How to Retrieve Rows Between Two Dates in MySQL

Authored by

mohammed looti

January 5, 2026

Filtering data based on time ranges is one of the most common requirements in database management and analysis. When working with MySQL, efficiently retrieving records that fall within a specified date period is essential for generating reports, tracking historical trends, and summarizing recent activity. Fortunately, MySQL provides several powerful mechanisms to handle temporal queries, chief among them being the BETWEEN operator. Understanding how to correctly implement this operator, especially concerning data formatting, is key to writing accurate and performant SQL queries.

The BETWEEN function is specifically designed to handle range checks, allowing developers to define a lower and an upper bound for a column's value. When applied to date columns, it simplifies the process of selecting all rows where the date value is greater than or equal to the start date and less than or equal to the end date. This functionality eliminates the need for more complex compound conditional statements using the >= and <= operators combined with SQL's **AND** keyword. However, to ensure seamless operation, the dates provided must adhere to a strict format recognized by MySQL, such as the standard **YYYY-MM-DD** structure.

This guide will provide a detailed, step-by-step explanation of how to leverage the BETWEEN operator in MySQL to extract data across specific date intervals. We will cover the fundamental syntax, walk through a practical example using a sample dataset, and explore advanced techniques like utilizing dynamic date functions. By the end of this tutorial, you will possess the expertise required to accurately filter records based on two defining dates, ensuring your database interactions are both precise and efficient.

Prerequisites: Understanding MySQL Date Formats

Before diving into the query structure, it is critically important to understand how MySQL handles date and time data types. The database system needs dates to be provided in an unambiguous format to perform accurate comparisons. While users might instinctively use regional formats like MM/DD/YYYY or DD-MM-YY, MySQL defaults to the standard ISO 8601 format for textual date input, which is **YYYY-MM-DD**. Failure to use this format can lead to incorrect results, unexpected behavior, or even syntax errors, especially when the day or month value exceeds 12.

The date column itself should be defined using an appropriate data type, such as **DATE**, DATETIME, or **TIMESTAMP**. The choice between these types impacts the granularity of the filtering. If your column is defined as **DATE**, the BETWEEN comparison is straightforward, checking only the calendar date. However, if you are using the more precise DATETIME or **TIMESTAMP** types--which include time components (hours, minutes, seconds)--you must be mindful of how the time component interacts with the range boundaries, particularly the end date.

For example, if you query between '2024-01-01' and '2024-01-31' on a DATETIME column, the query will only include records up to **2024-01-31 00:00:00**. Any events that occurred later on

January 31st will be excluded. To truly include the entire day of the end date, you must specify the last possible moment of that day, such as '2024-01-31 23:59:59', or utilize date manipulation functions like `DATE_ADD()` or `DATE()` casts to normalize the comparison, although the BETWEEN operator inherently includes both boundaries.

The Basic Syntax for Date Range Queries

The structure of a query utilizing the BETWEEN operator for date filtering is standard SQL syntax, focusing on the **WHERE** clause. The core functionality selects the desired columns (or all columns using `SELECT *`) from a specific table and then applies the conditional check to the relevant date column. This structure is highly efficient for retrieving large datasets within a targeted time frame.

The general template involves specifying the column name, followed by the **BETWEEN** keyword, and then defining the start date and end date, separated by the **AND** keyword. Note that the dates must be enclosed in single quotes, treating them as string literals which MySQL intelligently converts into temporal data for comparison. The simplicity of this syntax is its primary advantage over manually chaining together inequality comparisons.

Below illustrates the fundamental syntax used in MySQL to return all rows in a table where a date column falls between two specific date strings. We assume a table named `sales` and a column named `sales_date` for this generic example:

You can use the following basic syntax in MySQL to return all rows in a table where a date column is between two specific dates:

SELECT *
FROM sales
WHERE (sales_date BETWEEN '2020-01-01' AND '2024-01-20');

This particular example selects all rows in the table named **sales** where the date stored in the **sales_date** column is inclusively between **January 1st, 2020** and **January 20th, 2024**. It is critical to remember that the range defined by BETWEEN is inclusive, meaning records exactly matching the start date or the end date will be returned.

The following section provides a comprehensive demonstration of how to apply this syntax in a practical database scenario, starting with the necessary table creation and data insertion steps.

Comprehensive Example: Setting up the Sales Table

To effectively illustrate the use of the BETWEEN operator, we will construct a simple but realistic dataset. Imagine a retail environment where we track various sales transactions. We need a table

that stores a unique identifier, the item sold, and the exact DATETIME of the transaction. This choice of DATETIME will highlight the importance of time component considerations later in the filtering process.

Suppose we have the following table named **sales** that contains detailed information about produce sales made at various grocery stores over several years. We define the table structure using the SQL **CREATE TABLE** command, ensuring the `sales_date` column is defined as DATETIME to capture the time of day alongside the date. We then populate this table with five diverse rows, spanning dates from 2009 to 2023.

```
-- 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', '2015-01-12 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', '2022-04-09 03:29:55');
INSERT INTO sales VALUES (0005, 'Grapes', '2023-05-19 23:10:04');

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

When we run the final `SELECT * FROM sales;` query, the output confirms the successful creation and population of our sample data. This table serves as the foundation for our subsequent date filtering demonstrations. The varying years and precise time stamps across the records provide a robust testing ground for the BETWEEN operation, enabling us to clearly see which records are included or excluded based on the defined date range boundaries.

Output of the sales table:

```
+----------+---------+---------------------+
| store_ID | item | sales_date |
+----------+---------+---------------------+
| 1 | Oranges | 2015-01-12 03:45:00 |
| 2 | Apples | 2020-11-25 15:25:01 |
| 3 | Bananas | 2009-06-30 09:01:39 |
```

| 4 | Melons | 2022-04-09 03:29:55 |
| 5 | Grapes | 2023-05-19 23:10:04 |
+----------+---------+---------------------+

Executing the Date Range Query

Our objective is now to isolate only the recent sales transactions that occurred within a defined four-year window. Specifically, let us suppose that we are interested in selecting all rows where the date in the **sales_date** column falls between the start of 2020 and early 2024. This requires setting the lower boundary to '2020-01-01' and the upper boundary to '2024-01-20'. By specifying the dates in the **YYYY-MM-DD** format, we ensure that MySQL interprets the range correctly.

We apply the standard SQL **SELECT** statement combined with the powerful BETWEEN operator within the **WHERE** clause. The query specifies the exact column (`sales_date`) that is being evaluated against the temporal boundaries. This approach is significantly cleaner and more readable than using the equivalent comparison operators (`WHERE sales_date >= '2020-01-01' AND sales_date <= '2024-01-20'`), making the query intention immediately clear to other developers.

We can use the following syntax to perform this targeted filtering operation on our `sales` table. This query tells MySQL to scan the entire table and retrieve only those records whose DATETIME value is chronologically bounded by the two specified endpoints.

```
SELECT *
FROM sales
WHERE (sales_date BETWEEN '2020-01-01' AND '2024-01-20');
```

Analyzing the Results and Understanding Inclusivity

After executing the targeted query, MySQL processes the rows against the defined range. By comparing the output against the original table data, we can confirm the accuracy of the BETWEEN function and verify which records met the criteria. The records for Oranges (2015) and Bananas (2009) are correctly excluded because their `sales_date` falls outside the '2020-01-01' to '2024-01-20' window.

The resulting table contains only three rows: Apples (2020), Melons (2022), and Grapes (2023). All three of these transactions happened after the starting boundary (2020-01-01 00:00:00) and before or exactly at the ending boundary (2024-01-20 00:00:00). If we had a record dated exactly '2024-01-20 00:00:00', it would also be included, reinforcing the principle that BETWEEN is an inclusive operator for both its starting and ending values.

Output of the filtered query:

```
+----------+--------+---------------------+
| store_ID | item | sales_date |
+----------+--------+---------------------+
| 2 | Apples | 2020-11-25 15:25:01 |
| 4 | Melons | 2022-04-09 03:29:55 |
| 5 | Grapes | 2023-05-19 23:10:04 |
+----------+--------+---------------------+
```

Notice that each of the rows in the resulting table has a date in the **sales_date** column that is chronologically between **January 1st, 2020** and **January 20th, 2024**. This powerful filtering technique is fundamental to time-series data analysis and reporting, enabling users to quickly narrow down vast datasets to relevant time frames without extensive manual processing.

Advanced Technique: Querying Up to the Current Date using CURDATE()

Often in dynamic reporting, the required end date for the filter is not a fixed, historical value but the current date on which the query is executed. For instance, a user might want a report showing all activity from the start of the year up to the moment the report is run. MySQL facilitates this through built-in temporal functions, most notably the CURDATE() function.

The CURDATE() function returns the current date as a value in 'YYYY-MM-DD' format. When used as the upper boundary in a BETWEEN clause, it dynamically sets the end point of the range based on the server's clock at the time of execution. This is incredibly useful for automated scripts and continuously updated dashboards where manually changing the end date parameter is impractical or prone to human error.

If you would like to return all rows where the date is between a specific starting date and the current date, you can substitute the hardcoded end date string with the CURDATE() function. In the example below, we retrieve all sales records that occurred since the beginning of 2020 up until today.

```
SELECT *
FROM sales
WHERE (sales_date BETWEEN '2020-01-01' AND CURDATE());
```

Since the sample data contains sales records up to 2023-05-19, and assuming the execution date is sometime in early 2024, the output remains the same as the previous example, as all three recent sales fall within the bounds of '2020-01-01' and the current date (which is later than the

2023 sales).

```
+----------+--------+---------------------+
| store_ID | item | sales_date |
+----------+--------+---------------------+
| 2 | Apples | 2020-11-25 15:25:01 |
| 4 | Melons | 2022-04-09 03:29:55 |
| 5 | Grapes | 2023-05-19 23:10:04 |
+----------+--------+---------------------+
```

Note: This article assumes a writing date of **February 12th, 2024**. Therefore, this specific query returns all rows where the `sales_date` is between **January 1st, 2020** and **February 12th, 2024**. If the date column used was **DATE** instead of DATETIME, the results would be entirely predictable; however, because we are using DATETIME, MySQL treats `CURDATE()` (e.g., '2024-02-12') as '2024-02-12 00:00:00', meaning any sale occurring later on that day would only be included if it happened exactly at midnight. To include the full day, consider using `NOW()` or `DATE_ADD(CURDATE(), INTERVAL 1 DAY)` in combination with `<`, as discussed in the limitations section.

Limitations and Alternatives to the BETWEEN Operator

While the BETWEEN operator offers clean syntax, it is crucial to understand its potential limitations, particularly when dealing with columns of type DATETIME or **TIMESTAMP**. As previously noted, MySQL interprets a date string like '2024-01-20' as the very start of that day, specifically '2024-01-20 00:00:00'. If you intend to include events that occurred throughout the entire day of January 20th, using BETWEEN with the simple date string as the upper bound will result in silently dropped data points.

To guarantee the inclusion of all events up to the end of the final day, developers often turn to the explicit use of inequality operators (>= and <). This method offers finer control over the boundaries. By ensuring the start date uses `>=` and the end date uses `<` (less than) the beginning of the *next* day, you can perfectly capture the intended 24-hour period for the end date.

For example, to include all sales between January 1, 2020, and the absolute end of January 20, 2024, the alternative SQL query would look like this. Notice how we use `DATE_ADD()` to calculate the day immediately following the intended end date:

Query using explicit operators for full day inclusion:

```
SELECT *
FROM sales
```

```
WHERE sales_date >= '2020-01-01'
AND sales_date < DATE_ADD('2024-01-20', INTERVAL 1 DAY);
```

This alternative syntax, while slightly more verbose than BETWEEN, removes all ambiguity when dealing with time components in DATETIME fields. For queries involving simple **DATE** fields where time is irrelevant, BETWEEN remains the most elegant solution. Therefore, the selection of the correct filtering method depends heavily on the data type of the column being queried and whether midnight cutoff points are acceptable for the reporting needs.

Conclusion: Mastering Date Filtering in SQL

Filtering rows based on a date range is a foundational skill in database querying, and MySQL provides robust tools to handle this requirement efficiently. The BETWEEN operator stands out for its clarity and ease of use, allowing developers to define inclusive temporal boundaries using minimal syntax. We have demonstrated how crucial it is to use the standard **YYYY-MM-DD** date format to ensure that MySQL accurately compares the values against the column data.

Furthermore, we explored how dynamic functions like CURDATE() can be seamlessly integrated into the BETWEEN clause to create powerful, automated queries that adapt to the current time, fulfilling common reporting needs for current activity. Mastering these techniques ensures that data extraction is precise and optimized for performance.

For complex scenarios involving DATETIME columns, remember the potential pitfall of the upper boundary being treated as midnight (00:00:00). In such cases, switching to explicit inequality operators (>= and <) targeting the start of the next day provides the necessary precision to capture all records within the final 24-hour period. By choosing the right tool--be it the succinct BETWEEN operator or the more explicit inequality checks--you can ensure your SQL queries consistently deliver the accurate results required for effective data analysis.

Related MySQL Tutorials

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

How to Filter by Month and Year in MySQL

Using DATE_FORMAT() for Custom Date Outputs

Difference Between DATETIME and TIMESTAMP in MySQL