How to Filter Rows in PySpark Using NOT LIKE

Authored by

stats writer

January 20, 2026

Understanding PySpark Filtering and String Matching

PySpark is the Python API for PySpark, a powerful tool designed for large-scale data processing and analysis. When working with DataFrames, a common requirement is filtering rows based on complex string patterns rather than requiring exact matches. This necessity leverages functionality derived directly from the standard Structured Query Language (SQL), specifically the concept of the `LIKE` operator. To implement the critical inverse operation--filtering data that **does not match** a specific pattern, known as `NOT LIKE`--we combine PySpark's built-in `like` function with the bitwise NOT operator, represented by the tilde symbol (~). This combination allows data professionals to precisely exclude data based on specific substring occurrences, greatly enhancing data quality and relevance for subsequent analyses.

The direct implementation of the `NOT LIKE` concept in PySpark requires a deep understanding of how logical negation interacts with column expressions. Unlike standard SQL where `NOT LIKE` is a single keyword phrase, PySpark mandates the use of the `~` operator applied to the entire column expression generated by the `.like()` method. This distinction is vital for ensuring the code executes correctly and efficiently across a distributed cluster environment. The tilde operator acts on the resulting Boolean column, inverting the truth value of every row; if a row was a match (True), it becomes a non-match (False) and is filtered out, and vice versa.

The fundamental structure for achieving `NOT LIKE` filtering involves calling the **`.filter()`** transformation on the DataFrame, passing it a complete Boolean expression. This expression is constructed by targeting a column (e.g., `df.column_name`), applying the like function with the desired pattern, and then prefixing the entire resulting expression with the `~` operator. For instance, if you want to exclude all team names containing the substring "avs", you would use the pattern `'%avs%'`. The surrounding percent signs are crucial wildcards, signifying that the substring "avs" can appear anywhere within the string, and the negation ensures these matching rows are excluded.

The PySpark NOT LIKE Operator Syntax

The standard syntax to implement the `NOT LIKE` functionality within a PySpark operation is powerful due to its conciseness. It centers around the **`.filter()`** method, a core transformation used to subset rows based on specified conditions. When dealing with string patterns, the column object is chained with the **`.like()`** method, which accepts a string pattern defined using wildcards. To negate this condition, the tilde (~) operator must be applied directly to the column expression, signaling that only rows where the pattern match fails should be retained in the resulting DataFrame.

The following code snippet demonstrates the core syntax for filtering a PySpark DataFrame using

the logical NOT LIKE operation. This specific pattern is designed to exclude rows where the string in the **team** column contains the pattern "avs" anywhere within the text. This effectively achieves the same goal as the `NOT LIKE` keyword found in standard SQL implementations.

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

In this structure, the expression `df.team.like('%avs%')` first generates a column of Boolean values where True indicates a match (i.e., the string contains "avs"). By prepending the tilde symbol (`~`), we logically negate all those Boolean values. Consequently, the filter only retains rows where the original expression was False (meaning the team name did not contain "avs"). Understanding the role of the `~` operator is paramount; it acts as the logical negation for the column expression, making the `NOT LIKE` operation possible in PySpark's functional programming style. The use of the percent sign (wildcard) on both sides of "avs" ensures that the pattern matching is performed across the entire string.

Prerequisites: Setting up the PySpark Environment and DataFrame

To illustrate the practical application of the `NOT LIKE` filtering technique, we must first establish a sample PySpark environment and construct a simple source DataFrame. This DataFrame, which simulates data collected about basketball teams and their points scored, will serve as a concrete basis for our pattern-matching exercise. It is essential to visualize the initial data structure before applying any filtering transformations, as this context is necessary for verifying the accuracy of the output.

We begin by importing the necessary libraries, primarily `SparkSession`, which is the gateway to using Spark functionality within Python. After initializing the Spark session, we define our sample data. This dataset includes various team names, some of which contain the specific pattern we intend to exclude (e.g., 'Mavs' and 'Cavs', which both contain the substring 'avs'), and others which do not (e.g., 'Nets', 'Lakers', 'Spurs'). Defining the column names clearly--**team** and **points**--ensures that the resulting structure is intuitive and easily manageable for subsequent operations.

The code block below demonstrates the complete setup required: from creating the Spark session and defining the raw data structure to naming the columns and finally, creating and displaying the initial DataFrame. Note the multiple entries for teams containing 'avs'; these are precisely the rows that the `NOT LIKE` filter will target and remove from the final output, allowing us to confirm the robustness of our filtering logic against duplicate entries.

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 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|
+-------+------+
```

Step-by-Step Implementation: Filtering Data Using NOT LIKE

With the initial DataFrame successfully created and displayed, the next step involves applying the `NOT LIKE` logic to filter out the undesired rows based on the pattern "avs". Our specific requirement is to retain only those rows where the value in the **team** column does not contain the

substring "avs". This is accomplished by applying the logical negation operator (`~`) to the expression generated by the like function, effectively transforming a 'match' condition into a 'non-match' condition that the filter then accepts.

The following code snippet executes the filtering transformation using the established PySpark syntax. It demonstrates the precise implementation of the pattern match `'%avs%'` combined with the negation operator `~`. The resulting DataFrame will only contain teams like 'Nets', 'Lakers', 'Wizards', and 'Spurs', while explicitly excluding all instances of 'Mavs' and 'Cavs'. This operation is highly optimized, leveraging the performance capabilities of the Spark engine to handle the transformation across potentially massive datasets with high efficiency.

 # 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|
+-------+------+

Reviewing the final output confirms that the filter functioned exactly as intended. Every row in the resulting DataFrame displays a team name that **does not** contain the substring "avs". We successfully removed all 5 rows corresponding to 'Mavs' and 'Cavs', leaving 5 rows that meet the exclusion criteria. This result decisively highlights the power and flexibility of combining the like function for pattern recognition with the `~` symbol for logical negation within the PySpark environment. This method provides an elegant and declarative solution for handling data exclusion requirements, which are often encountered during data cleaning and preprocessing phases.

Deep Dive into Pattern Matching: Understanding Wildcards

When utilizing the like function in PySpark, understanding the precise role of wildcards is fundamental to defining effective patterns. Wildcards allow us to specify variable parts of a string pattern, providing essential flexibility that exact string matching lacks. PySpark, inheriting its functionality from standard SQL, supports two primary wildcards: the percent sign (%) and the underscore (_). Misunderstanding or misusing these characters can easily lead to filtering outcomes that are either too broad or too restrictive, failing to meet the original analytical objective.

The **percent sign (%)** represents zero, one, or multiple characters. In our previous example, `'%avs%'` means any string that has "avs" located somewhere inside it. The `%` placed before 'avs' accounts for any arbitrary number of characters preceding it, and the `%` placed after 'avs' accounts for any characters following it. This symmetrical placement is what makes the match location-agnostic. If we had used only `'avs%'`, the filter would only match strings that **begin** with "avs". Conversely, using `'%avs'` would only match strings that **end** with "avs". The extensive flexibility offered by the `%` wildcard is what enables powerful string subset filtering for inclusion or exclusion purposes.

The **underscore (_)** represents a single, required character. If, for instance, we used the pattern `'C_vs'`, it would match 'Cavs' and 'Cevs', but it would specifically fail to match 'Crnvs' (too many characters) or 'Cvs' (too few characters). This wildcard is particularly useful when you need to match strings that differ only by a single variable character at a specific position, such as when dealing with variations in short identifiers or fixed-length codes. When the single character wildcard is combined with the `~` operator for negation, you gain the ability to precisely exclude entries that fit a very specific structural pattern while accepting nearby variants.

Alternative Methods for String Negation in PySpark

While the combination of `~` and the like function is the canonical way to achieve `NOT LIKE`, PySpark provides alternative methods for highly advanced string filtering, most notably using regular expressions via the **`.rlike()`** function. If the filtering requirement involves overly complex patterns that cannot be adequately described using only the standard SQL wildcards (`%` and `_`), the `rlike` function becomes the superior tool. It allows developers to leverage the full expressive power of standard regular expressions, providing comprehensive granularity for pattern matching, including specifying character sets, repetition counts, and string boundaries.

The implementation structure remains parallel when using regular expressions: `df.filter(~df.column_name.rlike('regex_pattern'))`. This approach ensures that we can still utilize the logical negation (~) while benefiting from enhanced pattern capabilities. For example, if we wanted to filter out teams containing either 'avs' or 'ets', a complex regex pattern like `'(avs|ets)'` could be used within `rlike`. However, it is a crucial best practice to use the simpler `like` function if the pattern can be described efficiently using only `%` and `_`, as the native `like` operation is generally more optimized for performance within the Spark Catalyst Optimizer compared to general regular expression matching.

Another critical consideration in string filtering is **case sensitivity**. By default, PySpark's `like` function performs case-sensitive matching, mirroring standard SQL behavior unless the underlying database engine is configured otherwise. This means that `'%avs%'` will match 'Mavs' but will likely ignore entries like 'MAVS' or 'aVS'. If the analytical requirement demands a case-insensitive `NOT`

`LIKE` filter, you must first convert the target column to a consistent case (either upper or lower) before applying the pattern match. This is achieved using built-in functions such as `lower()` or `upper()` from `pyspark.sql.functions`, thereby guaranteeing that the pattern comparison is uniform across all rows irrespective of the original data casing.

Performance Considerations for String Matching in PySpark

When executing data transformations on large-scale datasets using PySpark, optimizing for performance is always paramount. The method of using **`.filter()`** combined with native column functions like `.like()` is highly efficient. Spark's execution engine, the Catalyst Optimizer, intelligently translates these high-level Python commands into streamlined physical execution plans that run across the cluster. Because `like` operations are common and map directly to optimized operations in the distributed computation graph, they benefit from specialized optimizations that minimize data shuffling and maximize parallel processing across worker nodes. This approach yields vastly superior performance compared to applying filtering logic using inefficient Python User-Defined Functions (UDFs).

Implementing filters early in the data processing pipeline is also a mandatory best practice, often referred to as predicate pushdown. By applying the `NOT LIKE` filter as soon as the relevant column data is available, we significantly reduce the volume of data that subsequent, potentially more expensive, transformations need to process. For instance, if a complex aggregation, window function, or large join operation is scheduled later in the script, running the filter beforehand ensures that these resource-intensive tasks only operate on the necessary, minimized subset of the DataFrame.

This systematic reduction of data volume contributes dramatically to overall job completion time and resource utilization efficiency. In summary, choosing the native PySpark column API--specifically the combination of `~` and `.like()`--for `NOT LIKE` operations guarantees that the transformation is handled natively and efficiently by the underlying Spark engine. This high-performance strategy avoids the pitfalls associated with row-by-row Python processing and allows the query optimizer to effectively distribute the workload, which is essential for maintaining scalability when dealing with big data volumes in a production environment.

Conclusion and Further Resources

Implementing the `NOT LIKE` operation in PySpark is a fundamental skill for data professionals who require precise control over string pattern exclusions. By leveraging the native column functions and the logical negation operator `~`, we can efficiently filter DataFrames to exclude rows matching specific substrings defined by wildcards. This powerful method not only aligns closely with traditional SQL filtering concepts but also ensures optimal performance within the demanding,

distributed computing environment of Apache Spark.

The key takeaway is the recognition that the `~` symbol must precede the entire column expression involving the like function. This specific syntax structure, `df.filter(~df.column.like('pattern'))`, allows for clear, readable, and highly efficient code execution. Whether the task involves excluding specific identifiers, filtering massive log messages, or cleaning textual data based on prohibited substrings, mastering this essential syntax unlocks significant power for advanced data manipulation in PySpark.

The following tutorials explain how to perform other common and related tasks in PySpark, building upon the foundational knowledge of filtering and pattern matching:

How to filter a DataFrame using complex conditional logic.
Utilizing regular expressions (`rlike`) for advanced pattern exclusion.
Optimizing PySpark job performance by implementing predicate pushdown.