How to Select Rows Older Than 7 Days in MySQL

Authored by

mohammed looti

January 5, 2026

When managing large datasets, particularly those involving time-sensitive information such as logs, transactions, or sales records, a common requirement is filtering data based on age. Specifically, identifying records that are older than a certain threshold--such as seven days--is crucial for tasks like archiving, reporting, or data cleanup. This article serves as an expert guide on constructing efficient and reliable SQL queries in MySQL to accurately retrieve rows where a specified date column predates the current moment by more than one week.

Achieving this dynamic comparison requires leveraging MySQL's robust set of built-in date and time functions. We must establish the current moment and then subtract the desired duration (seven days) to create a definitive cutoff point. Any record whose timestamp is less than (or older than) this calculated cutoff point will be returned. Understanding the nuances of date arithmetic is key to writing portable and performant database queries.

We will explore the primary method utilizing the `NOW()` function in conjunction with the `INTERVAL` clause, which is often considered the most straightforward and idiomatic approach in MySQL. Furthermore, we will detail an alternative solution involving the `DATE_SUB()` function and walk through a comprehensive practical example using a typical sales table structure to illustrate these concepts clearly.

The Challenge of Time-Based Filtering in MySQL

Filtering records based on a relative time frame, such as "older than seven days," presents a different challenge compared to filtering based on static dates. Static date filtering, for example, identifying records from '2024-01-01', is simple and unambiguous. However, dynamic filtering requires the database system to calculate the exact cutoff time at the moment the query is executed. This makes functions like `NOW()` indispensable tools for real-time data analysis and maintenance operations within a MySQL environment.

The core principle involves a simple comparison: the value in the date column must be strictly less than the calculated threshold date. If our target column, say `entry_date`, holds the timestamp for when a record was created, we are essentially looking for `entry_date < (Current Time - 7 Days)`. The precision of this calculation depends entirely on the datatype used for the date column. **DATETIME** or **TIMESTAMP** columns are typically employed for this purpose, providing the necessary granularity down to the second or fraction of a second.

This need for dynamic calculation highlights why relying on manual date input is impractical for recurring tasks. Utilizing MySQL's native time-handling syntax ensures that the query remains relevant regardless of when it is executed. Whether the query runs today, tomorrow, or a year from now, it will always target rows that are precisely seven days old or more, providing a robust solution for maintenance scripts, data integrity checks, and scheduled reports.

The Essential MySQL Time Functions: NOW() and INTERVAL

To perform date arithmetic effectively, we rely on two powerful components in MySQL: the `NOW()` function and the `INTERVAL` keyword. The `NOW()` function is fundamental; it retrieves the current date and time when the query is executed. It is crucial to note that `NOW()` returns a value representing the system time of the MySQL server, including both the date and the time components, which is ideal for precise comparisons.

The `INTERVAL` keyword is used to specify a time duration that needs to be added to or subtracted from a date or time expression. This mechanism simplifies complex time calculations into highly readable SQL syntax. For instance, subtracting seven days is expressed clearly as `INTERVAL 7 DAY`, or perhaps even more succinctly, `INTERVAL 1 WEEK`. This flexibility allows developers to easily adjust the time window--whether it's days, hours, weeks, months, or years--without needing to manually convert units.

Combining these elements creates the ideal syntax for establishing our cutoff time. By writing `NOW() - INTERVAL 1 WEEK`, the database first determines the current system time via `NOW()`, and then it subtracts the specified duration of one week. The result is a specific **DATETIME** value that represents the exact moment seven days prior to the query execution. This single, calculated value is then used as the benchmark against which all records in the target column are compared.

Implementing the Standard Solution: Using INTERVAL Subtraction

The most common and recommended method for selecting rows older than a specific duration in MySQL involves the direct subtraction of an `INTERVAL` from `NOW()`. This approach is highly efficient because it results in a single, static value for the comparison during the query execution, which facilitates better optimization by the MySQL query planner.

You can use the following basic syntax in MySQL to return all rows in a table where the date in a date column is older than 7 days ago:

```
SELECT *
FROM sales
WHERE sales_date < NOW() - INTERVAL 1 WEEK;
```

This particular example selects all rows in the table named **sales** where the date in the **sales_date** column is older than 7 days ago.

Note: The **NOW()** function in MySQL returns the current date and time, providing the baseline for the calculation.

In the query above, the condition `sales_date < NOW() - INTERVAL 1 WEEK` ensures that only records whose `sales_date` occurred before the calculated seven-day cutoff point are included in the results. The use of the less-than operator (`<`) is crucial, as it strictly excludes records that occurred exactly seven days ago up to the current moment, focusing only on truly older data.

For scenarios where you might need to filter data based on full calendar days rather than precise 24-hour intervals, you could wrap `NOW()` or the date column in the `DATE()` function to strip away the time component. However, for precise age filtering, especially in applications dealing with transactions or highly time-sensitive events, using the full **DATETIME** value returned by `NOW()` is generally the preferred best practice.

Alternative Approach: Utilizing the DATE_SUB() Function

While the `NOW() - INTERVAL` syntax is highly readable and direct, MySQL offers an equivalent function-based alternative: `DATE_SUB()`. This function explicitly subtracts a time value from a date. The syntax requires two main arguments: the starting date/time expression and the interval to subtract.

The equivalent query using the `DATE_SUB()` function looks like this: `WHERE sales_date < DATE_SUB(NOW(), INTERVAL 7 DAY)`. This syntax achieves the identical result as the subtraction operator method. The choice between the two often comes down to personal preference or adherence to specific coding standards within a development team, as both are generally optimized similarly by the MySQL engine.

Using `DATE_SUB()` can sometimes be perceived as clearer when dealing with more complex date manipulations, especially when combining multiple operations. However, for a simple subtraction of a time unit, the arithmetic operator approach (`NOW() - INTERVAL 1 WEEK`) tends to be the most concise and elegant solution. Both methods demonstrate the flexibility of MySQL in handling dynamic time comparisons.

A Practical Demonstration: Setting Up the Sales Data

To fully illustrate how this filtering works, let us establish a working example using a hypothetical `sales` table. This table tracks individual transactions, including the store identifier, the item sold, and, most importantly, the exact date and time of the sale, stored in a **DATETIME** column named `sales_date`.

The following example shows how to use this syntax in practice. Suppose we have the following table named **sales** that contains information about sales made at various grocery stores at various times:

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

-- 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;
```

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


The provided dataset includes sales spanning several years, with one record intentionally placed very recently (2024-02-10). The goal of our upcoming query is to accurately isolate and retrieve only those sales records that occurred a full seven days or more before the query execution time. This setup allows us to clearly verify the effectiveness and precision of our date subtraction logic in SQL.

This article is currently being written on **2/12/2024**.

Executing the Query and Interpreting Results

Assuming the execution date is **2/12/2024** (February 12th, 2024), the calculated cutoff date will be exactly seven days prior, accounting for the time component returned by `NOW()`. Any record with a `sales_date` that falls on or after 2/05/2024 will be excluded from the results. We expect the record for 'Oranges' (2024-02-10) to be filtered out, as it is only two days old relative to the execution date.

Suppose that we would like to select all rows where the date in the **sales_date** column is older than 7 days ago. We can use the following syntax to do so:

```
SELECT *
FROM sales
WHERE sales_date < NOW() - INTERVAL 1 WEEK;
```

Output:

```
+----------+---------+---------------------+
| store_ID | item | sales_date |
+----------+---------+---------------------+
| 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 |
+----------+---------+---------------------+
```

The output clearly demonstrates the successful application of the time-based filter. Notice that each of the rows in the resulting table have a date in the **sales_date** column that is older than 7 days since today's date of **2/12/2024**. The record for Store ID 1 (Oranges, 2024-02-10) is correctly excluded, as it is too recent. This confirms that the combination of `NOW()` and `INTERVAL` provides a precise filter against the current moment in time.

Performance Considerations and Indexing Date Columns

When dealing with tables containing millions or billions of rows, the performance of date-based filtering becomes a critical factor. Even a simple `SELECT` query can become resource-intensive if the database system has to perform a full table scan every time to evaluate the `WHERE` clause. To ensure efficient execution of dynamic time-based queries, proper **indexing** is paramount.

The primary rule for optimization in this context is to place an **index** on the column being filtered--in our case, `sales_date`. Because the comparison uses a calculable, static cutoff time (`NOW() -`

`INTERVAL 1 WEEK`), MySQL can utilize the **index** to quickly jump to the relevant range of data without scanning the entire table. This is highly effective because the date function is applied only once to determine the boundary, and not repeatedly to every row.

It is important to avoid applying functions directly to the indexed column within the `WHERE` clause (e.g., `WHERE DATE(sales_date) < ...`). Applying a function to the column itself forces MySQL to recalculate the value for every row before comparison, thereby negating the benefits of the **index**. Since our solution structure--`WHERE sales_date <` --compares the raw indexed column against a pre-calculated boundary, it is inherently optimized for speed. Always use `EXPLAIN` to analyze the query plan and confirm that the intended **index** is being utilized effectively.

Summary of Best Practices for Date Filtering

Successfully managing historical data in MySQL relies on selecting the right combination of functions and ensuring the underlying table structure supports fast lookups. The method of subtracting an `INTERVAL` from `NOW()` provides the most robust and readable solution for dynamic aging checks.

Key takeaways for working with time-based filtering:

Utilize Dynamic Functions: Always use functions like `NOW()`, `CURDATE()`, or `UTC_TIMESTAMP()` to calculate the current time dynamically, ensuring the query remains accurate regardless of execution time.

Choose the Right Interval Unit: Use the `INTERVAL` keyword with appropriate units (DAY, WEEK, MONTH) for clear and precise date arithmetic.

Ensure Indexing: For large tables, the `sales_date` column must be indexed to maintain high query performance, especially for archival or cleanup operations that run frequently.

Avoid Column Function Wrappers: To leverage indexes, ensure that functions are applied only to the benchmark time (e.g., `NOW()`), and not to the indexed column itself.

By following these guidelines, developers and administrators can write highly optimized SQL queries that efficiently handle the complex requirement of filtering data based on how old it is relative to the present moment. This is a crucial skill for maintaining healthy and responsive database systems.

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

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