PySpark: Filter for Rows that Contain One of Multiple Values

Authored by

stats writer

November 16, 2025

Introduction to Multi-Value Filtering Challenges

Working with massive datasets often requires highly specific filtering operations. In PySpark, developers frequently need to select rows where a specific column contains one of several defined substrings. While simple equality checks are straightforward using functions like `where` or `filter` with standard comparison operators, handling partial string matches against a dynamic list of values presents a unique challenge. Standard methods, such as using multiple `.contains()` checks linked by `|` (OR operators), quickly become cumbersome and inefficient, especially when the list of required substrings grows large. This manual linking approach is not scalable and makes code difficult to maintain, prompting the need for a more robust and streamlined solution capable of handling complex string logic efficiently within the distributed environment of PySpark.

The core difficulty lies in translating an array or list of search terms into a single, cohesive filtering condition that the Spark engine can optimize effectively. When dealing with unstructured or semi-structured data, exact matches are rare, and finding records that partially match a set of criteria is essential for data cleansing, classification, or feature extraction tasks. Therefore, mastering the technique for filtering a DataFrame based on whether a column's value contains **any** element from a predefined list of substrings is a fundamental skill for any data engineer working with Spark. This technique leverages the power of regular expressions, providing a flexible and high-performance mechanism for string pattern matching across the cluster.

This guide focuses on the most effective method in PySpark for achieving this multi-value substring filtering: utilizing the specialized `rlike` function. We will explore how to dynamically construct a single regular expression pattern from a standard Python list, allowing the Spark engine to perform a single, powerful pattern matching scan across the target column. This approach drastically simplifies the code structure compared to repetitive conditional logic and provides performance benefits by minimizing the number of predicate pushdowns required for filtering operations.

The Power of Regular Expressions in DataFrames

Regular expressions (or **regex**) are indispensable tools in data processing, offering sophisticated ways to search, match, and manipulate strings based on complex patterns. Unlike simple string functions that look for literal sequences, regex allows for the definition of rules, character classes, repetitions, and alternatives. In the context of large-scale data processing using PySpark, leveraging regex ensures that filtering logic remains concise yet powerful enough to handle intricate matching requirements efficiently across a distributed DataFrame.

When attempting to check if a column contains one of multiple possible substrings, the key regex operator is the alternation symbol: `|`. This operator functions similarly to an "OR" condition within the pattern itself. By combining all desired search terms using this pipe symbol, we create a unified

pattern that matches any string containing 'TermA' **OR** 'TermB' **OR** 'TermC'. This mechanism is central to solving the multi-value filtering problem efficiently because it allows us to pass a single, composite pattern to the Spark filtering function, rather than multiple separate conditions.

The implementation of regex in Spark is highly optimized. When Spark executes the filtering condition involving a regex pattern, it utilizes its underlying catalyst optimizer to push this operation down to the execution engine (like the Java Virtual Machine or JVM). This means the complex pattern matching occurs at the distributed worker nodes, minimizing data movement and maximizing performance. Understanding how to dynamically construct and deploy these patterns is the difference between writing slow, iterative code and writing fast, scalable Spark applications.

Understanding the PySpark rlike Function

The specific function PySpark provides for regex matching is rlike (regular expression like). This function is a member of the `Column` class in `pyspark.sql` and is designed to evaluate a string column against a given regex pattern. It returns a boolean result: **True** if the string matches the pattern, and **False** otherwise. It is crucial to distinguish rlike from other string functions like `like`, which only supports standard SQL wildcard matching (using `%` and `_`), or `contains`, which only searches for a single literal substring.

The syntax for using rlike is straightforward: `df.column_name.rlike(pattern)`. The real complexity, and the source of its power for multi-value filtering, lies in generating the `pattern` string itself. For instance, if we wanted to find rows where the 'name' column contains either 'John' or 'Jane', the pattern passed to rlike would be `"John|Jane"`. The rlike function is essential because it allows us to leverage the alternation feature of regex to test against numerous potential matches simultaneously within a single computational step.

Furthermore, unlike simple string searching functions, rlike is **case-sensitive** by default. If case-insensitivity is required, the regex pattern must be modified using appropriate flags, such as `(?i)` at the start of the pattern. This level of control over the matching process makes rlike the preferred method for advanced string manipulation tasks in PySpark, allowing engineers to precisely define the boundaries and conditions for a successful match, far beyond what simple equality or containment checks can offer.

Step-by-Step Implementation: Preparing Search Values

The first critical step in implementing multi-value filtering is converting the standard Python list of desired search terms into the required regex format. If we start with a list of strings, say , we need to transform this into a single string: `"apple|banana|cherry"`. This transformation relies heavily on Python's join() method, which is specifically designed to concatenate list elements using a specified delimiter. In our case, the delimiter must be the regex alternation operator `|`.

The implementation is surprisingly simple yet highly effective. We define the list of values we wish to search for, and then we apply Python's join() method to concatenate them, using `"|"` as the joining string. This method ensures that the final resulting string is a syntactically correct regex pattern that can be directly consumed by the rlike function. This approach is highly dynamic: if the list of search values changes (e.g., loaded from a configuration file or a database), the code handles the pattern generation automatically without requiring manual string manipulation or conditional updates.

Consider the initial logic provided below, which demonstrates this exact preparation technique. The list `my_values` holds the desired substrings, and the resulting `regex_values` variable becomes the single, powerful pattern used for the filtering operation. This two-step process--defining the list and then joining it--is the standard, clean method recommended for implementing dynamic multi-value substring filtering in PySpark.

#define array of substrings to search for
my_values =
regex_values = "|".join(my_values)

filter DataFrame where team column contains any substring from array
df.filter(df.team.rlike(regex_values)).show()

Code Example: Creating the Sample DataFrame

To illustrate the application of this technique, we will first construct a sample DataFrame. This dataset, representing basketball scores, contains two columns: `team` (a string column where we will perform the substring search) and `points` (an integer column). The dataset is intentionally designed to include entries where the search substrings appear in the middle of the team name (e.g., 'Nets' contains 'ets', 'Spurs' contains 'urs'), demonstrating the partial matching capability of `rlike`.

We initialize a **SparkSession**, which is the entry point for using Spark functionality, and then define our data structure. This structure is a simple list of lists, where each inner list represents a row of data. Following the data definition, we define the corresponding column names, ensuring clarity and structure within the resulting DataFrame object. This setup mimics a common scenario where raw data is ingested before being loaded into a persistent storage layer.

Finally, we use the `spark.createDataFrame()` method to materialize the data structure into an actual DataFrame (`df`). Displaying the initial DataFrame allows us to visualize the unfiltered data, providing a clear baseline against which the filtered results can be compared, confirming the accuracy of our subsequent filtering logic.

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	14
Nets	22
Nets	31
Cavs	27
Kings	26
Spurs	40
Lakers	23
Spurs	17
+------+------+

Applying the Multi-Value Filter Using rlike

With the sample DataFrame established, we proceed to implement the core filtering logic. Our objective is to isolate only those rows where the `team` column contains either the substring "ets" or the substring "urs." As previously discussed, simply chaining multiple `.contains()` calls would be

inefficient and difficult to manage; instead, we leverage the dynamic regex pattern generation.

We first define `my_values` as the list of desired substrings: . Then, we use Python's join() method with the pipe character `|` to create the combined regular expression pattern, `regex_values`, which evaluates to `"ets|urs"`. This single string represents the full "OR" condition required for our search. This preparation step ensures that the final filtering function receives a concise and highly optimized search instruction.

The final step is applying this pattern using the rlike function within the `filter` clause. We instruct Spark to filter the DataFrame where the `df.team` column matches the generated regex pattern. The resulting output demonstrates the efficacy of this approach, successfully extracting only those rows corresponding to the 'Nets' (containing 'ets') and 'Spurs' (containing 'urs') entries, confirming that the partial string matching across multiple values worked correctly and simultaneously.

#define array of substrings to search for
my_values =
regex_values = "|".join(my_values)

filter DataFrame where team column contains any substring from array
df.filter(df.team.rlike(regex_values)).show()

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

Analyzing the Results and Performance Considerations

Upon examining the filtered results shown above, it is clear that only four rows remain, corresponding to 'Nets' and 'Spurs'. Every record in this subset satisfies the condition defined by the regex pattern `"ets|urs"`. The key takeaway here is that the rlike function successfully interpreted the pipe symbol `|` as a logical OR operator, allowing the filtering to happen in a single, unified operation rather than a series of sequential checks. This validation confirms the methodology is sound for complex substring searches.

From a performance perspective, utilizing `rlike` with a concatenated regex pattern is generally superior to using multiple chained `.contains()` or `.like()` functions for dynamic multi-value

partial matching. When Spark's Catalyst Optimizer processes a single, combined regex, it can execute the pattern matching using highly optimized internal regular expression engines (like the Java implementation) on the underlying partitions. This avoids the overhead associated with combining multiple separate boolean column expressions with OR operators, which can sometimes lead to less optimal execution plans, especially as the number of search terms grows.

However, it is important to note the complexity trade-off. While powerful, complex regex patterns can sometimes be computationally intensive. If the list of search values is exceptionally long (hundreds or thousands of terms), or if the terms themselves require extensive backtracking (though unlikely for simple substring matching), there might be a minor performance degradation compared to highly specialized functions. For most practical scenarios involving a moderate number of search terms for substring containment, the rlike and Python's join() method combination remains the cleanest and most efficient approach for dynamic multi-value filtering in PySpark DataFrames.

Alternatives to rlike: When to Use isin or contains

While the `rlike` function is the gold standard for dynamic partial string matching, it is essential to understand when other PySpark filtering functions might be more appropriate. The choice depends entirely on whether you are looking for an exact match, a partial match, or a match against a set of values. If the goal is to filter rows where the entire column value is **exactly** one of the values in a list (e.g., team equals 'Nets' OR team equals 'Mavs', but not 'TeamNets'), the `isin()` function is the most efficient and recommended tool.

The `isin()` function is specifically optimized for checking membership against a list of exact values and should be prioritized whenever possible, as it avoids the computational overhead of regex processing. For instance, `df.filter(df.team.isin())` is much faster than using rlike for exact matches. Conversely, if you only need to check for one single substring occurrence, the `contains()` function (e.g., `df.filter(df.team.contains('Nets'))`) is simpler to read and marginally more performant than using rlike for that singular purpose, although it loses the flexibility of handling multiple values dynamically.

Therefore, the decision matrix is clear: use `isin()` for lists of exact matches, use `contains()` for single, literal substring searches, and reserve the rlike function, paired with Python's join() method, specifically for scenarios requiring the power of regex, such as searching for multiple dynamic substrings simultaneously, or when implementing complex patterns like word boundaries (`b`) or lookaheads, which are beyond the capabilities of simpler string functions.

Conclusion: Best Practices for String Matching in PySpark

Efficiently filtering DataFrames based on complex string criteria is a core requirement in modern

data engineering. For the common task of finding rows that contain any one of multiple possible substrings, the combination of Python's join() method and PySpark's rlike function provides a scalable, clean, and highly performant solution. By dynamically constructing a single regex pattern using the alternation operator `|`, we dramatically simplify the filtering logic and enable Spark's optimizer to execute the search optimally across the cluster.

Key takeaways for best practice include always defining your search terms clearly in a list and utilizing the `"|".join(my_list)` structure to prepare the pattern. This methodology ensures maintainability, as adding or removing search terms only requires modifying the initial Python list, without altering the filter expression itself. Furthermore, always be mindful of when to switch strategies: if exact matching is needed, prioritize `isin()` over rlike for better performance.

Mastery of rlike and regex integration is essential for advanced data manipulation tasks in PySpark. This technique not only solves the immediate problem of multi-value substring filtering but also opens the door to far more sophisticated pattern-based analysis, enabling richer and more nuanced data extraction from large, distributed datasets.