How to Filter Rows in PySpark That Do Not Match a Pattern Using NOT LIKE

Authored by

stats writer

February 6, 2026

The use of the SQL **NOT LIKE** operator within PySpark is a fundamental technique for manipulating and refining large datasets. This powerful operator allows developers and data analysts to precisely filter rows within a DataFrame based on the absence of a specific character sequence or **pattern matching** string. Utilizing **NOT LIKE** effectively enables the exclusion of data records that contain unwanted or irrelevant substrings, thereby providing a cleaner, more focused view of the underlying information. This approach is superior when seeking to isolate records that strictly do not conform to a given criterion, moving beyond simple equality checks to complex textual exclusions.

To successfully implement the **NOT LIKE** functionality in PySpark, one must leverage the existing functionality of the native `.like()` function combined with the logical negation operator. While standard SQL environments offer **NOT LIKE** as a single keyword, PySpark's DataFrame API requires a combination of methods to achieve the same result. Specifically, you define the column and the pattern to be matched using `.like()`, and then prepend the entire expression with the negation symbol (~). This combination ensures that only rows failing the similarity match are returned, proving invaluable for tasks requiring rigorous data exclusion during data analysis and preprocessing steps.

This guide delves into the precise syntax and provides a practical, step-by-step example demonstrating how to implement **NOT LIKE** filtering within a PySpark environment, ensuring you can efficiently exclude records and streamline your data processing workflows involving distributed data structure management.

PySpark: Filter Rows Using NOT LIKE

Understanding the PySpark NOT LIKE Mechanism

In the context of the PySpark DataFrame API, there is no standalone `.notLike()` method analogous to its SQL counterpart. Instead, engineers must employ the standard `.like()` function--which performs the positive pattern match--and subsequently negate the boolean result of that operation. This reliance on logical inversion is a common pattern in DataFrame manipulation across various programming languages and frameworks, prioritizing composability over verbose, specialized functions, and maintaining compatibility with Python's native operator conventions.

The standard syntax for positive pattern matching involves calling `df.column_name.like('pattern')`, where the pattern typically utilizes standard SQL wildcards. These include the percentage symbol (`%`), which represents zero or more characters, and the underscore (`_`), which represents a single character placeholder. To achieve the **NOT LIKE** functionality, we introduce the tilde symbol (~) immediately before the column expression. This tilde acts as the logical NOT operator in PySpark, converting rows that evaluate to `True` (meaning they

match the pattern) into `False`, and converting non-matching rows into `True`.

The resultant expression, `~(df.column_name.like('pattern'))`, is then seamlessly passed directly into the `.filter()` transformation method. This transformation efficiently scans the specified column, applies the pattern check, negates the outcome, and retains only those rows where the combined expression evaluates to `True`. Mastering this syntax is crucial for advanced filtering tasks where exclusion criteria are paramount for data preparation and analysis.

Core Syntax for NOT LIKE Filtering

To filter a DataFrame in PySpark to exclude rows based on a specified string pattern, the general syntax requires chaining the `.filter()` method with the negated `.like()` expression. It is essential to ensure that the negation operator (~) is correctly positioned to negate the entire pattern matching result, guaranteeing that the exclusion logic is applied consistently across all partitions of the distributed dataset.

The following is the canonical syntax used for filtering rows that do **not** contain a specific substring, demonstrated here using the hypothetical column named **team** in a DataFrame named `df`:

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

This powerful, single-line instruction directs PySpark to inspect the string values within the **team** column. The pattern `%avs%` signifies a match for any string containing the literal substring "avs" anywhere within it. By prefixing this condition with the tilde (~), we reverse the outcome, instructing the system to retain only those rows where the **team** string strictly does **not** contain "avs". This combination effectively reproduces the functionality of the **NOT LIKE** operator found in standardized SQL query languages.

Practical Example: Setting Up the PySpark Environment

To provide a concrete illustration of this filtering technique, we will first establish a small, reproducible DataFrame containing sample sports data. This setup involves the essential steps of initializing a SparkSession, defining the input dataset, and specifying the appropriate column names. This ensures a transparent environment for demonstrating the **NOT LIKE** operation and clearly observing the resulting data transformation.

The sample dataset we use contains information about points scored by various basketball teams. It is intentionally designed to include team names (such as 'Mavs' and 'Cavs') that contain the target pattern ('avs'), as well as names that do not (e.g., 'Nets', 'Lakers'). Our core objective is to execute a filter that removes all records associated with teams whose names contain the specific

pattern 'avs'.

The necessary code to initialize the SparkSession and construct the sample DataFrame is provided below, allowing you to replicate the environment:

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

# Define input data containing team names and scores
data = ,
,
,
,
,
,
,
,
]

Define column headers
columns =

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

# View the initial DataFrame structure and content
df.show()

+-------+------+
| team|points|
+-------+------+
Mavs	18
Nets	33
Lakers	12
Mavs	15
Cavs	19
Wizards	24
Cavs	28
Nets	40
Mavs	24

| Spurs| 13|
+-------+------+

Executing the NOT LIKE Filter Operation

With the DataFrame successfully initialized, the next step is applying the exclusion filter. We aim to retain only those rows where the **team** column does not contain the pattern `%avs%`. This operation is expected to eliminate all records for 'Mavs' and 'Cavs', leaving only the rows for 'Nets', 'Lakers', 'Wizards', and 'Spurs'.

This filtering process showcases the efficiency of PySpark transformations, as the operation is executed lazily and distributed across the cluster, guaranteeing high performance even when processing truly massive datasets. Developers must remember that the `.like()` comparison is inherently case-sensitive by default. If case-insensitive filtering is required, the column content should be converted to a consistent case (e.g., lowercase) using functions like `pyspark.sql.functions.lower()` before applying the `.like()` comparison.

We apply the filter operation using the precise combination of the negation operator and the `.like()` function:

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

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

Analyzing the Filtered Results

The output displays five rows, which represent the data that successfully passed the negative pattern match. We can clearly observe that every row associated with the 'Mavs' team and the 'Cavs' team has been rigorously excluded, confirming the successful implementation of the desired exclusion logic. The resulting teams are 'Nets', 'Lakers', 'Wizards', and 'Spurs'.

This outcome confirms that the combination of the `.like()` function for positive pattern identification and the tilde (~) for logical negation accurately achieved the precise data exclusion

required. This technique is indispensable for critical tasks such as data cleansing, creating segmented groups, or performing compliance checks where specific categories must be entirely removed based on complex character pattern matching rules. The reliability and distributive nature of this method make it a cornerstone of modern data analysis workflows.

Deep Dive into Negation and Pattern Matching

It is beneficial to formalize the understanding of the roles played by the two primary components used in this filtering expression. The `.like()` function is an essential method available on all PySpark Column objects. It is designed specifically to handle pattern matching using standard SQL wildcards, returning a boolean Column that indicates, row by row, whether the string matches the provided pattern.

The tilde (`~`) symbol serves specifically as the unary logical NOT operator within the PySpark expression context. When applied to the boolean output of `df.team.like('%avs%')`, it effectively reverses the truth value: any row where the match was `True` (meaning the team name contained 'avs') now becomes `False`, and any row where the match was `False` (meaning the team name did not contain 'avs') now becomes `True`. Since the `.filter()` transformation retains only rows where the condition evaluates to `True`, the resulting DataFrame contains only the desired non-matching records.

This explicit separation of pattern definition and negation provides developers with significant flexibility. For instance, if the requirement evolves to combine **NOT LIKE** with additional boolean conditions--such as filtering out teams that don't match 'avs' AND scored above 20 points--the expression can be easily nested or combined using standard PySpark logical operators (`&` for AND, `|` for OR).

Alternative Filtering Methods: SQL Expressions

While the DataFrame API method using `~df.col.like()` is considered the most idiomatic PySpark approach, filtering can also be achieved efficiently by leveraging raw SQL expressions through the `.filter()` or `.where()` methods. If a user possesses a strong background in traditional database systems and prefers pure SQL syntax, they can pass the exact **NOT LIKE** condition as a string argument to the filter function.

This SQL-based approach requires encapsulating the entire condition as a string. For example, the equivalent operation demonstrated previously could be written concisely as: `df.filter("team NOT LIKE '%avs%'").show()`. This method often enhances code readability for individuals or teams accustomed to complex, multi-condition filters written in SQL.

However, it is generally recommended to utilize the DataFrame API expression

(`~df.team.like('%avs%')`) as it typically offers better integration with Spark's Catalyst Optimizer, allowing for optimal query planning and serialization by relying on native Column objects. This adherence to the DataFrame API aligns more closely with the functional programming style encouraged by PySpark. Both methods produce identical results, offering developers a choice based on coding standards and specific performance considerations.

Summary and Further Resources

The ability to use the **NOT LIKE** logic is an indispensable skill for advanced data manipulation tasks in PySpark. By mastering the technique of combining the `.like()` function with the logical negation operator (~), users gain precise and fine-grained control over which records are excluded from their analysis based on complex character pattern matching rules.

Remember that the core mechanism relies on three distinct operational parts: the `.filter()` transformation, the column access (e.g., `df.column_name`), and the mandatory logical negation (`~`) applied directly to the `.like()` function call. This combination ensures robust and accurate data exclusion across vast, distributed systems.

For developers seeking more extensive detail on the functions used, including specifics on pattern matching syntax, wildcard escaping, and character set handling, the official Apache Spark documentation serves as the ultimate authoritative reference.

Note: You can find the complete documentation for the PySpark **like** function in the official Apache Spark documentation.

Additional PySpark Tutorials

The following tutorials explain how to perform other common data transformation and manipulation tasks in PySpark, helping you expand your proficiency in distributed data analysis:

How to handle null or missing values effectively in DataFrames.
Implementing various types of joins between two distributed DataFrames.
Using UDFs (User-Defined Functions) for applying custom, non-standard Python logic.
Mastering Window functions for advanced aggregations and analytical operations.