How to Select MySQL Rows with Dates in the Last 30 Days

Authored by

mohammed looti

January 5, 2026

When working with large datasets in a database system like MySQL, one of the most common requirements is filtering records based on time--specifically, retrieving data that falls within a recent period. Whether analyzing recent customer activity, checking the last month's sales figures, or reviewing newly created logs, efficiently selecting rows based on a date range is a fundamental SQL skill. This guide provides a comprehensive breakdown of how to construct a robust query using built-in MySQL functions to isolate records where the designated date field is within the last 30 days, ensuring your data analysis is timely and relevant.

The process leverages time arithmetic within the WHERE clause to compare the stored date value against a dynamically calculated threshold. This dynamic approach means the query always executes relative to the current moment, ensuring accuracy regardless of when the query is run. We will detail the exact syntax and explain the crucial date and time functions that make this filtering technique possible, moving beyond simple equality checks to sophisticated time-series analysis.

The Essential Syntax for 30-Day Filtering in MySQL

To isolate data points from the last 30 days, we must formulate an SELECT statement that calculates a cutoff date and then retrieves all rows whose date column is greater than that cutoff. This method avoids hardcoding dates, making the query maintenance-free and perpetually relevant. The core idea is to use the current time as a baseline and subtract the desired time period--in this case, 30 days--to establish the beginning of the required window.

The standard syntax utilizes the NOW() function, which returns the server's current date and time, combined with the INTERVAL keyword, which specifies a unit of time to be added or subtracted. When applied correctly within the WHERE clause, this combination creates an efficient comparison predicate. Below is the fundamental structure applicable to any table containing a date column:

```
SELECT *
FROM sales
WHERE sales_date > NOW() - INTERVAL 30 DAY;
```

In this specific example, we are querying the table named **sales**. The SELECT * command retrieves all columns. The critical filtering happens in the WHERE clause, instructing MySQL to only return records where the value in the **sales_date** column is strictly greater than (>) the date 30 days ago. This effectively confines the results to the rolling 30-day window leading up to the moment the query is executed.

Deconstructing the Key MySQL Date Functions

A thorough understanding of the functions used is vital for mastering date manipulation in MySQL. The power of this query lies in its ability to handle time dynamically without manual calculation or intervention. The two primary elements responsible for this functionality are NOW() and the INTERVAL operator, which work in tandem to define the exact temporal boundary.

The NOW() function is straightforward: it returns the current date and time of the server, including hours, minutes, and seconds. It is essential to remember that since **NOW()** returns a DATETIME value, the comparison works precisely, down to the second. If your column (like **sales_date**) is defined as DATETIME, this high precision is maintained. If your column is only a DATE type, MySQL automatically handles the implicit conversion, usually performing the comparison correctly against midnight of the specified date.

The INTERVAL keyword is used to perform arithmetic operations on date and time values. It requires a numerical value and a unit (e.g., DAY, HOUR, MONTH, YEAR). In our query, `INTERVAL 30 DAY` creates a time duration of 30 days. When this duration is subtracted from `NOW()`, the resulting calculated date becomes the starting point of our desired 30-day period. This calculation, `NOW() - INTERVAL 30 DAY`, yields the exact date and time 30 days prior to the execution of the SQL statement.

Setting Up the Demonstration Environment: The Sales Table

To illustrate the filtering mechanism effectively, we will simulate a common database scenario involving sales transactions. We need a table structure that accurately captures transaction details alongside a precise timestamp. The table, named `sales`, will contain identifiers, product information, and the critical sales date field, defined using the DATETIME data type to ensure we capture both the date and time of the transaction.

The following SQL script defines the structure for our demonstration table. Note the use of the `sales_date` column, which is the target of our filtering operation. Defining it as DATETIME allows us to demonstrate filtering with maximum precision, though the logic is largely identical for columns defined simply as DATE.

```
-- create table
CREATE TABLE sales (
store_ID INT PRIMARY KEY,
item TEXT NOT NULL,
sales_date DATETIME NOT NULL
);
```

This structure provides a unique identifier (`store_ID`), a description of the item sold (`item`), and

the timestamp of the sale (`sales_date`). The combination of these fields represents a simple yet realistic transactional log, making it perfect for demonstrating temporal filtering techniques in MySQL. Once the table is created, we can proceed to populate it with diverse data points, ensuring that some dates fall within the last 30 days and others are significantly older, allowing us to test the query's precision.

Populating the Sample Data for Testing

To rigorously test the 30-day filtering query, we need sample data that spans various dates, ensuring a mix of recent and archival records. For the purpose of this demonstration, we assume that the current date (when the query is executed) is **February 12, 2024**. Our data set includes entries from 2009, 2020, 2023, and two entries from early 2024. This deliberate date distribution will clearly show which records are successfully selected by the rolling 30-day window.

```
-- insert rows into table
INSERT INTO sales VALUES (0001, 'Oranges', '2024-02-10 03:45:00');
INSERT INTO sales VALUES (0002, 'Apples', '2020-11-25 15:25:01');
INSERT INTO sales VALUES (0003, 'Bananas', '2009-06-30 09:01:39');
INSERT INTO sales VALUES (0004, 'Melons', '2024-01-14 03:29:55');
INSERT INTO sales VALUES (0005, 'Grapes', '2023-05-19 23:10:04');
```

```
-- view all rows in table
SELECT * FROM sales;
```

Executing the final SELECT statement above displays the complete table before filtering. This full output serves as our baseline, confirming all data points are present before applying the time constraint. Reviewing the initial output ensures that our filtering logic will operate on the intended data set and that the subsequent filtered results accurately reflect the imposed temporal limitations.

Output:

```
+----------+---------+---------------------+
| store_ID | item | sales_date |
+----------+---------+---------------------+
| 1 | Oranges | 2024-02-10 03:45:00 |
| 2 | Apples | 2020-11-25 15:25:01 |
| 3 | Bananas | 2009-06-30 09:01:39 |
| 4 | Melons | 2024-01-14 03:29:55 |
| 5 | Grapes | 2023-05-19 23:10:04 |
+----------+---------+---------------------+
```

Executing the 30-Day Filtering Query

We are now ready to apply the filtering logic developed earlier to our sample data set. As established, this article is written assuming a current date of **February 12, 2024**. Therefore, the query will calculate the cutoff date, which is 30 days prior. The cutoff point is midnight on January 13, 2024, if we consider only the date component, or more precisely, the exact time 30 days ago, ensuring only records from 03:45:00 on January 13, 2024, onwards are included (depending on the exact time NOW() is executed).

Our objective is to select all rows where the **sales_date** is greater than the date 30 days ago. This is achieved by directly embedding the date arithmetic within the WHERE clause. The use of the greater-than operator (>) is crucial, as it ensures that the calculated start time of the period is excluded, defining a window that begins immediately after that specific moment 30 days prior.

The syntax remains concise and highly performant for this operation:

```
SELECT *
FROM sales
WHERE sales_date > NOW() - INTERVAL 30 DAY;
```

This query instructs MySQL to perform the time calculation first, determining the cutoff point dynamically using NOW() and the INTERVAL keyword. It then applies this calculated date against every record in the **sales_date** column, returning only the recent entries that satisfy the condition.

Output:

```
+----------+---------+---------------------+
| store_ID | item | sales_date |
+----------+---------+---------------------+
| 1 | Oranges | 2024-02-10 03:45:00 |
| 4 | Melons | 2024-01-14 03:29:55 |
+----------+---------+---------------------+
```

Analyzing the Filtered Results

Upon execution of the 30-day filtering query, the resulting output table confirms which records fall within the specified period (relative to the assumed execution date of 2/12/2024). The cutoff date (30 days prior) falls on January 13, 2024.

We can clearly see that only two records were returned: the sale of Oranges on February 10, 2024,

and the sale of Melons on January 14, 2024. Let's analyze why the other records were excluded:

Store ID 1 (Oranges, 2024-02-10): This date is only two days prior to the reference date (2/12/2024) and is therefore included.
Store ID 4 (Melons, 2024-01-14): This date is 29 days prior to the reference date and is included, as it is greater than the January 13th cutoff.
Store ID 2 (Apples, 2020-11-25): This date is several years old and far outside the 30-day window, hence excluded.
Store ID 5 (Grapes, 2023-05-19): Although within the last year, it still exceeds the 30-day limit and is excluded.

This analysis confirms that the syntax `sales_date > NOW() - INTERVAL 30 DAY` correctly performs the rolling date calculation, ensuring that only highly recent data is retrieved. This technique is indispensable for generating real-time dashboards or reports that require a snapshot of recent activity.

Alternative Approach: Using the DATE_SUB Function

While using `NOW() - INTERVAL X DAY` is the most idiomatic and often clearest way to perform date subtraction in MySQL, an alternative function exists that achieves the same result: `DATE_SUB()`. Understanding this alternative can be useful for compatibility or when integrating with legacy SQL codebases.

The `DATE_SUB()` function takes three arguments: the starting date/time expression, the INTERVAL magnitude, and the unit. Structurally, it explicitly calculates the subtracted date rather than relying on mathematical operators.

The equivalent query using `DATE_SUB()` would look like this:

```
SELECT *
FROM sales
WHERE sales_date > DATE_SUB(NOW(), INTERVAL 30 DAY);
```

Both methods yield identical results when applied correctly. For most modern SQL operations within MySQL, the simple arithmetic syntax (`NOW() - INTERVAL 30 DAY`) is often preferred due to its readability and concise nature, closely resembling standard mathematical subtraction, even though it operates on date objects.

Best Practices for Date and Time Handling

When implementing time-based filtering, several best practices should be considered to ensure

performance and accuracy. The choice of data type for the date column is paramount, as is understanding the performance implications of date functions.

First, always use appropriate data types. The DATETIME type, used in our example, is suitable when second-level precision is needed, such as for transaction logs. If time is irrelevant and only the calendar day matters, using the simpler DATE type can save storage space and slightly simplify queries (although the 30-day interval calculation works fine with both). Avoid storing dates as plain strings (TEXT or VARCHAR) as this prevents proper indexing and dramatically slows down range queries like the one demonstrated here.

Second, ensure that the date column being filtered (e.g., **sales_date**) is properly indexed. When MySQL executes the query `WHERE sales_date >`, it calculates the threshold once and then performs a comparison. Since the threshold is a fixed value during query execution, an index on **sales_date** can be effectively utilized for a fast index range scan, leading to highly optimized performance. Filtering without an index, especially on large tables, forces a full table scan, which is inefficient.

Finally, if you need to filter only by whole calendar days (ignoring the time component), you can wrap the functions in `DATE()`. For instance: `WHERE sales_date > DATE(NOW() - INTERVAL 30 DAY)`. This ensures the cutoff is exactly at midnight 30 days ago, making the result set cleaner for day-based reporting, though note that applying a function to the indexed column itself (`WHERE DATE(sales_date) > ...`) is generally discouraged as it prevents the use of the index.

Conclusion: Mastering Temporal Filtering

The ability to accurately and efficiently filter records based on a rolling time window is a cornerstone of effective database management and reporting. By utilizing the arithmetic capabilities provided by NOW() and the INTERVAL keyword, developers can create dynamic SQL queries that always provide the most up-to-date snapshot of recent data without requiring constant manual adjustment. This pattern is easily adaptable to other time frames--whether you need the last 7 days, 6 months, or 1 year--simply by adjusting the numerical value and the unit within the INTERVAL clause.

Mastering this technique ensures your applications and reports remain agile and responsive to evolving data needs. For those interested in exploring further date-based queries, such as selecting only rows corresponding to the current day, the following resources provide additional specialized guidance:

MySQL: How to Select Rows where Date is Equal to Today (External Tutorial Link)
Understanding the nuances of DATETIME versus DATE types in MySQL.
Advanced uses of the WHERE clause for composite indexing strategies.