How to Get the First Day of the Previous Month in MySQL

Authored by

mohammed looti

January 5, 2026

The calculation of specific calendar points, such as the first day of the previous month, is a frequent requirement in relational database management systems like MySQL. Although one might initially consider complex combinations of functions like `DATE_SUB()`, `DAYOFMONTH()`, and `CONCAT()`, these often lead to cumbersome or less efficient SQL statements. A cleaner, more elegant, and universally accepted solution relies on sequential date manipulation using the LAST_DAY() function combined with strategic use of the INTERVAL operator. This powerful combination allows developers to pinpoint the required date using logical steps of subtraction and addition, ensuring precision regardless of the starting date.

This comprehensive guide details the precise syntax required to extract the first day of the preceding month from any given date field within your database. We will demonstrate how to construct a reliable query that utilizes the inherent power of MySQL date functions, resulting in highly readable and maintainable code. Understanding this technique is fundamental for generating monthly reports, calculating billing cycles, or performing any time-series analysis that relies on consistent calendar boundaries. We will explore the mechanism behind this specific formula and provide detailed, runnable examples to illustrate its implementation in a real-world scenario.

Understanding the Core Syntax for Date Calculation

To accurately retrieve the first day of the month immediately preceding a specific input date in MySQL, we employ a sophisticated but straightforward arithmetic operation. This technique is often preferred because it avoids complex string manipulation and leverages MySQL's internal date handling capabilities, which are robust against month length variations and year changes. The foundational logic involves three sequential steps: first, shifting the date far enough back to guarantee landing in the previous month; second, identifying the last day of that shifted month; and finally, incrementing the date by one day to arrive at the desired first day.

The crucial part of this method lies in the combination of the LAST_DAY() function and the `INTERVAL` arithmetic. The `LAST_DAY()` function automatically returns the last day of the month for the date provided as its argument. By combining this with the `INTERVAL 2 MONTH` subtraction, we guarantee that we land two months prior to the original date. This ensures that when we calculate the last day of that resulting month, we are actually finding the last day of the month before the previous month. Adding one day back then cleanly shifts us to the first day of the desired previous month.

The standard syntax to achieve this, demonstrated using a hypothetical column named `sales_date` from a table named `sales`, is structured as follows. This query generates a new derived column alongside the original date, illustrating the precise outcome of the date calculation function. Pay close attention to the use of the INTERVAL operator for subtraction and addition,

which is the standard mechanism for date arithmetic in SQL environments.

You can use the following syntax to get the first day of the previous month for a given date in MySQL:

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

This particular example creates a new column that contains the first day of the previous month for the corresponding date in the **sales_date** column of the table named **sales**. This function chaining is the most reliable way to handle the complexities inherent in date boundary calculations within transactional data management systems.

Dissecting the Date Arithmetic Logic

To fully appreciate the efficacy of the demonstrated query, it is vital to break down the calculation into its discrete components. The formula `LAST_DAY(sales_date - INTERVAL 2 MONTH) + INTERVAL 1 DAY` appears abstract, but its power lies in forcing the date manipulation to utilize calendar end points. This approach guarantees that the resulting date will always be the first day (i.e., day '01') of the correct preceding month, irrespective of the specific day of the month (1st through 31st) provided in the original `sales_date` column.

The process starts with `sales_date - INTERVAL 2 MONTH`. By subtracting two months, we intentionally overshoot the desired previous month and land in the month before that. For instance, if the `sales_date` is March 15th, subtracting two months yields January 15th. We use this date purely as a reference point to ensure the subsequent function, `LAST_DAY()`, operates correctly on the month preceding our target. This initial subtraction is the foundational maneuver that sets up the rest of the calculation sequence.

The next operation is wrapping the result within the LAST_DAY() function. Continuing our example, applying `LAST_DAY('2024-01-15')` returns January 31st, 2024 (or 29th, 30th, etc., depending on the month). Critically, this result--the end of the month two periods ago--is also the day immediately preceding the first day of the previous month. This step successfully converts a variable date (e.g., the 15th) into a fixed calendar boundary (the last day of the reference month).

Finally, we add one day using `+ INTERVAL 1 DAY`. Taking January 31st, 2024, and adding one day results in February 1st, 2024. If the original date was March 15th, 2024, the result (February 1st, 2024) is precisely the first day of the previous month. This final addition is the clean-up step that locks the calculated date onto the required start-of-month boundary, fulfilling the primary objective of the SQL operation. This methodical approach ensures accuracy across all calendar transitions.

Practical Example: Setting Up the Sales Data

To illustrate this functionality clearly, we will work with a sample dataset representing transactional information. Suppose we manage a database containing sales records, and we need to calculate rolling monthly metrics based on the start of the previous period. For this demonstration, we create a simple table named `sales` which contains core information about items sold, including a unique identifier, the item description, and, most importantly, the date of the sale.

The structure of the `sales` table must be defined to include a column capable of storing date values accurately. We define `store_ID` as the primary key, `item` using the `TEXT` datatype, and `sales_date` utilizing the `DATE` datatype. The use of the standard `DATE` type is essential here, as the functions we employ (such as LAST_DAY() and `INTERVAL` arithmetic) are designed to interact seamlessly with this format within MySQL. This setup allows us to test the date calculation logic against various dates spread throughout the calendar year.

Below is the SQL code required to create the table structure and populate it with five sample rows. These rows contain dates distributed across different months and years (in the case of the January entry rolling back to December of the previous year), providing robust test cases for our date calculation query. We then include a standard `SELECT * ` statement to confirm the initial state of the dataset before applying the 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;
```

Output of the initial dataset:

```
+----------+---------+------------+
```

| 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 Calculation Query

The objective is to augment our existing sales data by calculating and displaying the first day of the month preceding each recorded `sales_date`. This requires running the specialized SQL query developed earlier, applying the date arithmetic directly within the `SELECT` clause. The output will demonstrate the effectiveness of using LAST_DAY() and the INTERVAL operator across different date values, including those that cross year boundaries.

We specifically instruct MySQL to select the original `sales_date` for reference, followed by the complex date expression. This expression is processed row by row, ensuring that for every entry in the `sales` table, the corresponding first day of the previous month is correctly derived. Note that when executing this initial query without an alias, MySQL automatically names the calculated column based on the full function call, which, while technically correct, can result in a lengthy and unreadable column header.

The query execution is performed using the exact syntax we defined previously. We are selecting both the original date and the result of the complex calculation from the `sales` table:

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

Output of the 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 |

+------------+--+

Upon reviewing the output, it is clear that the function operates precisely as intended. For the entry `2024-02-10`, the calculation correctly returns the first day of January 2024 (`2024-01-01`). Furthermore, the entry `2024-01-14` demonstrates the crucial year boundary handling, returning `2023-12-01`. This confirms the robustness of the methodology in handling complex date shifts, which is essential for consistent historical reporting across calendar years. The dates in the newly generated column invariably represent the start of the previous month relative to the corresponding `sales_date`.

Enhancing Output Readability Using the AS Clause

While the calculated output successfully delivers the required date, the automatically generated column header, `LAST_DAY(sales_date - INTERVAL 2 MONTH) + INTERVAL 1 DAY`, significantly hinders readability and usability, particularly when integrating this query into larger reports or applications. To rectify this common issue in SQL, we leverage the powerful `AS` keyword, which allows us to assign a concise, descriptive alias to the derived column.

The use of the `AS` statement is critical for generating clean, professional query results. By defining a meaningful alias, such as `first_previous`, we streamline the result set, making it instantly understandable for any analyst or developer consuming the data. This best practice not only improves presentation but also simplifies subsequent queries or joins that might reference this calculated field.

To implement this improvement, we append `AS first_previous` directly after the complex date expression in the `SELECT` statement. The following query demonstrates this syntax improvement:

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

The resulting output clearly shows the enhanced readability, confirming that the alias has been correctly applied without altering the calculated date values:

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

By using the `AS` clause, the new column is now concisely named **first_previous**. This practice makes the query result far more accessible and maintainable, which is a critical consideration in any professional database environment utilizing MySQL.

Mechanism Deep Dive: Why Subtract Two Months?

The most counter-intuitive aspect of this technique is the subtraction of two months (`INTERVAL 2 MONTH`) followed by the addition of only one day (`INTERVAL 1 DAY`). This specific structure is not arbitrary; it is a necessary workaround to consistently handle variations in month length and ensure the calculation always lands on the first day of the target month, regardless of the starting date's day-of-the-month value.

Consider a scenario where we attempted a simpler calculation, such as `LAST_DAY(sales_date - INTERVAL 1 MONTH) + INTERVAL 1 DAY`. If the original date were February 15th, subtracting one month yields January 15th. Applying LAST_DAY() gives January 31st. Adding one day results in February 1st. This seems correct, as the previous month was January. However, if the original date was March 15th, subtracting one month yields February 15th. Applying `LAST_DAY()` yields February 29th (in a leap year). Adding one day results in March 1st--the start of the **current** month, not the **previous** month (which should be February 1st).

By subtracting two months, we guarantee that the date we feed into the LAST_DAY() function is safely located in the month prior to our target previous month. This ensures that when `LAST_DAY()` calculates the end boundary, that boundary is precisely the day preceding the first day of the desired previous month. For instance, using our key example date, **February 10, 2024**:

The formula first executes the subtraction: `'2024-02-10' - INTERVAL 2 MONTH` results in **2023-12-10**.
Next, the `LAST_DAY()` function is applied to the result: `LAST_DAY('2023-12-10')` results in **2023-12-31**.
Finally, one day is added: `'2023-12-31' + INTERVAL 1 DAY` results in **2024-01-01**.

This systematic process ensures that we correctly identify January 1st, 2024, as the first day of the previous month (January) relative to February 10th, 2024. The two-month subtraction acts as a buffer, making the calculation robust against month transitions and differing month lengths, which is critical for accurate MySQL date manipulation.

Related Date Functions and Alternative Approaches

While the `LAST_DAY(X - INTERVAL 2 MONTH) + INTERVAL 1 DAY` method is often considered the gold standard for its reliability and directness in MySQL, other built-in functions can be combined to achieve similar results, often involving slightly more complex nesting or reliance on string manipulation. Understanding these related functions is crucial for developers working with diverse date requirements.

One notable alternative involves using the DATE_SUB() function, which was briefly mentioned in the original introduction. `DATE_SUB()` is highly effective for simple subtractions. To find the first day of the previous month using this approach, one might first use `DATE_SUB(CURDATE(), INTERVAL 1 MONTH)` to move to the previous month, and then use `DATE_FORMAT()` or similar string manipulation to force the day component to '01'. However, this usually involves converting the date into a string and back, potentially leading to performance bottlenecks or subtle format errors compared to the purely arithmetic approach demonstrated.

For finding the last day of the current month, which is an inverse problem often paired with this task, the LAST_DAY() function is used directly on the current date: `SELECT LAST_DAY(CURDATE())`. This highlights the foundational role of `LAST_DAY()` in defining monthly boundaries within MySQL. Mastering these functions enables precise control over time-based filtering and grouping necessary for advanced business intelligence reports.

A final point of reference is the use of `TRUNCATE()` or `DATE_TRUNC()` (if available in the specific MySQL version or flavor) for resetting date components. While not always directly applicable to "previous month" calculations, functions that truncate or format dates to the beginning of a month (e.g., `DATE_FORMAT(CURDATE(), '%Y-%m-01')`) provide a conceptual framework for date boundary creation. Ultimately, the `LAST_DAY()` method remains the most straightforward path for deriving the specific "first day of the previous month" date object required by our task.

Further Resources for MySQL Date Operations

Mastering date and time manipulation is fundamental to effective data management and reporting. The techniques showcased here serve as a solid foundation for more complex time-series analysis and filtering operations. We encourage further exploration of related functions to enhance your SQL proficiency.

The following tutorials explain how to perform other common tasks in MySQL, complementing the knowledge gained regarding month boundary calculations:

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