Learning PySpark: How to Filter Rows Based on Multiple Values

Authored by

Mohammed looti

November 10, 2025

Mastering Complex Filtering in PySpark DataFrames

The efficient manipulation of large-scale data is the cornerstone of modern data engineering, and filtering stands out as one of the most frequently executed operations within PySpark DataFrames. While applying filters based on simple, exact equality checks is straightforward, significant complexity arises when the requirement mandates searching a column for the presence of one of several potential **substrings**. This challenge is pervasive in real-world scenarios, particularly when dealing with unstructured text or columns containing diverse identifiers where standard exact matching is insufficient.

Consider a situation where a data scientist needs to extract all rows where a product description contains "aluminum," "steel," or "titanium." Attempting to solve this using traditional methods in a distributed environment--such as chaining dozens of individual `df.col.contains("value1") | df.col.contains("value2")` conditions--results in code that is both cumbersome to write and severely inefficient for the underlying Apache Spark engine to optimize. This verbose, repetitive approach leads to poor execution plans and slow processing times when scaled across petabytes of data.

To overcome the performance limitations and logical complexity of chained `OR` statements, Apache Spark provides a powerful, scalable alternative: utilizing a single, consolidated search pattern powered by regular expressions. This technique simplifies the filtering logic, centralizes the search criteria, and crucially, allows the distributed framework to execute the operation with optimal efficiency, making it the preferred method for multi-value substring matching.

Leveraging Regular Expressions for Distributed Substring Matching

The most powerful and performant method for filtering a column based on the presence of multiple possible substrings involves adopting **regular expressions (regex)**. Regular expressions offer a highly flexible mechanism for defining complex patterns that must be matched within a string. PySpark seamlessly integrates this capability through the dedicated column method, rlike, which stands for "regex like." This function is designed to apply sophisticated regex patterns across the distributed partitions of a DataFrame, returning only those rows where a successful pattern match occurs.

The key to searching for multiple values simultaneously within a single column is the logical OR functionality inherent in regex syntax. This is achieved using the pipe symbol (**|**). By preparing a single string where all desired substrings are listed and separated by this pipe symbol, we construct a composite pattern that effectively says: "Match term A OR term B OR term C." For instance, if the search terms are "cat," "dog," and "bird," the resulting regex string passed to `rlike` would be `"cat|dog|bird"`. This consolidated pattern allows the Spark optimizer to handle the entire complex search as a single, unified operation.

This approach drastically improves code maintainability and execution speed compared to manually generating hundreds of sequential `OR` conditions. Furthermore, the dynamic generation of the regex pattern using standard Python string methods ensures that the solution is highly adaptable. As the list of required search terms evolves, the underlying filtering logic requires no alteration, making this technique an essential tool for robust data pipelines built on the PySpark DataFrame API.

Preparation Phase: Transforming a List into a Regex Pattern

Before the filtering operation can be executed within the distributed Spark context, it is necessary to prepare the search criteria using core Python logic. This preparation ensures that the input provided to the `rlike` function is a single, syntactically correct regular expression (regex) string that the distributed framework can interpret efficiently. This process involves two critical steps: defining the collection of search terms and then formatting this collection into the required OR-separated string.

First, we initialize a standard Python list containing all the partial strings we aim to locate within the target DataFrame column. Second, and most importantly, we utilize the Python string object's `.join()` method. By specifying the pipe symbol (**|**) as the joining delimiter, this operation transforms the iterable list of terms into the necessary OR-separated regex string. This single string is the crucial input required by the `rlike` function for execution across the distributed dataset.

The following example illustrates this essential preparation stage. We define the list of values and then construct the composite regex string, which acts as the filter predicate in the subsequent PySpark operation:

 # Define the array (list) of substrings we intend to search for
my_values =
 # Construct the regex string using the pipe (|) as the OR operator
regex_values = "|".join(my_values)

# Apply the filter: search the 'team' column for any match in the regex string
df.filter(df.team.rlike(regex_values)).show()

Demonstration: Setting Up the PySpark Environment

To concretely demonstrate the efficacy of the `rlike` filtering technique, we must first establish a representative PySpark DataFrame. The sample dataset we will use simulates basketball score data, comprising two simple columns: the team name (**team**) and the points scored (**points**). This clean, structured environment allows us to clearly observe how the multi-substring filter isolates the

target rows.

The setup involves the standard preliminary steps necessary for any PySpark script: initiating a Spark session, defining the raw data (as a list of tuples), and specifying the column schema. By defining the schema explicitly, we ensure that the data types are correctly assigned upon creation, preparing the data for optimized distributed processing before any complex filtering logic is applied. This methodical setup guarantees that the filtering demonstration accurately reflects real-world PySpark usage.

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

# Define the sample data: a list of team names and their points
data = ,
,
,
,
,
,
,]

Define the column schema
columns =

# Create the PySpark DataFrame from the data and schema
df = spark.createDataFrame(data, columns)

# Display the initial state of the DataFrame
df.show()

+------+------+
| team|points|
+------+------+
Mavs	14
Nets	22
Nets	31
Cavs	27
Kings	26
Spurs	40
Lakers	23

| Spurs| 17|
+------+------+

Executing the rlike Filter and Analyzing Results

The defined sample DataFrame is now ready for the application of our sophisticated substring filter. The goal is to isolate rows where the **team** column contains either the substring "ets" (present in 'Nets') or the substring "urs" (present in 'Spurs'). As established in the preparation phase, the required composite pattern is the regular expression (regex) string `'ets|urs'`. This pattern is passed directly into the rlike function applied to the target column.

When the filter is executed, the Apache Spark engine leverages its distributed architecture to scan the column in parallel across all nodes, searching for the presence of either of the specified substrings defined by the logical OR operator. This methodology offers significant performance advantages over non-regex based string searching, especially as the data volume and the number of search terms grow. The declarative nature of the regex pattern ensures that the filtering logic is precise and highly optimized for mass parallel execution.

 # Define array of substrings
my_values =
regex_values = "|".join(my_values)

# Filter DataFrame where team column contains any substring defined in 'ets|urs'
df.filter(df.team.rlike(regex_values)).show()

+-----+------+
| team|points|
+-----+------+
| Nets| 22|
| Nets| 31|
|Spurs| 40|
|Spurs| 17|
+-----+------+

The resulting DataFrame successfully isolates all rows corresponding to the 'Nets' and 'Spurs' entries. A crucial observation is that the filter functioned based solely on the existence of the partial substring within the column's value; it did not require the column value to equal the substring exactly. This confirms that dynamic regex generation combined with the rlike function is the robust and scalable solution for multi-value substring filtering within PySpark.

Choosing the Right Tool: rlike vs. isin()

While the rlike function is the definitive tool for powerful partial matching and complex pattern searches, it is vital for developers to select the appropriate PySpark function based on the filtering requirement. If the objective shifts from searching for a substring to filtering for rows that match one of several **exact, whole-string** values (e.g., finding rows where the team name is precisely 'Mavs' OR 'Cavs'), the regular expression approach should be avoided in favor of the optimized isin() function.

The `isin()` function is specifically engineered for efficient set membership checks against a collection of known values, offering superior performance for exact-match scenarios across the distributed PySpark DataFrame. However, when partial matching or complex text pattern recognition is required, `rlike` remains the necessary and dominant function. To ensure peak performance when using regular expressions, always prioritize clarity and efficiency in pattern definition. Developers should avoid overly complicated patterns where simpler OR syntax suffices, ensuring the pattern remains highly declarative and easily optimizable by the underlying Apache Spark execution engine.

Conclusion and Resources for Advanced PySpark Techniques

The ability to dynamically construct and apply specialized filters for multi-value substring matching is a fundamental skill for advanced data analysis and preparation in PySpark. By leveraging the power of regular expressions and the efficient OR syntax provided by the pipe symbol, we can transform a potentially slow and verbose operation involving chained `OR` conditions into a single, highly performant distributed query using the `rlike` function. This approach is essential when dealing with the ambiguity and variability often found in large, real-world text-heavy datasets.

Mastery of PySpark requires understanding not just how to apply filters, but knowing which filter to apply for maximum performance. Whether the requirement calls for the optimized set membership checking of isin() or the flexible pattern matching of `rlike`, choosing the correct method ensures scalable and efficient data processing. The following resources offer further guidance on crucial PySpark tasks necessary for continued skill development:

Tutorial on using the `isin()` function for exact matches across multiple columns.

Guide to leveraging PySpark's User Defined Functions (UDFs) for custom logic.

Detailed explanation of PySpark Window Functions for aggregation and ranking.