How to Easily Filter Data with the LIKE Operator in SQL

Authored by

stats writer

January 1, 2026

The LIKE operator is a fundamental and powerful tool used across various database management systems and data processing frameworks, including PySpark, to perform sophisticated string matching and filtering operations. Unlike simple equality comparisons, which require an exact match, LIKE allows users to filter rows from a table or a PySpark DataFrame based on whether a column's value conforms to a specified text **pattern**. This is achieved using wildcard characters, such as an underscore (_) to match any single character or a percent (**%**) to match a sequence of zero or more characters. The LIKE operator can be easily combined with other conditions to construct highly selective filtering expressions that return only the desired data subset.

1. Introduction to the LIKE Operator and Pattern Matching

The core utility of the LIKE operator lies in its flexibility. It enables data engineers and analysts to retrieve records where a text field contains, starts with, or ends with a certain sequence of characters, or even matches a specific structure regardless of the content in certain positions. Implementing pattern matching efficiently is crucial for data cleaning, reporting, and specialized searches across massive datasets where partial information or ambiguous search criteria are common.

In the context of distributed computing frameworks like PySpark, the LIKE operation is optimized to handle massive scales, ensuring that pattern matching remains fast and reliable even when working with terabytes of information stored across clusters. Understanding the nuances of how LIKE interacts with PySpark's distributed architecture is vital for writing performant data processing code. We will explore how to leverage this operator effectively within the PySpark environment, providing clear syntax examples and a comprehensive, practical demonstration.

2. Understanding Wildcard Characters in LIKE

The power of the LIKE operator is amplified by its reliance on wildcard characters. Mastering the use of these wildcards is the key to creating precise and effective filters. In standard **SQL** environments, and consequently in PySpark, two primary wildcard characters are utilized: the percent sign (**%**) and the underscore (_). Each serves a distinct purpose in defining the matching criteria.

The **percent sign (%)** represents zero, one, or multiple characters of any type. For instance, if you are searching for all team names that start with the letter 'M', the pattern would be 'M%', matching 'Mavs', 'Miami', or 'Mavericks'. Conversely, if you want all names ending in 's', the pattern '%s' would be used. To find a specific sequence, such as 'avs', anywhere within the string, the pattern `'%avs%'` is employed. This flexibility makes the percent sign essential for broad, containment-based searches.

The **underscore (_)**, in contrast, serves as a placeholder for any single character. This wildcard is crucial when the structure of the string is known, but the content of a specific position is variable. For example, if you are looking for codes that are exactly three characters long and start with 'A', you would use the pattern 'A__'. This pattern would match 'A12', 'AXY', or 'A00', but crucially, it would not match 'A123'. By using the underscore, we impose a strict length constraint on the pattern match.

3. Applying the LIKE Operator in PySpark Syntax

The implementation of the LIKE operator within PySpark is achieved using the dedicated column function, `.like()`. This method is invoked directly on the DataFrame column intended for filtering. The resulting boolean expression is then passed to the `.filter()` transformation, which evaluates the condition across the distributed dataset and returns a new DataFrame containing only the matched rows.

You can use the following standard Python syntax to filter a PySpark DataFrame using the LIKE operator:

df.filter(df.team.like('%avs%')).show()

This particular example filters the DataFrame to only show rows where the string in the **team** column contains the pattern 'avs' somewhere within the string, irrespective of characters preceding or following it. This functionality is invaluable for partial name searching or identifying data entries that share a common substring identifier.

4. Step-by-Step Example: Defining the DataFrame

To provide a practical demonstration, we will first create a sample PySpark DataFrame named `df`. This DataFrame contains mock information about points scored by various basketball teams. This requires setting up the Spark environment, defining the dataset, and creating the DataFrame structure.

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = ,
,

,
,
,
,
,
,
,
]

#define column names
columns =

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+-------+------+
| team|points|
+-------+------+
Mavs	18
Nets	33
Lakers	12
Mavs	15
Cavs	19
Wizards	24
Cavs	28
Nets	40
Mavs	24
Spurs	13
+-------+------+

The resulting output clearly shows the initial state of the dataset. Our objective is now to isolate only those records where the team name contains the specific sequence of characters "avs", thus filtering out teams like 'Nets', 'Lakers', 'Wizards', and 'Spurs'. This requires the precise application of the `.like()` function.

5. Executing the LIKE Filter in Practice

To successfully filter the DataFrame based on the presence of the substring "avs" within the `team` column, we construct the pattern `'%avs%'`. This pattern, utilizing the percent sign at both ends, ensures that the filter is inclusive of any string that merely contains the sequence 'avs', regardless of its position. This includes both 'Mavs' and 'Cavs'.

We can use the following syntax to filter the DataFrame to only contain rows where the **team** column contains the pattern 'avs' somewhere in the string:

#filter DataFrame where team column contains pattern like 'avs'
df.filter(df.team.like('%avs%')).show()

```
+----+------+
|team|points|
+----+------+
|Mavs| 18|
|Mavs| 15|
|Cavs| 19|
|Cavs| 28|
|Mavs| 24|
+----+------+
```

This execution yields a new, smaller DataFrame containing only the records corresponding to teams that satisfy the pattern match. The use of `df.team.like()` is the functional equivalent of the SQL `WHERE team LIKE '%avs%'` clause, seamlessly integrated into the Python environment provided by PySpark. This method demonstrates the efficiency and clarity of using native DataFrame operations for complex string filtering tasks.

6. Analyzing the Filtered Results

Notice that each of the rows in the resulting DataFrame contain 'avs' in the **team** column. After executing the filter operation, careful analysis of the resulting DataFrame confirms the success of the pattern matching. The output shows five rows, three for 'Mavs' and two for 'Cavs'. Every row in this result set precisely matches the criteria defined by the `'%avs%'` pattern. This validation step is critical in any data manipulation workflow, confirming that the filtering logic performed as intended.

If the requirement had been stricter, for example, matching only teams that start with 'C' and have 'avs' within them, the pattern would be adjusted to `'C%avs%'`, demonstrating the precise control afforded by combining literal characters with the percent wildcard. Furthermore, using the single-character wildcard, such as `'_avs'`, would result in an empty DataFrame here, as no team name is exactly four characters long ending in 'avs'. This highlights the importance of choosing the

correct wildcard characters for the desired level of string matching.

7. Summary and Further Resources

The LIKE operator, implemented in PySpark via the `.like()` function, is a fundamental mechanism for performing flexible and robust pattern-based filtering on DataFrames. By effectively utilizing wildcard characters, data practitioners can precisely target and extract necessary rows from vast datasets. This capability is critical for data quality checks, custom reporting, and complex textual analysis across large-scale data environments.

Note: You can find the complete documentation for the PySpark **like** function, offering detailed insights into its parameters and behavior, in the official Apache Spark documentation.

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