How to Filter Rows in PySpark DataFrames Using the LIKE Operator

Authored by

stats writer

February 6, 2026

The **LIKE operator** in **PySpark** is an indispensable tool utilized for filtering rows within a **DataFrame** based on the existence of a specific pattern or substring. Unlike exact match filtering, the **LIKE operator** facilitates powerful partial matching capabilities, which makes it exceptionally valuable during sophisticated data cleaning, validation, and exploratory data analysis processes. While traditional SQL implementations often rely on converting the DataFrame into a temporary view using methods like `.createOrReplaceTempView()` to enable direct **SQL query** execution, PySpark provides a more idiomatic Python approach through the dedicated `.like()` function on DataFrame columns. This method allows analysts to efficiently pinpoint records that contain specific words, phrases, or complex character sequences, offering a versatile and highly optimized mechanism for data manipulation within the distributed computing environment of Spark.

PySpark: Filter Rows Using LIKE Operator

Understanding PySpark Filtering and the LIKE Operator

The ability to efficiently filter large datasets is fundamental to working with **PySpark**. When dealing with unstructured or semi-structured textual data, simple equality checks often prove insufficient. This is where the **LIKE operator**--a concept borrowed directly from standard **SQL**--becomes essential. It enables pattern matching, allowing data engineers and scientists to search for substrings within string columns, greatly enhancing the flexibility of data retrieval operations. Implementing pattern matching directly on a distributed **DataFrame** ensures that these computationally intensive tasks are executed efficiently across the Spark cluster.

In the **PySpark** framework, filtering is typically achieved using the `.filter()` transformation, which accepts a column expression representing the condition to be evaluated for each row. When applied to string columns, this expression can incorporate the `.like()` method. This function operates identically to the traditional **LIKE operator**, relying on specific wildcard characters to define the search pattern. Understanding the precise usage of these wildcards is critical to successfully implementing accurate and performant partial match filtering across massive datasets.

While PySpark offers other filtering options, such as using regular expressions via the `.rlike()` function, the `.like()` function is often preferred for simpler, common substring searches due to its straightforward syntax and high readability. Furthermore, in scenarios where the DataFrame might be queried using a mix of Python and SQL interfaces--a common practice in complex data pipelines--using the SQL-native **LIKE operator** ensures consistency across all querying methods, simplifying maintenance and debugging efforts. The following sections will detail the exact syntax and provide comprehensive examples for utilizing this powerful feature.

Syntax and Fundamentals of the PySpark like Function

The fundamental syntax for applying the **LIKE operator** within a **PySpark** **DataFrame** leverages the built-in column functions. Instead of constructing an entire SQL string, the modern PySpark approach chains the `.like()` method directly onto the targeted column object, which is then passed as an argument to the DataFrame's `.filter()` method. This concise structure is both highly readable and optimized for execution within the Spark environment.

The standard structure begins with the DataFrame object (`df`), followed by the `.filter()` transformation. Inside the filter, you reference the specific column (e.g., `df.column_name`) and invoke the `.like()` method, passing the desired pattern string as the argument. Crucially, the pattern string must incorporate the necessary SQL wildcards to define the boundaries and location of the substring match. The result of the filter operation is a new DataFrame containing only the rows where the specified column value matches the pattern defined by the `.like()` expression.

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

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

This particular example demonstrates how to filter the **DataFrame** to exclusively display rows where the string data held within the **`team`** column contains the substring "avs" located anywhere within the full string value. The surrounding percent signs (`%`) are the fundamental wildcards that dictate this partial matching behavior, ensuring that the filter captures results regardless of leading or trailing characters.

Understanding Wildcards: % and _ in PySpark LIKE

Mastering the **LIKE operator** requires a deep understanding of the two primary SQL wildcards: the percent sign (`%`) and the underscore (`_`). These symbols are not literals; instead, they serve as placeholders that define flexible matching conditions within the pattern string provided to the `.like()` function. Their correct placement determines whether the search is for a prefix, a suffix, or an embedded substring.

The `%` wildcard represents zero or more characters. It is the most commonly used wildcard for broad pattern matching. If you use `'A%'`, you are searching for any string that begins with the letter 'A'. If you use `'%Z'`, you are searching for any string that ends with the letter 'Z'. The pattern `'%query%'`, as utilized in our example, dictates that the substring "query" must appear at least once, with any number of characters allowed before or after it. This flexibility is what makes **PySpark** filtering so powerful for tasks like finding product names containing certain brand abbreviations or identifying customer comments mentioning specific keywords.

Conversely, the `_` wildcard represents exactly one single character. This is invaluable when the length of the string is important or when you need to match patterns where only one character varies. For example, the pattern `'c_t'` would match "Cat," "Cot," or "Cut," but it would strictly exclude "Cart" or "Court." While less frequently used than `%` in general substring searches, the `_` wildcard offers precise control for scenarios involving standardized codes, model numbers, or fixed-length identifiers where a specific position variation needs to be isolated. Combining both wildcards allows for the construction of highly complex and refined search expressions.

Setting up the PySpark Environment and Sample Data

Before any filtering operation can occur, a **PySpark** **DataFrame** must be initialized. This process begins with the establishment of a **SparkSession**, which serves as the entry point to utilizing the Spark functionality, including the ability to create and manipulate distributed datasets. The **SparkSession** manages the connection to the Spark cluster and handles the necessary context for executing distributed computations.

The following example illustrates the entire process, starting from the necessary imports and the instantiation of the **SparkSession**. We will then define a sample dataset containing basketball team names and their respective points scored, which is ideal for demonstrating string pattern matching. Defining the schema (column names) separately ensures the resulting **DataFrame** is correctly structured for subsequent filtering operations using the `.like()` function.

Suppose we have the following **PySpark** **DataFrame** that contains information about points scored by various basketball players. We use the standard setup script to generate this dataset:

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

Reviewing the output, we observe a mix of team names. Our goal is to isolate only those teams whose names contain the specific substring "avs", which should include "Mavs" and "Cavs." This initial visualization helps confirm the structure and content of the dataset before applying any transformations.

Applying the LIKE Filter: Step-by-Step Implementation

With the **DataFrame** successfully constructed, the next step is to apply the filtering logic using the `.filter()` method combined with the column `.like()` function. This operation is performed lazily by Spark, meaning the actual computation only executes when an action, such as `.show()`, is called. This efficiency is a core tenet of Spark's architecture.

We specifically target the **`team`** column and employ the pattern `'%avs%'`. As previously discussed, the leading and trailing percent signs ensure that the match is successful whether "avs" is a full word, a prefix, a suffix, or an embedded substring within the team name. Since both "Mavs" and "Cavs" contain this sequence, we anticipate that all rows associated with these two teams will be preserved in the resulting output DataFrame, while all others (Nets, Lakers, Wizards, Spurs) will be

excluded.

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

Notice that each of the rows in the resulting **DataFrame** successfully contain the string "avs" in the `team` column. This outcome confirms the correct application of the **LIKE operator** for general partial matching. This straightforward yet powerful technique is essential for initial data exploration or for isolating subsets of data based on non-exact string criteria.

Alternative Filtering Methods: SQL Context and RLIKE

While the native **PySpark** column expression `df.col.like('%pattern%')` is the recommended Pythonic approach, two other prevalent methods exist for pattern-based filtering: utilizing the **SQL query** context and employing the more powerful `.rlike()` function for regular expressions.

The first alternative involves using the native **SQL query** functionality built into Spark. To use this, the DataFrame must first be registered as a temporary view using `df.createOrReplaceTempView("temp_table")`. Once registered, analysts can execute standard SQL queries directly against the view, embedding the LIKE clause exactly as they would in any traditional relational database. This method is particularly useful when integrating Spark operations into pipelines heavily reliant on SQL syntax or when performing complex joins and aggregations simultaneously with the filter operation.

The second, and often more powerful, alternative is the `.rlike()` function. This function enables filtering using full **Regular Expressions**, providing capabilities far beyond the simple wildcard matching of the **LIKE operator**. If the filtering requirement involves complex character classes, repetition counts, or boundaries (e.g., finding whole words only, or matching specific phone

number formats), `.rlike()` is the necessary tool. However, due to the increased computational complexity of regular expression engines, `.rlike()` can sometimes be slower than the basic `.like()` function for simple substring checks. Therefore, choosing between `.like()` and `.rlike()` should be based on the complexity of the pattern required versus the performance constraints of the application.

Best Practices for LIKE Filtering in Distributed Environments

When working with massive datasets in **PySpark**, performance and resource utilization are paramount. Although the `.like()` function is optimized for distributed processing, poor implementation can still lead to bottlenecks. A key best practice is to always push the filter operation down to the data source as early as possible in the processing pipeline. Applying the `.filter()` transformation early reduces the amount of data that needs to be moved and processed in subsequent steps, significantly improving execution time and resource consumption across the cluster.

Another crucial consideration involves case sensitivity. By default, string comparisons in Spark SQL (which underlies the **LIKE operator**) are generally case-insensitive in many configurations. However, this behavior can vary based on the underlying file system and Spark configuration parameters (like `spark.sql.caseSensitive`). To guarantee predictable results, especially when searching for patterns in mixed-case data, it is highly recommended to explicitly normalize the column data (e.g., converting to lowercase using `df.col.lower()`) before applying the `.like()` filter. For example: `df.filter(df.team.lower().like('%avs%'))` ensures that both "Mavs" and "MAVS" are correctly identified.

Finally, avoid overly complex or highly fragmented patterns when using the `%` wildcard unnecessarily, as this can degrade performance. While the **DataFrame** API is optimized, relying on indexable prefixes or suffixes when possible (e.g., using `'Mav%'` instead of `'%Mavs%'` if you know the pattern starts at the beginning) can allow the query optimizer to potentially employ more efficient strategies for filtering, especially if the underlying storage layer supports partition pruning or indexing based on prefix searches.

Conclusion and Further PySpark Resources

The **LIKE operator**, implemented via the `.like()` function in **PySpark**, provides an essential mechanism for performing efficient partial string matching within large distributed **DataFrames**. By mastering the use of the `%` and `_` wildcards, developers can construct powerful and flexible filter conditions necessary for sophisticated data preparation and analysis tasks. This method balances simplicity of use with the high performance demanded by modern big data processing pipelines.

For users who require more intricate pattern matching capabilities beyond basic wildcards, the `.rlike()` function offers the full power of regular expressions, serving as a comprehensive fallback. However, for most common use cases involving simple substring identification, the `.like()` method remains the cleanest and often the most performant choice within the PySpark environment.

Note: You can find the complete documentation for the **PySpark** `like` function at the official Apache Spark documentation site. It is always recommended to consult the official documentation for the latest API specifications and version-specific details.

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

How to perform joins in PySpark.
Using aggregation functions like `groupBy` in PySpark.
Writing custom User Defined Functions (UDFs) for DataFrames.
Converting RDDs to DataFrames in Spark.