Learning PySpark: Filtering Data with String Contains

Authored by

Mohammed looti

November 10, 2025

Introduction to String Filtering in PySpark

When navigating and processing massive, distributed datasets within the **PySpark** environment, the ability to efficiently isolate specific data subsets is paramount. A particularly common requirement, especially when dealing with columns containing textual information, involves filtering rows based on whether a column value includes a defined substring. This operation is foundational for many data engineering tasks, including crucial activities like data cleansing, locating specialized identifiers, or segmenting records based on embedded patterns. **PySpark**, leveraging the power of the Apache Spark framework, provides highly optimized, native methods tailored for handling these string operations directly within a distributed **DataFrame**. These methods ensure that even complex filtering tasks are executed with high performance across the entire cluster, making mastery of these built-in functions essential for any professional working with large-scale data manipulation.

While traditional relational database systems rely heavily on the SQL `LIKE` operator for pattern matching, the native **PySpark** **DataFrame** API encourages the use of specialized column expressions. For the specific and frequent need of determining if a column value contains a specific sequence of characters, the `.contains()` function is the most direct, straightforward, and idiomatic choice. This function operates directly on a column object within the **DataFrame**, generating a boolean result for every row that indicates the presence or absence of the specified string fragment. This inherent simplicity makes `.contains()` invaluable for quick, precise data isolation, allowing developers to target relevant records without the added complexity associated with defining user-defined functions (UDFs) or constructing intricate SQL query strings.

This guide is designed to provide a comprehensive walkthrough of the `.contains()` function, detailing its application, syntax, and practical implications. We will move beyond theoretical concepts by demonstrating the exact code required to implement this powerful filter on a sample **DataFrame**, illustrating how to retrieve only those rows that successfully meet the substring criteria. Furthermore, we will address important operational constraints, notably the function's strict **case-sensitive** matching behavior, and introduce alternative, more flexible methods, such as utilizing regular expressions via `.rlike()`, for scenarios demanding broader pattern recognition. This holistic approach ensures a deep and complete understanding of all available string filtering capabilities within the **PySpark** framework.

Mastering the .contains() Method Syntax

The fundamental mechanism for applying string containment logic in PySpark involves pairing the `.filter()` transformation with the conditional expression generated by the `.contains()` method. The `.filter()` method serves as the primary gateway for applying row-level predicates to any DataFrame, retaining only those records where the condition evaluates to true. When

`.contains()` is employed, it dynamically checks every string value in the designated column against the provided substring. This harmonious combination offers a powerful, declarative means of expressing complex filtering logic, bypassing the need for less efficient or more verbose approaches, thereby maintaining the high-performance efficiency that is mandated by Spark's distributed architecture.

The required syntax for implementation is remarkably concise and follows the object-oriented nature of the PySpark API. To execute the filter, you invoke the `.filter()` method on your target DataFrame and pass it a single condition. This condition is formulated by first selecting the column of interest (e.g., `df` or `df.column_name`) and then immediately calling the `.contains()` method upon that column object, supplying the desired substring as the argument. The resulting expression is a boolean column that Spark's Catalyst Optimizer uses internally to determine which rows should be preserved. This highly readable pattern aligns with the functional programming principles prevalent in modern data processing libraries, ensuring that the code remains maintainable, even when combining multiple **filtering** criteria using standard logical operators like `&` (AND) or `|` (OR).

Consider a practical example: if we are working with a DataFrame named `df` and our goal is to isolate all records where the column designated as `team` includes the substring 'avs', the necessary code snippet is highly intuitive. This construction directly interacts with the column object, allowing Spark to optimize the operation effectively. It is crucial to remember that the argument passed to `.contains()` must be a standard string literal representing the exact character sequence being sought. Any variance, particularly in capitalization, will result in a mismatch, a detail that we will examine closely when discussing the inherent **case-sensitive** nature of this method. The following standard structure illustrates the implementation of this particular string filtering operation:

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

Mastering this fundamental filtering technique is foundational for executing more advanced text processing tasks efficiently within PySpark. The syntax not only provides immediate feedback, allowing developers to display the resulting subset of the DataFrame directly, but also integrates seamlessly into larger processing pipelines where subsequent transformations or actions are necessary.

Practical Implementation: Setting Up the Environment and Sample Data

Before we can demonstrate the practical utility of the `.contains()` function, we must first properly configure a working PySpark environment and create a representative sample **DataFrame**.

DataFrames in Spark represent immutable, distributed collections of data structured into named columns, analogous to tables found in relational databases. Our example utilizes a simple dataset designed to simulate score information for several fictional basketball teams. This setup begins with initializing a **SparkSession**, which acts as the unified, primary entry point for leveraging all Spark functionality across the cluster.

The data creation process is transparent and involves defining the raw data structure as a list of lists, where each inner list corresponds to a single row in the final dataset. Simultaneously, we define the explicit column schema, ensuring clarity and control over the structure of the resulting distributed collection. We will establish two columns: `team` (a string column holding the team name) and `points` (an integer column representing the accumulated score). Once both the raw data and the column names are prepared, the `spark.createDataFrame()` method is invoked to materialize the distributed DataFrame, rendering it ready for high-speed processing and analytical operations. This initial environment setup is non-negotiable, as all subsequent transformations, including our desired **filtering** action, depend entirely on this foundational distributed structure.

The following comprehensive code block illustrates every step necessary to import the required PySpark modules, initialize the **SparkSession** instance, define our sample data, and display the resulting DataFrame using the `.show()` action. Reviewing the output of `.show()` is essential, as this visual confirmation validates that the DataFrame has been correctly constructed, providing the necessary context to anticipate the exact outcomes of our upcoming containment filtering operation based on the visible team names.

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 Operation and Analyzing Results

With the sample DataFrame successfully created and visualized, the subsequent and most critical step is the application of the string containment **filtering** logic using `.contains()`. Our primary objective remains focused: to isolate only those rows where the team name explicitly includes the substring 'avs'. Based on a quick visual assessment of the sample data displayed above, we anticipate that two teams--'Mavs' and 'Cavs'--will satisfy this criterion. The `.contains()` function is specifically engineered to achieve this result, producing a new, derived DataFrame that represents a precise subset of the original data, all while maintaining the immutability characteristic that defines Spark DataFrames.

The actual execution is triggered by coupling the `df.filter()` method with the column expression `df.team.contains('avs')`, exactly as defined earlier in the syntax section. Crucially, this operation initiates the distributed execution plan across the Spark cluster. Spark employs lazy evaluation, meaning the transformation is not physically performed until an action, such as `.show()`, is called. When `.show()` is executed, Spark evaluates the boolean predicate for every single row in the `team` column. If the specific string 'avs' is discovered anywhere within the team name string--be it at the beginning, in the middle, or at the end--the row's predicate evaluates to true, and the record is subsequently included in the final result set. If the substring is entirely absent, the row is efficiently discarded from the output.

The resulting output confirms the precision and effectiveness of the containment filter. Only two records survive the transformation, corresponding perfectly to the teams we identified visually prior to execution. This confirms that the `.contains()` method successfully executed a partial string

match, underscoring its immense utility in real-world scenarios where full column value equality is unnecessary, and the mere presence of a key identifier or pattern within a larger text field is the required condition. The resulting DataFrame is targeted, accurate, and immediately prepared for any subsequent analytical processing, aggregation, or storage operations.

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

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

It is critically important to observe the precise characteristics of the resulting output set. Every row in the filtered DataFrame, specifically within the `team` column, contains the exact substring 'avs'. Conversely, all other teams--including 'Nets', 'Kings', 'Spurs', and 'Lakers'--were excluded because their respective team names did not include this specific sequence of lowercase characters. This successful **filtering** process showcases the high precision and practical utility of the `.contains()` method for rapid substring identification within structured data columns in the **PySpark** environment.

Addressing Case Sensitivity and Solutions

A critical operational constraint of the **filtering** logic implemented via the `.contains()` function in PySpark is its strict adherence to **case-sensitive** matching. This means the search string provided to the function must match the capitalization of the substring within the column value precisely. In our demonstration, searching for 'avs' (all lowercase) successfully returned 'Mavs' and 'Cavs'. However, had we searched for 'AVS' (all uppercase) or 'Avs' (title case), the outcome would be dramatically different, likely resulting in an empty DataFrame since our underlying data does not contain an exact, matching case variant of the search term.

While this strict **case-sensitive** nature is often a desirable feature for scenarios requiring high precision, such as validating identifiers, database keys, or specific product codes where case holds semantic meaning, it presents a challenge when dealing with unstructured or user-generated input. In environments prone to capitalization inconsistencies, relying solely on `.contains()` will lead to missed matches. For instance, if a developer attempts to filter the DataFrame using the expression `df.filter(df.team.contains('AVS')).show()` on our existing sample data, zero rows would be returned because neither 'Mavs' nor 'Cavs' contains the fully uppercase substring 'AVS'. This

outcome serves as a crucial reminder for all practitioners to always confirm and account for the expected case of their target strings prior to execution.

To effectively circumvent this inherent limitation and perform true **case-insensitive** containment searches, the data must first be normalized to a consistent case before the comparison operation is executed. PySpark offers powerful, built-in string functions such as `lower()` and `upper()`, which are readily available by importing `pyspark.sql.functions`. These functions can be applied directly to the column within the filter expression itself. For example, to search for 'avs' irrespective of the original capitalization, one would modify the filter predicate to: `df.filter(lower(df.team).contains('avs')).show()`. This robust approach guarantees that both the column content and the search term are standardized to a lowercase format immediately prior to the containment check, ensuring a comprehensive and accurate match across all variations of case formatting.

Utilizing Regular Expressions with .rlike() for Advanced Flexibility

While the `.contains()` method is perfectly suited for straightforward, exact substring matches, scenarios requiring complex pattern recognition or the implementation of case-insensitivity without explicit column transformation often necessitate the superior power of regular expressions. PySpark provides the `.rlike()` method, which grants users access to the full scope of standard regular expression syntax for **filtering** string columns. The `.rlike()` function accepts a regular expression pattern as its string argument and returns true if any segment of the column value successfully matches that pattern, offering significantly greater flexibility compared to the fixed string comparison utilized by `.contains()`.

For example, attempting to find teams that contain either 'avs' or 'ets' would require combining multiple `.contains()` filters using the `|` (OR) operator. Using `.rlike()`, however, a simple, concise pattern like `'(avs|ets)'` achieves this complex logical filtering in a single, expressive step. More critically, regular expressions provide native support for **case-insensitive** searching through the integration of embedded flags. By simply prefixing the pattern string with `(?i)`, the underlying regular expression engine is instructed to disregard case during the matching process, thereby simplifying the code structure considerably compared to the necessity of explicitly applying the `lower()` function.

If the objective is to perform a case-insensitive containment search for the substring 'avs', the equivalent `.rlike()` expression would be: `df.filter(df.team.rlike('(?i)avs')).show()`. This single line of code is robust enough to accurately match 'Mavs', 'mavs', 'MAVS', or any other variant of capitalization that includes the target substring. This powerful versatility establishes `.rlike()` as the preferred tool for intricate text processing tasks, validating complex input formats, or implementing sophisticated search algorithms that extend far beyond simple, fixed substring

presence. Data professionals should carefully evaluate the simplicity and high performance of `.contains()` against the expansive power and flexibility offered by `.rlike()` when selecting the most appropriate string method for their specific filtering requirements within a **DataFrame** context.

Conclusion and Further Resources

The `.contains()` function represents an essential and highly effective tool within the **PySpark** DataFrame API, purpose-built for executing straightforward substring matching and **filtering** operations. Its clear and minimal syntax facilitates the rapid isolation of records based on the presence of a specific string fragment within a column. We have meticulously detailed the implementation pathway, starting from the initialization of the **SparkSession** and the subsequent DataFrame creation, through to the execution of the filter and the analysis of the resulting subset. Achieving mastery of this fundamental operation is key to ensuring efficient and accurate data preparation and analysis across the vast distributed datasets managed by the Apache Spark engine.

It is crucial for all users to internalize the function's strict **case-sensitive** requirement. For applications demanding flexibility--specifically, case-insensitive matching or complex pattern recognition--alternative strategies are necessary. These include transforming the column case using functions like `lower()`, or more powerfully, leveraging the robust capabilities of regular expressions through the `.rlike()` function. Selecting the correct string operation is paramount for both optimizing performance across distributed systems and guaranteeing the accuracy of analytical outcomes in large-scale data processing environments.

To continue advancing your expertise in data manipulation utilizing distributed computing systems, we encourage exploration of related functions and essential transformations provided by the **PySpark** library. A solid understanding of handling diverse data types, executing sophisticated aggregations, and efficiently joining disparate datasets forms the backbone of modern big data workflow design. The following resources offer detailed tutorials on performing other common, yet essential, tasks within the **SparkSession** environment:

Additional Resources

Tutorial on advanced PySpark aggregation techniques.

Guide to implementing window functions for complex calculations.

Deep dive into optimizing PySpark joins for performance.