Learning PySpark: How to Filter DataFrame Rows with the LIKE Operator

Authored by

Mohammed looti

November 11, 2025

The ability to filter large datasets based on specific text patterns is a fundamental requirement in data analysis. In the context of big data processing using PySpark, this capability is efficiently provided by the standard SQL **LIKE operator**. This guide explains the precise syntax and practical application required to filter rows within a DataFrame using this powerful pattern-matching mechanism. Understanding how to deploy the `like()` function is crucial for analysts and data engineers working with textual or categorical data where exact matches are often insufficient.

The core syntax involves calling the `filter()` method on your existing DataFrame, specifying the target column, and chaining the `like()` function with the desired pattern string. This structure allows for highly readable and performant filtering operations, leveraging the optimized execution engine of Apache Spark. We will demonstrate how standard SQL wildcards, particularly the percentage symbol (%), enable flexible searches that return rows matching a pattern anywhere within the specified column's string value.

Understanding Pattern Matching in PySpark DataFrames

When dealing with real-world data, especially textual fields, data scientists frequently encounter situations where they need to retrieve records based on partial string matches rather than exact comparisons. For instance, you might want to identify all customers whose city name starts with "San" or find all product codes containing the sequence "XYZ". This is where the **LIKE operator** becomes indispensable. In PySpark, the `.like()` function is the native method for performing these SQL-style pattern-matching operations directly on a column within a DataFrame.

The effectiveness of the `like()` method hinges entirely on the use of **wildcards**. These special characters allow the definition of flexible search patterns. The most common wildcard is `%` (percentage), which matches any sequence of zero or more characters. By strategically placing the `%` symbol around a substring, you dictate where the pattern must appear. For example, a pattern like `'A%'` finds strings that start with 'A', while `'%Z'` finds strings that end with 'Z'. Using `'%pattern%'`, as often required, matches the sequence regardless of its position within the string.

It is important to note that when PySpark executes the `filter()` command combined with `like()`, it translates this operation into highly optimized instructions that run across the distributed cluster. This ensures that even when filtering petabytes of data, the operation remains efficient. This filtering mechanism is fundamentally different from a simple equality check (==) because it involves a more complex comparison logic that evaluates the string character by character against the defined pattern, often resulting in a small performance overhead compared to exact matches, though it is significantly optimized in modern Spark versions.

Essential Syntax of the PySpark `like` Operator

The syntax for applying the **LIKE operator** in PySpark is concise and follows standard SQL conventions, making it intuitive for users familiar with database querying. To apply this filter, you must first access the column object on which the pattern match should occur. Following this, the `.like()` method is called, taking the desired pattern string as its single argument. The result of this operation is a Boolean column indicating which rows satisfy the pattern, which is then passed to the overarching `filter()` transformation.

The general structure for filtering a DataFrame (`df`) based on a column (e.g., `df.column_name`) using a specific pattern is demonstrated below. This specific example targets rows where the string in the **team** column contains the sequence "avs" anywhere within it.

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

This particular command is a powerful instruction. It tells PySpark to scan the **team** column across all partitions of the DataFrame and retain only those rows where the content of the string field matches the pattern defined by the `%avs%` literal. The placement of the `%` symbols on both sides ensures that rows containing "Mavs", "Cavs", or even hypothetical entries like "GravelyAvsTeam" would be included in the resulting filtered output, illustrating the flexible nature of the **LIKE operator**.

Furthermore, while the above syntax uses the column object notation (`df.column_name.like()`), PySpark also supports the SQL expression method, which can sometimes offer greater flexibility when dealing with complex queries. Alternatively, you could use `df.filter("team LIKE '%avs%'").show()`, which achieves the identical result by passing a raw SQL string to the filter function. However, using the native column method (`df.team.like()`) is generally preferred in Python environments as it integrates better with the DataFrame API and provides slightly clearer intent regarding the object-oriented nature of the operation.

Setting Up the Environment and Sample DataFrame

To demonstrate the practical application of the **LIKE operator**, we must first establish a SparkSession and create a sample DataFrame. This DataFrame will simulate real-world data, in this case, tracking points scored by various professional basketball teams. Setting up the environment correctly is the foundational step for any data manipulation task in PySpark, ensuring that the necessary libraries are imported and the Spark context is initialized.

The following comprehensive code block initiates the SparkSession, defines the input data (a list of lists representing team names and corresponding points), specifies the schema via column names, and finally constructs the DataFrame. Reviewing the initial output allows us to confirm the structure and content before applying any filtering logic. This initial visualization is critical for verifying that

the subsequent filtering operations yield the expected subset of data.

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

# Define sample data for demonstration
data = ,
,
,
,
,
,
,
,
]

# Define column names
columns =

# Create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

# View the initial DataFrame structure and content
df.show()
```

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

This initial DataFrame contains 10 records, encompassing five distinct team names: Mavs, Nets, Lakers, Cavs, Wizards, and Spurs. Our goal is to isolate only those teams whose names contain the specific sequence "avs". By inspecting the output above, we can manually predict that the 'Mavs' (Dallas Mavericks) and 'Cavs' (Cleveland Cavaliers) entries should be the only ones retained after the filter is applied. This preparatory step ensures that we have a reliable starting point for our pattern-matching exercise and validates the data quality before moving on to the filtering step.

Practical Application: Filtering Data Using `%avs%`

Now that the sample data is prepared, we can execute the filtering command using the `like()` function. We specifically seek to filter the DataFrame such that only rows where the **team** column contains the pattern "avs" somewhere in the string are returned. This operation effectively uses the LIKE operator to perform substring matching across the distributed dataset, demonstrating a powerful technique for cleaning or subsetting data based on partial textual identifiers.

The command below concisely expresses this requirement. It invokes the `filter()` method on our DataFrame `df`, applies the `like()` condition to the `team` column with the pattern `'%avs%'`, and then displays the resulting filtered DataFrame using `.show()`.

 # 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|
+----+------+

Upon reviewing the output, it is immediately evident that only the rows corresponding to 'Mavs' and 'Cavs' have been retained. The pattern `'%avs%'` successfully matched 'Mavs' (since 'avs' is contained within the string) and 'Cavs' (since 'avs' is also contained within that string). Importantly, entries for 'Nets', 'Lakers', 'Wizards', and 'Spurs' were successfully excluded because none of those team names contain the literal sequence "avs". This result confirms the correct implementation of the partial string match using the **LIKE operator** in PySpark.

Mastering PySpark Wildcards: % and _

While the percentage symbol (%) is the most frequently used wildcard for general substring searching, the LIKE operator also supports another crucial wildcard: the underscore (_). Understanding the difference between these two symbols is essential for defining precise and effective pattern-matching criteria in PySpark. The % symbol, as previously discussed, matches any sequence of zero or more characters, offering maximum flexibility. Conversely, the _ (underscore) matches exactly one single character, making it ideal for fixed-length or positional searching.

Consider a scenario where you are searching for product codes that follow a strict format, such as three letters followed by a hyphen and then two numbers, like 'ABC-12'. If you wanted to find all codes that start with 'A' and end with '2', regardless of the intervening characters, you could use the pattern `'A__-__2'`. In this pattern, each underscore specifically requires the presence of exactly one character at that position. If you had mistakenly used `'A%2'`, it would match codes of any length starting with A and ending with 2, which would be far less precise than intended.

Furthermore, mastering the combination of both wildcards allows for highly nuanced queries. For instance, if we wanted to find all teams whose name is exactly four characters long and contains 'a' as the second character, we could use the pattern `'_a__'`. This would match 'Mavs' and 'Cavs' but would exclude longer names like 'Lakers' or shorter names. Developers should also be aware of the need to escape these wildcard characters if they are searching for the literal characters '%' or '_' within the data itself. While the specific escaping mechanism depends on the underlying SQL dialect Spark is using, typically a backslash () is used to treat the wildcard as a standard character, such as in the pattern `'100%'` to find the string "100%".

Comparison: `like` vs. Regular Expressions (`rlike`)

While the `like()` function is robust for simple wildcard-based pattern matching, PySpark also provides the `rlike()` function (also accessible as `regexp_extract` or `regexp_replace`) for more complex filtering needs that require the full power of regular expressions. It is essential for data practitioners to understand the trade-offs between these two methods to select the most appropriate tool for a given task.

The primary advantage of `like()` is its simplicity and direct alignment with standard SQL. It is extremely fast for basic prefix, suffix, and infix searches using the % and _ wildcards. If your requirement is simply to check if a substring exists or if a string starts or ends with a certain sequence, `like()` is the preferred choice due to its better readability and optimized performance for these simple cases. It avoids the potentially steep learning curve associated with complex regular expression syntax.

However, `rlike()` becomes necessary when the pattern complexity exceeds what the simple SQL

wildcards can handle. Regular expressions allow for filtering based on criteria such as: the presence of specific character sets (e.g., only alphanumeric characters), enforcing specific counts of characters (e.g., exactly 3 digits), or checking for alternative patterns (e.g., 'Team A' OR 'Team B'). Because regular expression evaluation is inherently more complex than wildcard matching, using `rlike()` can sometimes incur a greater computational cost. Therefore, the best practice is to always default to `like()` unless the expressive power of regular expressions is explicitly required to define the filtering criteria accurately.

Additional Resources

For users seeking to expand their knowledge of data manipulation and filtering techniques within the PySpark environment, the following resources provide valuable information on related common tasks:

Official PySpark documentation for the `like` function.

Tutorials explaining how to use Boolean logic (AND, OR, NOT) to combine multiple filter conditions.

Guides detailing performance tuning strategies for distributed filtering operations on large datasets within Apache Spark.

Examples illustrating the use of `rlike()` for advanced text parsing and cleaning operations.

By mastering the use of the **LIKE operator**, data engineers gain a crucial tool for efficient and readable pattern-based data filtering, forming a core competency in working with structured data on the Spark platform.