Learning to Query Data Between Two Dates in MySQL

Authored by

Mohammed looti

November 12, 2025

Filtering data based on chronological criteria is a fundamental necessity for effective data management within any modern relational database. Whether the task involves generating precise monthly revenue reports, conducting deep historical trend analysis, or efficiently managing large volumes of archival information, the core requirement remains the same: the ability to accurately retrieve rows that fall within a defined temporal interval. The SELECT statement in **MySQL** offers robust and specialized mechanisms for achieving this precision, primarily by leveraging comparison operators specifically designed to handle complex date and time data types.

The most efficient and highly recommended approach for isolating data between two specified dates involves utilizing the **`BETWEEN`** operator, placed strategically within the **`WHERE`** clause. This powerful structural combination allows developers and analysts to clearly define both the starting and ending points of the desired temporal window, guaranteeing that the resulting data set is highly precise and directly relevant to the specific analytical goals being pursued.

The following basic syntax demonstrates how to instruct **MySQL** to return all rows in a designated table where a specific date column falls between two static, predefined dates. This method is the cornerstone for fixed-period reporting.

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

This example query performs a filtered retrieval on the **`sales`** table, selecting every column (indicated by the wildcard ***) only if the value stored in the **`sales_date`** column is chronologically greater than or equal to **January 1, 2020**, and simultaneously less than or equal to **January 20, 2024**. It is vital to remember that the BETWEEN operator is inherently **inclusive**, meaning that records timestamped exactly on either the starting boundary or the ending boundary will be fully included in the final result set, provided a matching record exists.

The Core Syntax: Utilizing the BETWEEN Operator

The **`BETWEEN`** operator serves as an essential logical operator in SQL, dramatically simplifying the process of verifying whether an expression falls within a predefined range of values. When this operator is applied to columns containing date or time information, the database engine interprets it as efficient shorthand for two distinct comparison operations: specifically, that the value must be greater than or equal to the lower bound, and simultaneously less than or equal to the upper bound. This consolidated structure provides significantly cleaner and more readable code compared to manually combining explicit **`>=`**, **`<=`**, and **`AND`** operators.

When defining date ranges within **MySQL** queries, maintaining strict consistency in the date string

format is paramount. The recommended standard format is **YYYY-MM-DD**. Although **MySQL** exhibits a degree of flexibility in parsing date formats, adhering to this canonical structure proactively prevents potential misinterpretations or ambiguities, which are especially common when integrating international date formats or when the target column uses the detailed DATETIME data type that incorporates a time component. Using the standard format ensures that the database engine correctly establishes the chronological order for filtering.

Understanding the **inclusive** nature of the BETWEEN operator is critical for accurate reporting. For example, if a range is defined from '2020-01-01' to '2024-01-20', any record timestamped precisely at the stroke of midnight (00:00:00) on either of these two dates will be automatically included. If the business requirement necessitates the strict exclusion of the endpoints--that is, selecting data *after* the start date and *before* the end date--then developers must bypass **BETWEEN** and instead employ the standard comparison operators (specifically **>** for the start and **<** for the end).

Setting Up the Demonstration Environment

To effectively illustrate the practical application of date range filtering, we will begin by provisioning a sample table named **sales**. This setup is designed to simulate a real-world scenario by tracking specific sales transactions, including a unique identifier, the item purchased, and the precise date and time of the sale. Utilizing the **DATETIME** data type for the **sales_date** column is deliberate, as it allows us to demonstrate how the **BETWEEN** operator accurately processes queries that involve both date and highly specific time components simultaneously.

The following structured SQL commands are used to establish the schema for the **sales** table and subsequently populate it with five distinct records. These records are intentionally spread across several years, providing a diverse baseline dataset against which we can rigorously test our date filtering logic. Note the necessary precision in the **sales_date** entries, as these detailed timestamps are fundamental for accurate time-based analysis and demonstration.

```
-- 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;
```

The resulting table structure, which now contains our foundational data set, is presented below. Observing the data, we can confirm that the sales range chronologically from 2009 up to 2023. This variation is essential as it clearly allows us to predict and verify which specific rows will be successfully filtered and returned when we execute the targeted date range query in the next section.

Output of Sample Data:

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

Practical Application: Filtering Data Between Fixed Dates

For the purpose of our initial practical exercise, let us assume a business requirement to generate a comprehensive report covering all sales activity from the start of the year 2020 through the early weeks of 2024. Our specific objective is to retrieve all rows where the timestamp in the **`sales_date`** column falls chronologically between **January 1, 2020**, and **January 20, 2024**. This carefully selected four-year span will demonstrate the operator's ability to capture transactions across multiple years seamlessly.

To successfully execute this specific temporal filtering requirement, we must construct a clear and concise SELECT statement that integrates the BETWEEN operator. The resulting query structure effectively defines the necessary temporal boundaries using standard date strings, enabling the **MySQL** engine to process the filter criteria against the indexed date column with maximal efficiency and accuracy.

SELECT *

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

Upon executing the query, the database engine sequentially evaluates every record against the established chronological limits. Immediately, Row 1 (dated 2015) and Row 3 (dated 2009) are correctly excluded because their timestamps precede the starting boundary of 2020-01-01. The remaining three rows, which contain dates from 2020, 2022, and 2023, respectively, are confirmed to fall within the designated range and are thus included in the final, filtered result set.

Output of 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 |
+----------+--------+---------------------+

The output definitively confirms the expected result: every row successfully returned possesses a date in the **`sales_date`** column that resides chronologically between the precise boundaries of **January 1, 2020**, and **January 20, 2024**. This outcome underscores the high efficacy and inherent precision of using the **`BETWEEN`** operator for handling fixed date range filtering tasks in **MySQL**.

Handling Dynamic Date Ranges with CURDATE()

While specifying static, hardcoded dates is perfect for analyzing fixed historical periods, many modern applications demand reports that update dynamically to reflect activity right up to the present moment. If the requirement is to return all records where the date falls between a fixed starting point and the current date (today), **MySQL** provides robust, built-in functions to manage this dynamic endpoint efficiently. The most appropriate function for this specific scenario is the highly useful CURDATE() function.

The CURDATE() function is designed to return the server's current date, formatted precisely as a **`'YYYY-MM-DD'`** string value. By substituting the static end date boundary in our query with **`CURDATE()`**, the entire database query instantly transforms into a dynamic instrument. Every time the query is executed, the upper boundary automatically adjusts to the server's current date, eliminating any need for manual query string updates and guaranteeing that the generated report always captures the most recent data available.

We can adapt our previous example to dynamically select all sales that have occurred since the beginning of 2020 up to and including the current day of execution. This critical flexibility is achieved by setting the lower bound to the static **`'2020-01-01'`** and setting the upper bound dynamically using the **`CURDATE()`** function, as demonstrated below:

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

Executing this dynamic query will typically yield the same result set as the previous fixed-date example, assuming the current date falls within or after the specified historical period. However, the paramount difference lies in the inherent flexibility offered by **`CURDATE()`**, making this specific method the industry standard for populating live dashboards, automatically generated reports, and time-sensitive operational interfaces.

```
+----------+--------+---------------------+
| 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: Assuming the date of execution is **February 12, 2024**, this particular query effectively returns all rows that possess a **`sales_date`** between **January 1, 2020**, and **February 12, 2024**. Crucially, if the exact same query were to be executed the following day, the upper bound would automatically and instantly shift to include data up to February 13, 2024, without any modification to the SQL code itself.

Understanding Date/Time Data Types and Precision

When constructing complex date range filters in **MySQL**, the specific data type chosen for the chronological column--whether it is **`DATE`**, **`DATETIME`**, or **`TIMESTAMP`**--exerts a significant influence on the final query results, especially when paired with the BETWEEN operator. In our demonstration, we used the DATETIME data type, which meticulously tracks time down to the second.

A common pitfall arises when developers specify date boundaries without explicitly including a time component, such as setting the end date simply as **`'2024-01-20'`**. **MySQL** automatically interprets this string as the very start of that day: **`'2024-01-20 00:00:00'`**. Since the **`BETWEEN`** operator is inclusive, any record that occurred precisely at the stroke of midnight is

included. However, any transaction occurring later that same day (e.g., **`'2024-01-20 15:30:00'`**) would be inadvertently excluded from the query results because the upper boundary was defined too strictly at midnight.

To ensure the comprehensive capture of an entire day's worth of transactions when utilizing a DATETIME data type, the upper boundary must be deliberately adjusted to the absolute end of the final day. For instance, to accurately include all sales that transpired on January 20, 2024, the effective end date should be set to **`'2024-01-20 23:59:59'`**. A more robust and cleaner technique involves using time functions like **`DATE_ADD`** to add one full day to the boundary date, and then using the strict less-than operator (**`<`**) to ensure records from that added day are excluded while maximizing coverage of the desired final day.

Alternatively, if the analytical requirement is explicitly limited to comparing only the date portion, regardless of the corresponding time stamp, developers can force this behavior by explicitly converting the date column using the **`DATE()`** function directly within the **`WHERE`** clause. This technique compels the BETWEEN operator to evaluate only the date component of the **`sales_date`** column, effectively simplifying the boundary definitions back to simple date strings without any concern for handling the underlying time stamps.

Conclusion and Further Reading

The effective querying of temporal data stands as a core, foundational skill for any developer or analyst working with **MySQL**. By masterfully utilizing the **`BETWEEN`** operator, along with careful consideration of data types, programmers can construct concise, highly efficient, and easily readable SELECT statements capable of filtering records across both fixed historical periods and dynamic, ever-changing date ranges. It is crucial to always remember that precision is key, particularly when defining date-only boundaries against columns storing the detailed DATETIME data type. For comprehensive dynamic reporting needs, incorporating functions such as the CURDATE() function guarantees that reports consistently reflect the most current information available to the system. Mastering these specific techniques unlocks powerful capabilities for both sophisticated historical analysis and critical real-time operational reporting.

Additional Resources

The following curated tutorials explain how to perform other common and highly advanced tasks in **MySQL**, assisting you in further optimizing and refining your database query skills for complex scenarios: