PySpark: Filter Using Contains”

Authored by

stats writer

November 16, 2025

When working with large-scale datasets in PySpark, precise data manipulation is essential for deriving meaningful insights. One of the most common requirements is filtering a DataFrame based on specific string patterns within a column. The built-in `contains` operator provides a straightforward and highly efficient method for achieving this. This function, which is part of the Column API, allows developers to easily check if a target string exists as a substring within the values of a specified column. It is an indispensable tool for operations like isolating records based on partial identifiers, team names, or product codes. Understanding its usage, limitations, and alternatives is key to mastering data wrangling in the Spark ecosystem.

Mastering the `contains` Operator Syntax

The `contains` method is syntactically simple, integrating seamlessly with the primary DataFrame.filter() transformation. The PySpark framework uses the `filter()` method to select rows based on a conditional expression applied across one or more columns. When employing string matching, the condition is created by selecting the target column, applying the `contains` function, and passing the desired substring as an argument. This structure ensures that only rows satisfying the precise containment condition are retained in the resulting DataFrame.

The general approach involves calling the `filter` method on the existing DataFrame object, passing a boolean expression where the column reference is chained with the `contains` method. For instance, if you have a DataFrame named `df` and wish to filter the `team` column for the substring 'avs', the syntax is concise and highly readable, clearly demonstrating the intent of the data operation. This operator significantly simplifies operations that might require complex regular expressions or user-defined functions (UDFs) in other database environments, keeping the code native to the Spark SQL context for better optimization.

The following standard syntax demonstrates how to apply this filter to select rows in the `team` column containing the specified substring. This operation returns a new DataFrame containing only the matches, without modifying the original data structure, adhering to Spark's immutable data model.

#filter DataFrame where team column contains 'avs'
df.filter(df.team.contains('avs')).show()

Detailed Example: Filtering Team Data in PySpark

To properly illustrate the functionality of the `contains` operator, we must first establish a representative dataset. In this example, we will simulate a scenario common in sports analytics, where we track points scored by various basketball teams. This involves defining the schema, creating the data rows, and initializing the DataFrame using a SparkSession instance. The

resulting DataFrame provides a clear, verifiable structure against which we can test our filtering logic, ensuring the output is predictable and aligns with the expected behavior of the `contains` function.

The process begins by importing the necessary components from `pyspark.sql` and initializing the SparkSession--the entry point to all Spark functionality. We then define the sample data, which includes various team abbreviations and their corresponding point totals. For clarity, we define the column names separately before using the `spark.createDataFrame` method. This setup is standard practice in PySpark development and ensures reproducibility across different environments. The initial data display confirms that the DataFrame is correctly structured and ready for transformation operations.

The following code block demonstrates the complete setup required to create our sample dataset, which we will subsequently use to apply the string filtering techniques discussed in this guide.

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

Executing the Filter and Analyzing the Results

Once the initial DataFrame is established, we can proceed with applying the `contains` filter. Our objective is to isolate only those rows where the `team` column contains the sequence of characters "avs." This demonstrates a practical application where a dataset might contain variations of abbreviations or specific identifiers, and we need to group them based on a common partial string match. The `contains` function evaluates each entry in the specified column against the provided substring, returning `True` only if the substring is present.

When applying `df.filter(df.team.contains('avs'))`, PySpark processes the DataFrame efficiently, utilizing its optimized engine to perform the string comparison across all partitions. In our sample data, the teams "Mavs" and "Cavs" both satisfy this condition. Crucially, teams like "Nets," "Kings," "Spurs," and "Lakers" are excluded because they do not contain the exact substring 'avs' within their names. The resulting DataFrame is significantly smaller, comprising only the relevant rows, which is the core purpose of a filtering operation.

Executing the filtering operation and displaying the results confirms the successful application of the `contains` logic. Observe the output closely: only the entries associated with "Mavs" and "Cavs" remain, demonstrating that the operation successfully identified and isolated all records containing "avs" in the team identifier.

#filter DataFrame where team column contains 'avs'
df.filter(df.team.contains('avs')).show()

+----+------+
|team|points|
+----+------+
|Mavs| 14|
|Cavs| 27|
+----+------+

Understanding Case Sensitivity and contains

A fundamental aspect of the `contains` function in PySpark that users must grasp is its adherence to case sensitivity. By default, the `contains` method performs an exact character match, meaning that the case of the characters provided in the search string must match the case of the characters in the column data for a match to occur. This behavior is standard across many database and data processing systems and is critical for ensuring data integrity where case differences represent distinct values (e.g., 'ID-A' versus 'id-a').

For example, if we attempted to filter our dataset using the search term "AVS" (all uppercase) instead of "avs" (all lowercase), the filter would return an empty DataFrame. This is because neither "Mavs" nor "Cavs" contains the exact substring "AVS" in uppercase. This strict matching mechanism highlights why careful consideration of data input format and cleanliness is required before applying string filters. If the goal is to perform a case-insensitive search, the filtering logic must be modified.

To achieve a case-insensitive containment check, the recommended practice involves normalizing both the column data and the search term to a uniform case (either uppercase or lowercase) before applying the `contains` operation. This is done using the built-in PySpark SQL functions `lower` or `upper`.

The procedure for a case-insensitive match involves these steps:

Import the necessary functions, specifically `lower`, from `pyspark.sql.functions`.

Apply the `lower` function to the target column (e.g., `df.team`).

Convert the search string (e.g., "AVS") to the same case (e.g., "avs").

Apply the `contains` method to the resulting lowercase column expression.

This modified syntax ensures that the filtering is robust regardless of the initial casing inconsistencies in the source data. For example: **`df.filter(lower(df.team).contains('avs')).show()`** would successfully capture "Mavs," "cavs," and "MAVS," demonstrating a flexible approach to string matching in PySpark.

Handling Null Values and Edge Cases

When dealing with real-world datasets, the presence of null or missing values is inevitable, and it is crucial to understand how the `contains` operator interacts with them. In PySpark, if a cell in the column being filtered contains a null value, applying any string operation, including `contains`, will result in a null output for that specific condition evaluation. Since the `filter` transformation requires

a boolean condition (True or False), null values are typically treated as neither True nor False, leading to the exclusion of those rows from the resulting DataFrame.

If your intention is to explicitly include or exclude rows containing nulls, you must handle these records prior to or alongside the `contains` operation using explicit null checks. For instance, to filter for rows containing 'avs' while ensuring that records with null team names are also excluded (which is the default behavior), no special handling is needed. However, if you wanted to include rows where the team name is null *or* it contains 'avs', you would need to combine these conditions using the OR operator (`|`) and the `isNull()` or `isNotNull()` methods provided by the Column API.

Another important edge case is filtering for an empty string. The `contains` function checks for the presence of the substring within the target column value. If you search for an empty string (`df.team.contains('')`), PySpark treats an empty string as being contained within any non-null string. Therefore, filtering for an empty string will return all non-null rows in that column. If you specifically need to find cells that are entirely empty strings (not nulls), you should use a direct equality comparison: **`df.filter(df.team == '').show()`**, which is distinct from the `contains` functionality.

Alternative String Matching: `like` and `rlike`

While `contains` is perfect for simple substring checks, PySpark offers more powerful alternatives for complex pattern matching: `like` and `rlike`. Understanding these alternatives helps developers choose the most appropriate tool for a given filtering task, balancing complexity and performance.

The `like` operator utilizes SQL standard wildcard characters, specifically the underscore (`_`) for matching any single character and the percent sign (`%`) for matching any sequence of zero or more characters. While `contains('avs')` looks for the substring 'avs' anywhere in the field, using `like` requires placing wildcards explicitly. To replicate the functionality of `contains('avs')` using `like`, you would write: **`df.filter(df.team.like('%avs%')).show()`**. This explicitly states that 'avs' must be surrounded by zero or more other characters, achieving the same result but requiring additional syntax. The primary benefit of `like` over `contains` is its ability to handle patterns such as "starts with" (`'avs%'`) or "ends with" (`'%avs'`).

For even more complex requirements, such as matching specific formats (e.g., alphanumeric codes, phone numbers, or dates), the `rlike` operator, which stands for "regular expression like," is the superior choice. `rlike` allows the use of full Java regular expression syntax. For instance, if you needed to find team names that start with 'M' and end with 's', you could use: **`df.filter(df.team.rlike('^M.*s$')).show()`**. This level of control is not possible with `contains` or `like`. However, regular expression matching generally incurs a higher computational overhead compared to the simpler substring comparison performed by `contains`. Therefore, `contains`

should always be the preferred method when only a straightforward substring check is necessary.

Performance Considerations for String Filtering

When architecting large-scale data processing pipelines, the efficiency of filtering operations directly impacts overall job performance. While `contains`, `like`, and `rlike` all achieve pattern matching, they differ significantly in their execution profiles within the PySpark environment.

The `contains` operation is typically the fastest among the string matching functions because it performs a basic substring search algorithm, which is highly optimized within the Spark SQL engine. Because the pattern is fixed and simple, Spark can execute this operation very efficiently across distributed partitions. The performance of `like` is generally comparable to `contains` for simple wildcard patterns, but it may involve slightly more overhead due to the required wildcard resolution.

In contrast, the `rlike` operator, relying on complex regular expressions, is significantly more computationally expensive. Regular expression processing requires state tracking and potentially complex backreferencing, which adds substantial overhead per row. Developers should reserve `rlike` only for scenarios where the required matching complexity cannot be satisfied by `contains` or `like`. For the simplest task--checking for the presence of a known substring--the concise and high-performing `contains` operator is the clear winner, ensuring the filtering step adds minimal latency to the distributed computation. By choosing the simplest necessary function, developers ensure their PySpark applications remain performant and scalable.