Learning PySpark: Filtering DataFrames with the NOT LIKE Operator

Authored by

Mohammed looti

November 11, 2025

Introduction to Filtering and String Operations in PySpark

When working with large datasets, the ability to efficiently filter data based on specific criteria is paramount. In the realm of big data processing using PySpark DataFrames, string manipulation and conditional filtering are fundamental tasks. While filtering for exact matches or numerical ranges is straightforward, filtering rows based on whether a column **does not match** a specified text pattern requires a combination of SQL-like logic and Python's native operators. This technique is often referred to as the **NOT LIKE** operation, derived directly from standard SQL syntax.

The challenge in PySpark lies in translating this declarative SQL operation into the functional API used by the DataFrame structure. PySpark provides the built-in `.like()` function, which is analogous to the SQL `LIKE` clause. However, to achieve the desired negation--the **NOT LIKE** functionality--we must combine this function with the logical negation operator available in Python. Understanding this combination is essential for data scientists and engineers who need fine-grained control over which rows are included in their final analysis.

This article provides a detailed guide on implementing the **NOT LIKE** filter within a PySpark DataFrame. We will examine the precise syntax required, deconstruct the role of the negation symbol, and walk through a practical example using basketball team data to demonstrate how to exclude rows that contain specific substring patterns.

Understanding the PySpark NOT LIKE Syntax

In PySpark, filtering operations are primarily handled by the `.filter()` method, which accepts a Boolean expression. To implement string pattern matching, we utilize the column method `.like()`. This method checks if a string value matches a given SQL-style pattern, where the percent symbol (%) acts as a wildcard representing zero or more characters.

To negate the result of the `.like()` function--that is, to find rows where the pattern **is not** present--we employ the tilde symbol (~). The tilde symbol acts as the logical **NOT** operator when applied to column expressions in PySpark, effectively flipping the Boolean outcome of the `.like()` condition. If `df.team.like('%avs%')` returns `True` for a row, then `~df.team.like('%avs%')` will return `False`, ensuring that row is excluded from the filtered result.

The generic syntax to filter a PySpark DataFrame using the **NOT LIKE** operator is shown below. This structure is highly efficient and idiomatic within the PySpark environment, offering a direct translation of the exclusion logic needed for precise data subsetting.

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

This specific code snippet instructs PySpark to analyze the `team` column within the DataFrame `df`.

It first determines which rows contain the substring "avs" anywhere within the string (due to the surrounding `%` wildcards). The crucial step is the application of the tilde (~), which negates this selection, resulting in a DataFrame that only contains rows where the string in the **team** column does not contain the specified pattern.

Setting Up the Sample PySpark Environment

To demonstrate the application of **NOT LIKE** filtering, we must first establish a working SparkSession and generate a sample DataFrame. This DataFrame, representing scores from various fictional basketball teams, will serve as our foundation for the filtering exercise. The dataset is intentionally structured to include teams that both match and do not match the target pattern ("avs") so we can clearly observe the effects of the exclusion logic.

We begin by importing the necessary libraries and initializing the SparkSession. Following this, we define a list of data rows, where each row contains a team name and an associated point total. Finally, we assign appropriate column names and create the DataFrame using the `createDataFrame` method. This setup ensures that our environment is ready for testing filtering logic against a realistic, albeit small, structured dataset.

The complete setup sequence, including the definition of the data structure and the subsequent display of the initial DataFrame, is provided below. This initial view helps us establish a baseline against which we can compare the results of our filtered output, ensuring the **NOT LIKE** operation behaves as expected.

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
+-------+------+

Implementing the NOT LIKE Filter in Practice

With our DataFrame successfully initialized, we can now execute the filter operation designed to exclude specific patterns. Our objective is to generate a new DataFrame subset that contains only the rows where the **team** column does not contain the substring "avs". This means we aim to eliminate entries such as "Mavs" and "Cavs," as both contain the target pattern.

The implementation uses the `.filter()` method combined with the negated `.like()` expression, `~df.team.like('%avs%')`. The inclusion of the wildcard characters (`%`) before and after "avs" ensures that the pattern match is position-independent; it successfully identifies "Mavs" (where "avs" is in the middle) and "Cavs" (where "avs" is at the end). The subsequent negation then converts this identification into an exclusion criterion.

Executing the code below produces the filtered result, clearly demonstrating how the **NOT LIKE** logic successfully removes the undesired rows. The resulting DataFrame should only include teams like "Nets," "Lakers," "Wizards," and "Spurs."

#filter DataFrame where team column does not contain pattern like 'avs'
df.filter(~df.team.like('%avs%')).show()

```
+-------+------+
| team|points|
+-------+------+
| Nets| 33|
| Lakers| 12|
|Wizards| 24|
| Nets| 40|
| Spurs| 13|
+-------+------+
```

Upon reviewing the filtered output, we can confirm that every row retained in the resulting DataFrame completely lacks the "avs" pattern in the **team** column. This outcome validates the successful application of the PySpark **NOT LIKE** equivalent. The power of this approach lies in its succinctness; it allows complex pattern matching exclusion rules to be defined and executed using a single, clear line of code within the DataFrame API.

Deconstructing the Negation Operator (~) and Pattern Syntax

A deeper appreciation of the **NOT LIKE** functionality requires understanding the two critical components at play: the `.like()` function itself and the role of the tilde operator (~). The like function is the PySpark interface for SQL-style pattern matching, utilizing standard SQL wildcards. The primary wildcard is the percent sign (`%`), which matches any sequence of zero or more characters. If we had used `'avs%'`, it would only match strings starting with "avs"; using `'%avs'` would match strings ending with "avs." By enclosing the pattern (`avs`) in wildcards (`'%avs%'`), we search for the pattern anywhere within the string.

The second, and perhaps more unique, element is the use of the tilde (~). In standard Python, the tilde often functions as the bitwise NOT operator. However, within the context of PySpark column expressions, it is overloaded to serve as the logical **NOT** operator. When prepended to a Boolean column expression (like the output of `df.team.like(...)`), it negates the truth value. This mechanism is necessary because PySpark column objects do not support Python's standard logical negation operator (`not`) directly. If the `.like()` function evaluates to `True`, the `~` operator transforms it to `False`, ensuring the row fails the filter condition and is excluded.

It is important to note the distinction between using `.like()` for string operations and the standard Python comparison operators. When filtering DataFrames, all conditions must operate on column objects, leading to the necessary use of specialized PySpark operators like `~` for negation. This adherence to PySpark's column-based operations maintains the optimization and lazy evaluation capabilities inherent to the Apache Spark framework.

Alternatives to NOT LIKE: Using rlike and contains

While the `~df.col.like()` method is the most direct translation of SQL's **NOT LIKE**, PySpark offers other powerful string manipulation functions that can achieve similar results, particularly when the complexity of the pattern matching increases. Two notable alternatives are `.rlike()` and `.contains()`.

The `.rlike()` function allows for full Regular Expression (regex) support, providing significantly more flexibility than the basic SQL wildcards used by `.like()`. To achieve a **NOT LIKE** equivalent using `.rlike()`, one would similarly negate the expression: `~df.team.rlike('avs')`. The key difference is that `.rlike()` inherently searches for the pattern anywhere in the string (similar to using `%pattern%` in `.like()`), but it follows strict regex syntax rules rather than SQL wildcard rules. If your exclusion criteria involve complex character sets, repetition counts, or boundaries, `.rlike()` is the superior choice.

A simpler alternative for checking the absence of a fixed substring is `.contains()`. This function checks if a column contains a literal substring, without using wildcards or regex. While less flexible than `.like()` or `.rlike()`, it is often more readable for simple inclusion/exclusion tasks. The **NOT CONTAINS** equivalent would be written as `~df.team.contains('avs')`. Choosing between these methods often depends on the complexity of the pattern: use `.contains()` for fixed substrings, `.like()` for SQL wildcard patterns, and `.rlike()` for advanced regex requirements.

Summary and Best Practices for Data Filtering

Mastering filtering operations is crucial for efficient data preparation and analysis within PySpark. The **NOT LIKE** operation, achieved through the negation of the `.like()` column function, provides a reliable and readable way to exclude rows based on string patterns. This method ensures that your data cleansing and preparation stages are precise, allowing subsequent analysis to focus only on relevant records.

When implementing filtering logic, adherence to best practices will maximize performance. Always strive to use native PySpark column functions (like `.filter()`, `.like()`, and `~`) rather than converting the DataFrame to Pandas or using User Defined Functions (UDFs). Native functions are highly optimized and executed on the distributed cluster, ensuring scalability for truly big data workloads. Furthermore, be deliberate in your choice of wildcards (%) or, if necessary, transition to the more powerful `.rlike()` for regular expression-based exclusions, balancing complexity with performance needs.

For those seeking to expand their knowledge of PySpark's rich functional library, exploring the complete documentation for column operations is highly recommended. The official PySpark documentation provides exhaustive details on all available functions, including advanced pattern

matching and logical operators.

Additional Resources

The following resources offer further guidance on related PySpark tasks, helping users build a comprehensive skill set for data manipulation and transformation:

PySpark Documentation on the like function.

Tutorials explaining how to perform other common filtering tasks in PySpark.