Learning MySQL: Extracting the First Day of the Month from Dates

Authored by

Mohammed looti

November 12, 2025

Mastering Date Arithmetic in MySQL

Effective management of temporal data is a core requirement for almost every professional database system. Database administrators and business analysts frequently encounter the need to standardize temporal markers, particularly when aggregating data for monthly reports or performing complex time-series analysis. A common, yet critical, task is determining the first day of the month corresponding to any given transactional date stored in the database. Achieving this efficiently is paramount for systems relying on the powerful **MySQL** relational database. While Standard **SQL** offers various methods, leveraging MySQL's native date and time functions provides the most robust and high-performing solution.

The technique detailed in this guide involves a precise arithmetic calculation that determines the exact offset required to shift any date back to the first day of its month. This method is highly reliable, automatically accommodating complexities such as leap years and varying month lengths, thereby guaranteeing accuracy regardless of the input date. Developers who master this fundamental operation can write exceptionally clean and performant queries, avoiding the pitfalls of less optimized string manipulation strategies. Furthermore, utilizing native functions like this demonstrates a sophisticated understanding of how the **DATE** data type and temporal arithmetic are handled internally by the database engine.

To retrieve the first day of the month for records within your tables, we employ a specific syntax that is both highly portable and standardized across different versions of MySQL. This approach remains the recommended best practice for date calculation tasks due to its inherent efficiency and clarity. The following query illustrates the core structure, assuming we are working with a column named `sales_date`:

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

This succinct yet powerful query generates a new calculated column. For every row processed in the `sales` table, the logic applies arithmetic subtraction encapsulated within the **DATE_ADD** function. Understanding the inner workings of this formula is essential for leveraging its full potential, and we will now meticulously dissect how these components cooperate to accurately pinpoint the beginning of the month.

Deconstructing the Arithmetic: DATE_ADD, DAY(), and the Offset

The entire solution is fundamentally built upon two essential **MySQL** date functions: **DATE_ADD** and **DAY**. The **DATE_ADD** function is the mechanism for performing date arithmetic, allowing developers to add or subtract a specified time **INTERVAL** from a starting date expression. Its

standard syntax requires a date, the keyword `INTERVAL`, a value, and a unit (e.g., DAY, MONTH, YEAR). Conversely, the **DAY** function serves a simpler purpose: extracting the integer representing the day of the month (which ranges from 1 to 31) from the provided date.

The core logic of the query relies entirely on calculating the precise number of days that must be subtracted. Consider an input date, where we know the current day number (let's call it N). To reach the first day of the month (Day 1), we must subtract N - 1 days. For example, if the date is the 20th (N=20), we subtract 19 days. If the date is the 1st (N=1), we subtract 0 days. This simple mathematical relationship is perfectly modeled by the arithmetic expression embedded within the `INTERVAL` clause: `-DAY(sales_date) + 1 DAY`.

To solidify this concept, let us walk through a concrete example using the date `2024-02-10`. This three-step process clearly demonstrates how the formula achieves its objective through backward calculation:

The **DAY** function first isolates the day number: `DAY('2024-02-10')` returns the integer **10**.

Next, the expression calculates the total offset required: `-10 + 1`, resulting in **-9** days to subtract.

Finally, the **DATE_ADD** function applies this offset: `DATE_ADD('2024-02-10', INTERVAL -9 DAY)`.

The final result of this operation is `2024-02-01`, which is the correct first day of February 2024. This systematic process confirms the reliability and mathematical rigor of using native MySQL date functions for temporal calculations.

Practical Application: Setting up the Sample Data Environment

To provide a tangible illustration of this date manipulation technique, we will establish a simplified, hypothetical database table named `sales`. This table will contain transactional data, specifically recording the date of various grocery sales. Defining this clear, structured dataset is essential, as it allows us to execute the **SQL** query and visually confirm the results, ensuring a complete understanding of the practical impact of the date formula on real-world data.

The following SQL commands are used to define the schema and populate our sample environment. It is crucial to note that the `sales_date` column is explicitly defined using the **DATE** data type, which is mandatory for the date functions to operate correctly and efficiently. We have deliberately included a diverse set of dates spanning different months and days within 2024 to rigorously test the accuracy and consistency of our calculation across various scenarios.

-- 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;
```

The output below confirms that the table has been successfully created and populated with the specified data. This foundational step provides the necessary context for our date extraction exercise, clearly presenting the original sales dates that we aim to transform into their corresponding month-start dates.

Output of Sample Data:

```
+----------+---------+------------+
| 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 and Validating Results

With the sample data established, our primary goal is to execute the date calculation query. We need to retrieve the original `sales_date` and generate a calculated column containing the first day of the month for each respective entry. As established, this is achieved by instructing **MySQL** to use the arithmetic combination of the **DATE_ADD** function and the **DAY** function. This dynamic approach ensures that the precise number of days required to roll back to the first of the month is calculated automatically for every single record, maximizing efficiency and minimizing potential

human error.

We now execute the core query against the `sales` table:

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

The resulting output below provides clear evidence of the formula's effectiveness. Every date generated in the new, calculated column accurately corresponds to the beginning of the month for the original `sales_date` entry. For instance, the sale on `2024-11-25` is correctly standardized to `2024-11-01`, and the record for `2024-06-30` is appropriately mapped to `2024-06-01`. This verification step confirms that the formula consistently and accurately handles dates regardless of their position within the month.

Output of Date Calculation:

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

Improving Query Clarity with Column Aliases

While the previous query successfully achieves the desired temporal calculation, the column name generated by the expression, `DATE_ADD(sales_date, INTERVAL -DAY(sales_date)+1 DAY)`, is verbose and impractical for professional reporting or integration into application logic. A foundational principle of clean and maintainable **SQL** practice is the use of meaningful identifiers for derived outputs. We achieve this critical step by assigning an alias to the computed expression using the **AS** keyword.

By introducing the **AS** clause, we transform the complex functional output into a simple, intuitive column header, such as `first_day`. This minor modification yields a significant improvement in code readability and maintainability, allowing subsequent developers or end-users to immediately grasp the column's purpose without needing to interpret the underlying date arithmetic. This step is indispensable for creating production-ready queries suitable for large-scale data workflows.

We revise the query to incorporate the `AS first_day` alias:

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

The final output demonstrates the successful application of the alias, resulting in a standardized and descriptive column header. This clean format ensures that the results are easily consumed by reporting tools or integrated into application backends, solidifying the professional quality of the query.

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

Comparing the Preferred Method to Alternative Date Calculations

Although the **DATE_ADD** and **DAY** combination is widely regarded as the most performant and semantically superior approach for determining the first day of the month in **MySQL**, database developers should be aware of alternative methods. These alternatives often involve trade-offs related to version compatibility, query complexity, and execution speed. For instance, the `DATE_SUB` function can achieve the same arithmetic result as `DATE_ADD` (by using a positive interval value for subtraction), but its use offers no substantial performance advantage over the primary method presented here.

A different, commonly seen alternative relies heavily on string manipulation functions. This method typically uses `DATE_FORMAT` to extract the year and month components as a string, manually concatenates '01' to force the day, and then uses `STR_TO_DATE` to convert the resulting string back into a valid **DATE** type. While this approach, often looking like `STR_TO_DATE(CONCAT(DATE_FORMAT(sales_date, '%Y-%m-'), '01'), '%Y-%m-%d')`, is quite readable, the sequential conversion of data types (Date to String to Date) introduces significant performance overhead. When processing vast volumes of data, this string-based technique is noticeably less efficient than the purely arithmetic calculation that operates natively on date types.

For users operating on newer versions of MySQL (8.0 and above), other complex formulas involving functions like `LAST_DAY()` exist, but they generally add unnecessary complication to this specific problem. The consensus among expert database professionals is clear: the expression `DATE_ADD(date, INTERVAL -DAY(date) + 1 DAY)` represents the optimal strategy, balancing high performance, absolute accuracy, and excellent compatibility across the broadest range of MySQL installations.

Further Resources for MySQL Date Function Mastery

Developing expertise in manipulating date and time data is a cornerstone of effective database management and business intelligence. To continue enhancing your skills in handling temporal data within **SQL**, we recommend exploring additional documentation and tutorials covering other frequently required time-based operations in **MySQL**:

How to Select Rows where Date is Equal to Today (This is a placeholder for the original link, maintaining its structure).
MySQL: How to Select Rows where Date is Equal to Today
Official MySQL Documentation on Date and Time Functions.