PySpark: Filter for “Not Contains”

Authored by

stats writer

November 16, 2025

Introduction to String Filtering in PySpark

When working with large datasets, the ability to selectively include or exclude rows based on string patterns is fundamental. In the PySpark framework, complex data manipulation tasks, such as filtering, are efficiently handled using the powerful capabilities of the DataFrame API. While standard filtering often involves matching specific values or ranges, a common requirement in data cleaning and analysis is performing negative matching--that is, selecting rows that expressly do **not** contain a specific substring.

Achieving a "Not Contains" operation is crucial for situations where data quality needs improvement or when excluding highly specific categories or identifiers from the analytical scope. This process is distinct from exact matching because it searches for partial string existence within a column's text values. Understanding how to negate the standard string matching functions is key to mastering expressive and efficient filtering logic within Spark's distributed environment.

This guide will demonstrate the precise syntax required to implement this "Not Contains" logic in PySpark, focusing specifically on negating the built-in `.contains()` function. We will walk through a practical example, creating a sample DataFrame and applying the filter to illustrate how rows meeting the exclusionary criteria are successfully removed, resulting in a cleaner, targeted dataset ready for further processing.

The Essential Syntax for "Not Contains" Operations

The mechanism for filtering a DataFrame using a "Not Contains" condition relies on applying a logical negation operator to the output of the standard string search function. In Python and, by extension, PySpark, the tilde symbol (~) serves as the negation operator when applied to Spark Column objects. This symbol reverses the boolean outcome of the condition it precedes.

The standard method for checking if a string column includes a specific substring is the `.contains()` method. This method returns `True` for rows where the substring is found and `False` otherwise. To implement the "Not Contains" logic, we simply apply the negation operator `~` directly before the column expression utilizing `.contains()`. This flips the results: rows that originally returned `True` (meaning they contain the substring) now return `False`, and those that returned `False` now return `True`.

The following syntax represents the clean, efficient way to apply this negative filter using the DataFrame API. This structure ensures that only records where the specified column value explicitly does not include the target substring are retained in the resulting DataFrame.

#filter DataFrame where the 'team' column does not contain 'avs'
df.filter(~df.team.contains('avs')).show()

Notice the crucial placement of the `~` operator. It is applied to the entire expression `df.team.contains('avs')`, effectively generating a boolean mask where `True` indicates rows to keep (those that do not contain 'avs') and `False` indicates rows to discard (those that do contain 'avs').

Setting Up the PySpark Environment and Sample Data

To effectively demonstrate the "Not Contains" filtering technique, we must first initialize a SparkSession and create a sample dataset. This dataset will represent typical sports statistics, including team names--some of which share common letter combinations--and their corresponding point totals. Using a controlled dataset allows us to verify the exact behavior of the exclusionary filter.

The data preparation phase involves importing necessary modules, defining the raw data structure (a list of lists), specifying the column schema, and finally, instantiating the DataFrame using `spark.createDataFrame()`. This standard procedure ensures that our data is correctly structured and ready for distributed processing by the Spark engine. We are particularly interested in the 'team' column, as it contains the strings we will be targeting with our filter.

The following code block outlines the steps necessary to generate and view our initial dataset. Pay close attention to the teams: 'Mavs' and 'Cavs' both contain the substring 'avs', while other teams like 'Nets', 'Kings', 'Spurs', and 'Lakers' do not. These differences are key to testing the filter's accuracy.

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

#define data containing team names and points
data = ,
,
,
,
,
,
,]

#define column names
columns =

#create dataframe using data and column names

```
df = spark.createDataFrame(data, columns)

#view original dataframe
df.show()

+------+------+
| team|points|
+------+------+
| Mavs| 14|
| Nets| 22|
| Nets| 31|
| Cavs| 27|
| Kings| 26|
| Spurs| 40|
|Lakers| 23|
| Spurs| 17|
+------+------+
```

Understanding the Sample DataFrame Structure

The resulting DataFrame, `df`, is a tabular representation of our sports data, consisting of eight rows and two columns: `team` (string) and `points` (integer). Before applying any filtering logic, it is essential to clearly identify which rows are expected to be included and which are expected to be excluded based on our target substring, 'avs'.

The rows containing 'Mavs' and 'Cavs' are the primary candidates for exclusion. When we apply the `.contains('avs')` function, these rows will return `True`. Conversely, the rows containing 'Nets', 'Kings', 'Spurs', and 'Lakers' will return `False` because the substring 'avs' is not present. This distinction is paramount for validating the success of the negation operator.

Our goal using the "Not Contains" filter is to generate a new DataFrame that retains only the rows where the team name does **not** contain 'avs'. This means we are targeting an output DataFrame containing only the data points associated with the Nets, Kings, Spurs, and Lakers. This process highlights how simple, yet powerful, string manipulation can be when combined with boolean logic in PySpark.

Implementing the PySpark "Not Contains" Filter

Now we proceed to implement the actual filtering operation. We utilize the `.filter()` transformation available on all DataFrame objects, supplying it with the negated condition. This

operation is lazy, meaning the calculation is not performed until an action, such as `.show()`, is called.

The expression `~df.team.contains('avs')` is passed as the argument to the filter function. The inner function, `.contains('avs')`, checks for substring presence. The outer `~` operator flips the result, ensuring that only rows where the substring is absent (i.e., the original `.contains()` result was `False`) are selected and included in the resultant DataFrame.

Executing this command triggers Spark to process the data across the cluster, efficiently identifying and excluding the matching rows. The resulting DataFrame, shown below, visibly confirms the operation's success by presenting only the non-matching records.

#filter DataFrame where team does not contain 'avs'
df.filter(~df.team.contains('avs')).show()

```
+------+------+
| team|points|
+------+------+
| Nets| 22|
| Nets| 31|
| Kings| 26|
| Spurs| 40|
|Lakers| 23|
| Spurs| 17|
+------+------+
```

Analyzing the Filtered Results and Exclusion Logic

Upon reviewing the output of the filtering operation, we can immediately observe that the resulting DataFrame contains six rows, down from the original eight. Crucially, the records corresponding to the 'Mavs' (14 points) and 'Cavs' (27 points) have been successfully filtered out.

This exclusion confirms the successful implementation of the negative matching logic. The rows containing **Mavs** and **Cavs** in the **team** column were both removed because they contained the target substring 'avs' within their string values. The negation operator ensured that any row where the condition `df.team.contains('avs')` evaluated to `True` was converted to `False` for the filter, thus excluding those records.

It is important to appreciate the power of substring matching here. Unlike filtering for exact team names, the `.contains()` function allows for flexible pattern exclusion. This is highly useful in scenarios involving messy data, partial identifiers, or standardized suffixes/prefixes that need to be

universally ignored during an analysis. The remaining rows--Nets, Kings, Spurs, and Lakers--all successfully passed the negated condition and are now available in the refined dataset.

Addressing Case Sensitivity in String Operations

A critical operational detail when using the `.contains()` function in PySpark is its inherent case sensitivity. The matching process is highly literal, meaning it differentiates between uppercase and lowercase characters. If the provided substring pattern does not exactly match the case of the characters within the column data, the function will evaluate to `False`, even if the sequence of letters is otherwise identical.

For instance, if we had attempted to filter out 'Mavs' and 'Cavs' using the pattern "AVS" (all caps), the `.contains('AVS')` function would have returned `False` for those teams, as their names contain 'avs' (lowercase). Consequently, applying the negation operator `~` would result in `True`, meaning the rows would mistakenly be kept in the filtered result.

To achieve case-insensitive filtering when using `.contains()`, one must preprocess the column data before applying the filter. This involves converting the target column to a consistent case (usually lowercase) using functions like `pyspark.sql.functions.lower()`. This ensures that the substring match is performed consistently regardless of the original casing in the data.

Case-Sensitive Example (as demonstrated): Filtering for 'avs' correctly excludes 'Mavs' and 'Cavs'.

Hypothetical Case-Insensitive Approach: To exclude teams regardless of case (e.g., 'MAVS', 'Mavs', 'mavs'), the filter condition should be adjusted to: `df.filter(~lower(df.team).contains('avs'))`, assuming 'avs' is the search term in lowercase.

Alternative Methods for Exclusion Filtering

While `~df.col.contains()` is the clearest method for simple substring exclusion, PySpark offers other powerful functions suitable for more complex negative filtering patterns, particularly when dealing with complex rules or needing case insensitivity inherently.

One such alternative is the `.rlike()` function (regular expression match). Regular expressions provide highly flexible pattern matching capabilities, including built-in options for case insensitivity (using flags like `(?i)`). For instance, to exclude team names containing 'avs' regardless of case, one could use: `df.filter(~df.team.rlike('(?i)avs'))`. This single function replaces the need for separate case conversion functions when complex exclusion rules are required.

Another option is the `.like()` function, which utilizes SQL standard wildcards (`%`). While `.like()` is primarily used for pattern matching involving the beginning or end of a string (e.g., `%substring%`), it does not handle complex negative logic as cleanly as `.contains()` with the tilde operator, nor does it inherently offer regex flexibility. For general-purpose, simple substring exclusion, the combination of `~` and `.contains()` remains the recommended and most readable approach in PySpark.

Summary of Best Practices for PySpark Filtering

Efficient and accurate filtering is central to data preparation in DataFrame manipulation. When applying exclusionary filters using the "Not Contains" logic, adhering to best practices ensures optimal performance and code readability.

Always prioritize the use of the `~df.col.contains('substring')` syntax for straightforward negative matching, as it is highly expressive and easily understood by other developers. If case sensitivity is a concern, ensure that you explicitly address it, either by converting the column to a uniform case using `lower()` or by employing the more sophisticated `.rlike()` function with appropriate regex flags. Never rely on implicit case matching.

Finally, always verify the results of complex filtering operations using an action like `.show()` or `.collect()` on a small sample of the data. This validation step is essential to confirm that the expected rows were excluded (or included) and that no unexpected data loss occurred due to misapplied boolean logic. Mastery of the negation operator (`~`) is a powerful tool for cleaning and subsetting large-scale datasets in PySpark.