Learning PySpark: A Practical Guide to Filtering DataFrames with “Not Contains

Authored by

Mohammed looti

November 10, 2025

Mastering Exclusion Filtering in PySpark DataFrames

Data manipulation is the cornerstone of any analytical workflow or data pipeline. A critical and frequently performed operation within this process is filtering records based on specific criteria. When operating within the PySpark environment, which is designed for processing massive, distributed datasets, the syntax must be both efficient and explicitly clear. While including records that meet a condition is generally straightforward, the challenge often lies in the inverse operation: excluding records, or implementing a "Not Contains" filter. This technique is absolutely indispensable for robust data quality tasks, such as removing invalid entries, isolating subsets that deviate from expected norms, or simply cleaning up unwanted textual patterns within string columns.

The central structure we interact with in this distributed computing context is the DataFrame, which functions as a fault-tolerant, distributed collection of data organized into named columns. The inherent strength of the PySpark framework stems from its ability to execute complex logical tasks, including advanced string filtering, across an entire cluster. This parallelism handles voluminous information with high throughput and impressive speed. Therefore, mastering the methods for string exclusion is vital to ensure your data preparation phase is not only robust against errors but also highly performant. This comprehensive guide details how to leverage standard string methods and essential Boolean logic available in PySpark to achieve effective, precise exclusion filtering.

Before delving into the executable code, it is necessary to establish the foundational concepts. To perform a successful "Not Contains" operation, we must first identify the primary function responsible for checking containment, and then apply a necessary logical step to invert the function's result. This inversion, known as negation, is what transforms the standard "Contains" result into a "Not Contains" outcome. By understanding this core principle, developers can successfully isolate records where a specified substring is unequivocally absent from the target column, thus making the filtering logic crystal clear and scalable.

The Core Mechanism: `contains()` and Logical Negation (~)

In the PySpark API, the primary function utilized to determine if a column value holds a specific substring is the contains() method. This function is accessed directly through the column object of a DataFrame. Functionally, `contains()` returns a boolean outcome: it evaluates to **True** if the substring is present anywhere within the string value, and **False** if the substring is not found. Since our objective is exclusion, we only want to retain rows where the original containment condition evaluates to **False**.

To achieve this necessary logical inversion, we employ the Negation Operator (~). In Python and PySpark SQL expressions, the tilde symbol (~) is the standard boolean NOT operator. When this operator is placed immediately preceding the conditional expression, it flips the boolean outcome

of that condition. For instance, if `df.column.contains('substring')` returns **True** for a particular row (meaning the string is present), applying the tilde operator makes the entire expression **False**, consequently filtering out that row. Conversely, if the original expression is **False** (the string is absent), the tilde operator flips it to **True**, ensuring the row is retained in the resulting DataFrame.

The syntax required for executing this critical "Not Contains" filter is remarkably concise, combining the column reference, the containment function, and the negation operator seamlessly within the DataFrame's primary `filter()` method. This clean, expressive structure is one of the significant advantages of working with the PySpark API. The fundamental structure for applying a "Not Contains" filter to a DataFrame based on a specific column value is demonstrated below. This pattern is central to efficient string exclusion in distributed environments.

#filter DataFrame where team does not contain 'avs'
df.filter(~df.team.contains('avs')).show()

This single line of code provides explicit instructions to PySpark, commanding it to sweep through the entire distributed dataset. For every record, it evaluates the boolean outcome of the containment check and retains only those records where that containment check was logically negated--that is, where the specified substring was definitively absent. The following section provides a complete, runnable example, illustrating this powerful filtering mechanism in a real-world context.

Practical Implementation: Building and Filtering Data

To effectively illustrate the application of the negation filtering technique, we will begin by constructing a sample DataFrame. This DataFrame will contain simple, fictional basketball statistics, including a categorical column named `team`. This column will serve as the target for our string exclusion operation. Our objective is to systematically remove any team names that contain a specific substring ('avs'), simulating a typical data cleaning requirement where certain undesirable patterns or abbreviated entries must be eliminated from the dataset before analysis.

The initial setup necessitates defining the SparkSession, which serves as the entry point for all Spark functionality. We then define our data using a standard Python list of lists and explicitly declare the column schema. This clear declaration ensures structural integrity before creating the distributed DataFrame object. This setup process represents standard practice when initializing data for subsequent analysis or transformation within the PySpark environment, allowing us to accurately replicate complex, real-world data preparation tasks in a controlled manner.

The code block below demonstrates the complete setup of the sample data and the resulting initial

state of the DataFrame. Crucially, notice the inclusion of teams such as 'Mavs' and 'Cavs'; these intentionally contain the target substring ('avs') that we plan to exclude in the subsequent filtering step. The output confirms the structure and content of our starting dataset before any transformations are applied, establishing our baseline for the exclusion test.

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

Our primary goal is now to apply the string exclusion filter to this prepared dataset. We specifically

aim to refine the DataFrame such that it only includes rows where the `team` column does **not** contain the substring "avs". We achieve this precision by elegantly combining the contains() function with the Negation Operator (~), exactly as we established in the previous section. This method ensures all records failing the containment test are retained.

Executing the filter shown below results in a new DataFrame that successfully excludes all records associated with 'Mavs' and 'Cavs'. A careful analysis of the output confirms that the Negation Operator (~) correctly inverted the boolean outcome of the containment check. This inversion ensured that only teams without the target substring were retained, demonstrating the powerful and highly efficient manner in which PySpark handles sophisticated string exclusion on vast, distributed data volumes.

#filter DataFrame where team does not contain 'avs'
df.filter(~df.team.contains('avs')).show()

+------+------+
Nets	22
Nets	31
Kings	26
Spurs	40
Lakers	23
Spurs	17
+------+------+

Addressing Case Sensitivity and Normalization

A critical consideration when utilizing the contains() function in PySpark is its inherent case sensitivity. By default, this function performs an exact, literal match comparison between the target substring and the column data. This means that if you specify 'avs' as the exclusion criteria, it will not match variants like 'AVS' or 'Avs'. This behavior demands extreme precision when specifying the substring to be excluded, especially in real-world data where capitalization inconsistencies are common.

If the source column contains variations in capitalization (e.g., 'mAvs', 'Mavs', and 'MAVS'), a straightforward application of the exclusion logic, such as `~df.col.contains('avs')`, might fail to filter out all intended records. For example, the filter would successfully remove 'Mavs' but would inadvertently keep 'mAvs' because the exact lowercase 'avs' substring was not found due to the differing initial capitalization. This oversight is a common pitfall in string filtering operations across various SQL and distributed computing environments, often leading to incomplete data cleansing or skewed analytical results.

To comprehensively address the requirement for case-insensitive exclusion, the preferred and most robust strategy involves standardizing the case of the target column immediately before applying the containment check. This is typically achieved by chaining the `lower()` function, imported from `pyspark.sql.functions`, directly before the contains() method. By transforming the column data to a consistent lowercase, we ensure that both the column data and the filter substring are normalized, guaranteeing a comprehensive match regardless of the original capitalization within the source data. The resulting syntax, `df.filter(~df.team.lower().contains('avs'))`, effectively guarantees that all variations--'Mavs', 'mavs', 'MAVS', and 'mAvs'--are correctly identified and excluded, vastly improving the reliability of the data preparation step.

Advanced Exclusion: Leveraging `like` and `rlike` for Pattern Matching

While the combination of contains() and the Negation Operator (~) is highly effective for simple substring exclusion, PySpark provides more powerful alternatives for complex pattern matching: the `like` and rlike methods. These alternatives are indispensable when dealing with data that requires fuzzy matching using wildcards or adherence to formal regular expression patterns, which often arise in cleaning unstructured text fields.

The `like` function is a direct analog to the standard SQL `LIKE` operator. It allows data engineers to utilize SQL wildcards: the percent sign (`%`) represents zero or more characters, and the underscore (_) represents a single character. When paired with the Negation Operator (~), `like` facilitates exclusion based on fuzzy or positional patterns. For example, to exclude any team name that begins with 'S' and ends with 's', regardless of the intervening characters (like 'Spurs' or 'Sixers'), you would use: `df.filter(~df.team.like('S%s'))`. This approach offers significantly greater flexibility than `contains()` when the position of the substring within the column value is relevant, or when you need to match across the entire string length based on boundary conditions.

For the maximum degree of complexity and precision in defining exclusion patterns, the rlike method (Regular Expression Like) is the definitive tool. rlike enables developers to define intricate, highly specific patterns using industry-standard regular expression syntax. This capability is invaluable when excluding data that matches complex, non-contiguous patterns, or when filtering based on character sets, repetition, or predefined boundaries (e.g., excluding any string that contains exactly three consecutive numeric digits). The syntax for negation remains consistent, leveraging the tilde operator: `df.filter(~df.team.rlike('regex_pattern'))`. While requiring a deeper understanding of regex, rlike provides the most robust and versatile solution for meeting the most advanced exclusion requirements within high-stakes data processing pipelines.

Summary and Best Practices for Data Exclusion

Filtering for "Not Contains" is an absolutely essential requirement in professional data preparation, ensuring that irrelevant, incomplete, or unwanted data points are efficiently and accurately removed from the analytical set. In the PySpark ecosystem, this functionality is achieved by combining the appropriate string function--most commonly the contains() method--with the logical Negation Operator (~). This straightforward syntax provides a powerful, distributed mechanism for high-volume data cleansing and refinement.

To guarantee maximum reliability and computational efficiency when implementing exclusion filters on large DataFrames, several critical best practices should always be observed. First and foremost, always be acutely mindful of **case sensitivity**; if case-insensitivity is required, consistently preprocess the column data using `lower()` or `upper()` before applying any filter. Second, select the most appropriate string matching method for the task at hand: use `contains()` for simple substring checks, `like` for SQL-style wildcard matching, and the highly versatile rlike for complex, regular expression-based exclusions.

Finally, it is crucial to remember that filter operations in PySpark are inherently **lazy**. The filter logic defined using the tilde operator will not execute immediately; it is only triggered when an action (such as `.show()`, `.count()`, or `.write()`) is called upon the resulting DataFrame. This fundamental distributed nature ensures that filtering is performed in parallel across the entire cluster, maintaining peak performance even as dataset sizes scale into terabytes. By diligently adhering to these principles and selecting the correct tool for string matching, data engineers can write clean, efficient, and robust exclusion logic necessary for maintaining high-quality data engineering workflows.

Additional Resources

The following tutorials explain how to perform other common tasks in PySpark:

PySpark: Handling Null Values: Learn how to manage missing data within your distributed DataFrames using functions like `fillna()` and `dropna()`.

PySpark: Conditional Column Creation: Discover how to use `when()` and `otherwise()` clauses to create new columns based on complex conditions, enhancing your data transformation capabilities.

PySpark: Aggregation Techniques: Explore efficient ways to group and summarize data using distributed aggregation functions across your cluster.