How to Get the First Day of the Month in MySQL

Authored by

mohammed looti

January 5, 2026

Understanding Date Manipulation in MySQL

When working with relational databases, manipulating date and time information is a fundamental requirement for reporting, filtering, and aggregation tasks. In the context of MySQL, one of the most frequent date calculations involves determining the start date of the current calendar period. Specifically, finding the first day of the month for a given input date is essential for summarizing monthly sales figures, cohort analysis, or generating fiscal reports. Although MySQL does not have a single, dedicated function like `FIRST_DAY_OF_MONTH()`, achieving this result is straightforward by combining several powerful built-in date functions.

The concept of the first day of the month refers to the initial day of the month corresponding to a specific DATE value stored in a table column. For instance, if a record has a transaction date of '2024-05-19', the first day of that month would be '2024-05-01'. This conversion is necessary because it standardizes the time frame, allowing developers and analysts to easily group records regardless of the specific day they occurred. Without this standardization, complex queries involving range comparisons would be required, leading to decreased performance and maintenance difficulty.

To perform such intricate date calculations efficiently, MySQL provides a rich set of functions. The primary tools we will employ here are `DATE_ADD()` and `DAY()`. The `DAY()` function extracts the numeric day of the month (1 to 31) from a date, while `DATE_ADD()` allows us to adjust a date by adding or subtracting a specified time interval. By creatively utilizing these functions, we can devise a reliable and universally applicable method to pinpoint the start of any month represented in your datasets, providing flexibility for sophisticated data processing using SQL.

The Core Logic: Calculating the First Day of the Month

The mathematical logic behind deriving the first day of the month is remarkably simple, though its implementation in MySQL requires understanding how time intervals are handled. The core idea is to determine how many days must be subtracted from the input date to arrive at the 1st of that month. If today is the 10th day of the month, we need to subtract 9 days (10 - 1). If today is the 1st, we need to subtract 0 days (1 - 1). This relationship can be generalized using the `DAY()` function.

The `DAY(date_expression)` function returns an integer representing the day of the month (e.g., `DAY('2024-06-30')` returns 30). To calculate the number of days we need to remove, we take the current day number and subtract one. This calculation, `DAY(sales_date) - 1`, yields the exact offset from the 1st of the month. We then use the powerful DATE_ADD function to perform the subtraction, specifying the interval in days. However, since `DATE_ADD()` works by adding, we subtract the calculated interval by making the value negative.

While the calculation `DAY(date) - 1` gives the offset to subtract, the common and often clearer

structure used in SQL is to subtract the current day number and then add back one day. This results in the expression `-DAY(sales_date) + 1` being passed to the `INTERVAL` clause of `DATE_ADD`. For example, if the date is the 10th (DAY=10), the expression becomes `-10 + 1 = -9`. This means we are adding a negative interval of 9 days, effectively subtracting 9 days and landing precisely on the first day of the month. This standardized approach ensures that the calculation handles all days from the 1st to the 31st without conditional logic.

Syntax Breakdown: Utilizing DATE_ADD and DAY Functions

To retrieve the first day of the month for any given date column in MySQL, we combine the principles discussed above into a single, compact SQL statement. The key is the strategic use of the DATE_ADD function, which is designed for flexible date arithmetic. The function requires three arguments: the starting date, the interval value, and the unit of the interval.

The syntax used to achieve this calculation involves embedding the `DAY()` function within the interval parameter of `DATE_ADD()`. This allows the interval calculation to be dynamic, adjusting based on the input date for every row in the table. The following syntax is the standard and most efficient way to achieve this goal in MySQL.

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

SELECT sales_date, DATE_ADD(sales_date, INTERVAL -DAY(sales_date)+1 DAY)
FROM sales;

This particular example creates a new column that contains the first day of the month for the corresponding date in the **sales_date** column of the table named **sales**. The expression `INTERVAL -DAY(sales_date)+1 DAY` dynamically calculates the negative offset needed. For example, if `sales_date` is '2024-02-10', `DAY(sales_date)` returns 10. The interval calculation becomes `-10 + 1 = -9`. DATE_ADD then subtracts 9 days from February 10th, resulting in February 1st, successfully identifying the first day of that month.

This method is highly scalable and ensures precise calculation across diverse date ranges and leap years, as it relies on the internal date arithmetic engine of MySQL. Understanding this combination of functions is critical for advanced time-series analysis in any relational database environment.

Practical Demonstration: Setting up the Sales Table

To demonstrate this functionality clearly, we will use a hypothetical scenario involving sales data. Imagine we operate a retail business and track daily transactions in a table called `sales`. This table

contains essential identifying information, the item sold, and, crucially, the DATE of the transaction. Our goal is to derive the corresponding month-start date for each transaction, allowing us to summarize monthly performance easily.

We begin by creating the necessary table structure and populating it with sample data representing different dates across several months of the year 2024. This setup allows us to verify that our calculation works correctly across various day values (beginning, middle, and end of the month). The structure includes columns for `store_ID` (primary key), `item` (text description), and `sales_date` (the date of the transaction).

The following example shows how to define the schema and populate the table using standard SQL commands. We use the `CREATE TABLE` and multiple `INSERT INTO` statements to establish a representative dataset that simulates real-world usage patterns, preparing us for the date manipulation query.

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:

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

The resulting table confirms that our dataset is ready for querying. We have five distinct records, each with a `sales_date` that needs normalization to the first day of its respective month. This preparatory step is vital in any data manipulation task, ensuring that the initial conditions are clear before applying complex functions.

Executing the Calculation Query

With the `sales` table established, we can now apply the previously discussed `DATE_ADD` methodology to generate the required first-day-of-month column. Our objective is to select the original `sales_date` alongside a new calculated column that represents the standardized month start. This transformation allows immediate visual confirmation that the date arithmetic is working as intended across all rows.

The query utilizes the specific formula: `DATE_ADD(sales_date, INTERVAL -DAY(sales_date) + 1 DAY)`. DATE_ADD dynamically calculates the required offset for each row. For the date '2024-06-30', for example, `DAY()` returns 30. The interval becomes -29 days, and adding -29 days to June 30th correctly yields June 1st. This mechanism is applied iteratively across all records in the `sales` table.

Executing the query demonstrates how efficiently MySQL handles date arithmetic. The resulting output clearly shows the transformation, confirming that every transaction date has been mapped back to the beginning of its originating month. This derived column is invaluable for subsequent aggregations, such as using the `GROUP BY` clause to summarize total sales per month.

Suppose that we would like to select the dates from the **sales_date** column and create a new column that contains the first day of the month for each corresponding date in the **sales_date** column.

We can use the following syntax to do so:

SELECT sales_date, DATE_ADD(sales_date, INTERVAL -DAY(sales_date)+1 DAY)
FROM sales;

Output:

```
+------------+-----------------------------------------------------+
| sales_date | DATE_ADD(sales_date, INTERVAL -DAY(sales_date)+1 DAY) |
+------------+-----------------------------------------------------+
| 2024-02-10 | 2024-02-01 |
| 2024-11-25 | 2024-11-01 |
| 2024-06-30 | 2024-06-01 |
| 2024-01-14 | 2024-01-01 |
| 2024-05-19 | 2024-05-01 |
+------------+-----------------------------------------------------+
```

Notice that the dates in the new column represent the first day of the month for the corresponding date in the **sales_date** column. The output confirms the accurate calculation for all input dates, demonstrating the successful application of the dynamic interval calculation within DATE_ADD.

Enhancing Readability with Aliases (The AS Statement)

While the calculated output above is functionally correct, the column name generated by MySQL (`DATE_ADD(sales_date, INTERVAL -DAY(sales_date)+1 DAY)`) is verbose and impractical for use in application layers or complex subqueries. In professional SQL development, maintaining readability is paramount, and this is where the `AS` statement comes into play. The `AS` keyword allows us to assign a user-friendly alias to a calculated column or an expression.

By assigning an alias such as `first_day` to the calculated field, we transform the output header into something meaningful and easily referenceable. This practice drastically improves code maintainability and clarity, particularly when the calculation itself is complex or involves multiple nested functions. This simple addition makes the query results immediately accessible to anyone reviewing the output or integrating it into further analysis.

Applying the `AS` statement to our previous query results in a cleaner, more professional output. This final refinement ensures that while the underlying logic is complex, the exposed data is simple and intuitive. This technique is mandatory for any production-level report or query designed for consumption outside of a debugging environment.

If you'd like, you can also use the **AS** statement to give a specific name to this new column:

```
SELECT sales_date, DATE_ADD(sales_date, INTERVAL -DAY(sales_date)+1 DAY) AS first_day
FROM sales;
```

```
+------------+------------+
| sales_date | first_day |
+------------+------------+
| 2024-02-10 | 2024-02-01 |
| 2024-11-25 | 2024-11-01 |
| 2024-06-30 | 2024-06-01 |
| 2024-01-14 | 2024-01-01 |
| 2024-05-19 | 2024-05-01 |
+------------+------------+
```

Notice that the new column is now named **first_day**, which is much easier to read. This simplified naming convention greatly facilitates subsequent data analysis tasks, such as joining this result set with other tables or using the column in filtering criteria.

Exploring Alternative Methods for Date Calculations

While the `DATE_ADD(..., INTERVAL -DAY(...) + 1 DAY)` method is robust and widely used in MySQL due to its reliance on basic arithmetic and interval handling, developers often seek alternative methods, especially in scenarios involving version compatibility or specific performance optimizations. One common alternative involves using the `DATE_FORMAT()` and `STR_TO_DATE()` functions, which prioritize string manipulation for date normalization. This method involves converting the date to a string format representing only the year and month, then converting it back to a DATE type using the fixed day '01'.

For example, the alternative approach would look like `STR_TO_DATE(DATE_FORMAT(sales_date, '%Y-%m-01'), '%Y-%m-%d')`. First, `DATE_FORMAT()` extracts the year (`%Y`) and month (`%m`) from the date and concatenates them with the fixed string '01'. Then, `STR_TO_DATE()` parses this new string back into a valid DATE object. While this method is highly readable and conceptually simple, it often involves more overhead than the arithmetic calculation using DATE_ADD, as it relies on expensive string conversions.

Another specialized function is `LAST_DAY()`, which returns the last day of the month for a given date. While `LAST_DAY()` does not directly give the first day, it can be combined with `DATE_ADD()` (or `DATE_SUB()`) to calculate the interval between the last day and the first day, thereby deriving the start date. However, the initial method utilizing `-DAY() + 1` remains the most direct and idiomatic way to solve the specific requirement of finding the first day of the month in MySQL, balancing both clarity and execution efficiency across different query environments.

Further Date and Time Resources in MySQL

Mastering date and time manipulation is crucial for any developer or analyst working extensively with database systems. The concepts demonstrated here, particularly the use of interval arithmetic via DATE_ADD and SQL functions like `DAY()`, are transferable to many other common tasks, such as finding the last day of the previous quarter or calculating the number of workdays between two dates. We encourage the exploration of related tutorials to build a comprehensive skill set in time-series data handling.

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

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