendar boundaries, fostering greater data integrity.

Additional Resources for MySQL Date Operations

For those looking to expand their knowledge of temporal functions and capabilities within MySQL, the following resources offer comprehensive guidance on related common tasks and functions:

The Official MySQL Date and Time Functions Documentation, providing exhaustive detail on all available functions.

A tutorial on selecting rows based on the current date:

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