How to Get the First Day of a Quarter in MySQL

Authored by

mohammed looti

January 5, 2026

Accurately managing and calculating chronological periods is a fundamental requirement in database management, particularly when dealing with large datasets relating to business cycles, such as sales or financial reporting. A common requirement for analysts and developers working with MySQL is determining the exact start date of a specific fiscal quarter based on any given date within that period. While manually calculating these dates using hardcoded days (like 0, 91, 182, or 273) might seem feasible, this method is highly inefficient and prone to errors, especially when factoring in leap years or needing dynamic calculations across various time stamps. Therefore, leveraging built-in MySQL date and time functions provides a robust and reliable solution for obtaining the first day of any quarter.

The standard methodology in MySQL involves a sophisticated combination of functions that first identify the year and the corresponding quarter number, and then use interval arithmetic to rewind the date to the quarter's beginning. This approach utilizes powerful functions like MAKEDATE, QUARTER, and the INTERVAL keyword. By understanding how these components interact, database professionals can create highly efficient and accurate SQL queries capable of handling complex temporal calculations without relying on pre-calculated day counts or cumbersome conditional logic. The resulting syntax is remarkably compact yet powerful, yielding precise results across different years and dates, making it an essential technique for reporting dashboards and aggregations.

The Robust Formula for Quarter Start Dates

The primary challenge in calculating the quarter's start date is ensuring the result accounts for the transition across quarters, while remaining firmly rooted in the correct calendar year. The most effective and widely accepted method in the MySQL community bypasses complex conditional statements by combining date generation with temporal subtraction. This technique first identifies the beginning of the year, advances the date by the required number of quarters, and then corrects this advancement by stepping back one full quarter. This clever maneuver lands the calculated date precisely on the first day of the current quarter.

This method relies fundamentally on three steps executed within a single SELECT statement. First, we anchor the calculation to January 1st of the relevant year, which is easily extracted using the YEAR and MAKEDATE functions. Second, we determine which quarter the original date falls into using the powerful QUARTER function. Third, we manipulate the date using the INTERVAL keyword to move forward by the current quarter number and then backward by one quarter unit. This ensures that even if the date is in Q1, the calculation remains mathematically sound without requiring special handling for the first three months of the year.

Core Functions and Their Roles in Calculation

To fully appreciate the efficiency of the calculation, it is crucial to understand the purpose of each function involved. The MAKEDATE function is instrumental here, as it constructs a date based on the year and the day number within that year. By calling **MAKEDATE(YEAR(date), 1)**, we instantaneously obtain January 1st of the year corresponding to the input date. This establishes the absolute starting point for all subsequent temporal arithmetic, providing a consistent baseline regardless of whether we are in Q1 or Q4.

The QUARTER function is equally vital, returning an integer from 1 to 4 representing the quarter of the year for a given date. If the date is July 30th, **QUARTER()** returns 3. This integer value is then used directly with the INTERVAL keyword to dynamically calculate how many quarters forward we need to advance from the start of the year. For instance, if the quarter is 3, we add 3 quarters to January 1st, landing us on October 1st (the start of Q4). This seems counterintuitive but sets up the final subtraction step perfectly.

Finally, the INTERVAL keyword facilitates advanced date arithmetic. By adding **INTERVAL QUARTER(sales_date) QUARTER**, we jump ahead to the first day of the quarter following the current one. The subsequent subtraction, **- INTERVAL 1 QUARTER**, then shifts the date backward exactly three months, thus aligning the result precisely with the first day of the original date's quarter. This combination of addition and subtraction, utilizing the quarter number itself, provides a highly mathematical and leap-year-safe method for temporal manipulation.

The Complete Syntax for Derivation

The entire calculated logic is consolidated into a single, clean expression within a SELECT statement. This expression takes the raw date field, performs the required decomposition and recomposition steps, and assigns the resulting quarter start date to a new, user-defined column. This is the standard, optimized method that efficiently processes large tables without requiring stored procedures or complex user-defined functions.

You can use the following syntax to get the first day of the quarter for a given date in MySQL. Note how the various functions are nested and combined using arithmetic operators:

```
SELECT sales_date, MAKEDATE(YEAR(sales_date), 1) +
INTERVAL QUARTER(sales_date) QUARTER -
INTERVAL 1 QUARTER AS first_day
FROM sales;
```

This particular example creates a new column named **first_day** that contains the first day of the

quarter for the corresponding date in the **sales_date** column of the table named **sales**. The elegance of this solution lies in its brevity and its ability to dynamically handle any date input, relying only on standard built-in SQL functions and operators, ensuring maximum compatibility and performance across various MySQL versions.

Practical Application: Setting Up Sample Data

To demonstrate this powerful date calculation in a real-world context, we will use a sample table named **sales**. This table simulates transactional data, where each row contains a record of a sale, including a unique identifier, the item sold, and most importantly, the date of the transaction. Accurate reporting often requires aggregating sales data by fiscal quarter, making the determination of the quarter's start date critical for grouping records efficiently.

Suppose we have the following table named **sales** that contains information about sales made at various grocery stores on various dates. We will use standard Data Definition Language (DDL) and Data Manipulation Language (DML) commands to set up this environment:

```
-- 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', '2024-07-30');
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 resulting table structure contains sales dates spanning all four quarters of the year 2024 (a leap year), providing a perfect testbed for our quarter calculation query. We have dates in February (Q1), May (Q2), July (Q3), and November (Q4). The initial output confirms the successful creation and population of the table, ready for the analytical query:

Output of the Table:

```
+----------+---------+------------+
| store_ID | item | sales_date |
+----------+---------+------------+
| 1 | Oranges | 2024-02-10 |
| 2 | Apples | 2024-11-25 |
| 3 | Bananas | 2024-07-30 |
| 4 | Melons | 2024-01-14 |
| 5 | Grapes | 2024-05-19 |
+----------+---------+------------+
```

Executing the Calculation Query

The next step involves applying the derived SQL formula to the **sales_date** column. 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 quarter for each corresponding date in the **sales_date** column. This new column, aliased as **first_day**, will be invaluable for subsequent aggregation and reporting tasks, allowing us to easily group all sales that occurred within the same fiscal quarter.

We use the exact syntax discussed previously. Note that the entire logic is contained within the SELECT statement, demonstrating the power of date manipulation within MySQL's declarative language. The expression effectively transforms the dynamic date into a standardized anchor date for quarterly analysis:

```
SELECT sales_date, MAKEDATE(YEAR(sales_date), 1) +
INTERVAL QUARTER(sales_date) QUARTER -
INTERVAL 1 QUARTER AS first_day
FROM sales;
```

The execution of this query yields a dataset where every transactional date is paired with its calculated quarter start date. This immediate and accurate result validates the mathematical structure of the formula, confirming its utility for handling complex time-series data reporting requirements. The output clearly shows the mapping from the specific sale date to its standardized quarterly boundary.

Output of the Query:

```
+------------+------------+
| sales_date | first_day |
+------------+------------+
| 2024-02-10 | 2024-01-01 |
```

| 2024-11-25 | 2024-10-01 |
| 2024-07-30 | 2024-07-01 |
| 2024-01-14 | 2024-01-01 |
| 2024-05-19 | 2024-04-01 |
+------------+------------+

Analyzing the Results and Validation

Observe the dates in the **first_day** column; they precisely represent the first day of the quarter for the corresponding date in the **sales_date** column. This dynamic calculation avoids the pitfalls of fixed-day addition and correctly manages the shifting nature of calendar quarters. The results demonstrate the query's ability to consistently assign the correct quarterly boundary across different periods of the year.

For detailed validation, we can review the mappings based on the standard calendar quarters (Q1: Jan 1 - Mar 31; Q2: Apr 1 - Jun 30; Q3: Jul 1 - Sep 30; Q4: Oct 1 - Dec 31):

The sale on 2024-02-10 (February) falls in Q1. The calculated first day of the quarter is **2024-01-01**.
The sale on 2024-11-25 (November) falls in Q4. The calculated first day of the quarter is **2024-10-01**.
The sale on 2024-07-30 (July) falls in Q3. The calculated first day of the quarter is **2024-07-01**.
The sale on 2024-05-19 (May) falls in Q2. The calculated first day of the quarter is **2024-04-01**.

This validation confirms that the complex formula--which combines year extraction, date construction, quarter identification, and precise interval arithmetic--is the definitive method for accurately deriving the first day of a quarter in MySQL. By relying on native functions and standard temporal units (like **QUARTER**), the solution remains scalable and accurate regardless of underlying date configurations or calendar specifics, such as leap years.

Summary and Conclusion

Mastering date arithmetic is essential for any advanced database developer or data analyst. The challenge of finding the first day of a fiscal quarter is elegantly solved in MySQL by leveraging a specific sequence of built-in functions. Rather than attempting to calculate complex day offsets that are vulnerable to leap year variations, the preferred methodology utilizes the temporal precision afforded by the INTERVAL keyword in conjunction with the QUARTER and MAKEDATE functions. This approach ensures maximal accuracy and efficiency across all datasets.

The resulting SQL syntax is highly portable and serves as a powerful tool for standardizing

temporal data. By implementing this formula, users can quickly transform raw transactional dates into quarter-start benchmarks, significantly simplifying group-by operations and enhancing the clarity of time-series reports. This technique represents a best practice for handling quarterly aggregations within the MySQL environment.

Related MySQL Tutorials

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

The first day of the quarter for the date 2/1/2024 is **1/1/2024**.
The first day of the quarter for the date 11/25/2024 is **10/1/2024**.
The first day of the quarter for the date 7/30/2024 is **7/1/2024**.

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