How to Filter a PySpark DataFrame Using the Contains Function

Authored by

stats writer

February 10, 2026

Understanding the Power of PySpark in Modern Data Engineering

In the contemporary landscape of **data engineering** and **data science**, the ability to process massive datasets with speed and precision is paramount. **PySpark**, the **Python** API for **Apache Spark**, has emerged as the industry standard for **distributed computing**. By leveraging the computational power of a cluster, **PySpark** allows developers to perform complex operations on **Big Data** that would be impossible on a single machine. One of the most fundamental yet powerful operations within this framework is the ability to filter a **DataFrame** based on specific textual criteria, a task frequently accomplished using the **contains** function.

The **contains** function is an essential tool for **data manipulation**, specifically designed to identify rows where a column's value matches a specified substring. Unlike exact matching, which requires a cell to be identical to a search term, **contains** evaluates whether the target **string** exists anywhere within the data entry. This flexibility is vital when dealing with unstructured or semi-structured data where relevant information might be embedded within longer text fields, such as log files, social media posts, or descriptions. By utilizing this function, analysts can refine their focus to specific subsets of information, thereby streamlining the **exploratory data analysis** (EDA) process.

Furthermore, the integration of **contains** within the **PySpark** ecosystem ensures that these filtering operations are executed in a highly optimized manner. Spark's **Catalyst Optimizer** and **Tungsten** execution engine work behind the scenes to translate high-level **Python** code into efficient physical execution plans. This means that whether you are filtering a few thousand rows or several billion, the **contains** function provides a scalable solution for **pattern matching** and **data cleaning**. Understanding the nuances of this function is the first step toward mastering **data wrangling** at scale.

The Role of String Filtering in Comprehensive Data Analysis

Filtering data is often the most time-consuming part of any **data pipeline**. When working with **Relational Databases** or **Data Lakes**, the initial extraction often results in a broad dataset that contains significant noise. The **contains** function serves as a primary filter to reduce this noise, allowing the user to isolate rows that hold true value for their specific research question. For instance, in a dataset containing thousands of medical records, one might use **contains** to find every instance where a "diagnosis" column mentions a specific condition, regardless of other text present in the field.

Beyond simple identification, the **contains** function is a cornerstone of **feature engineering**. Data scientists often create new binary columns (often called **indicator variables** or **dummy variables**) based on whether a certain keyword exists in a text field. By applying a filter or a conditional

transformation using **contains**, one can easily categorize data into meaningful groups. This process is instrumental in preparing data for **Machine Learning** models, where qualitative text must often be converted into quantitative signals that a model can interpret.

The clarity provided by efficient filtering cannot be overstated. When a **DataFrame** is correctly filtered, the subsequent **aggregations**, **joins**, and **visualizations** become far more impactful. By removing irrelevant records early in the **workflow**, you not only improve the readability of your results but also significantly reduce the computational overhead of downstream operations. This makes the **contains** function a critical component of any robust **data architecture**.

PySpark: Filter Using Contains"

Implementing the Filter Syntax in PySpark DataFrames

To implement a substring search effectively, one must understand the specific syntax required by the PySpark API. The filter method (which is an alias for the where method) is used in conjunction with the Column object's contains method. This allows for a clean, readable programmatic interface that mirrors SQL logic while maintaining the benefits of Python's object-oriented structure. The following syntax represents the most common approach to filtering a DataFrame based on whether a specific column contains a target sequence of characters:

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

In this code snippet, df.team references a specific column within the DataFrame named df. The contains('avs') method is then invoked on that column, which generates a Boolean column expression. When passed into the filter function, PySpark evaluates this expression for every row in the dataset, retaining only those where the condition is True. Finally, the show() action is called to display the resulting records to the console, providing immediate feedback on the operation's success.

It is important to note that the contains method is just one of many string functions available in the pyspark.sql.functions module. While the example above uses the dot notation to access the column, you can also use bracket notation or the col() function, which is often preferred for more complex or dynamic column names. Regardless of the access method, the underlying logic remains consistent: searching for a substring within a larger string to facilitate precise data extraction.

Practical Demonstration: Filtering Basketball Team Data

To truly grasp how the contains operator functions, it is

helpful to examine its application within a real-world scenario. Consider a dataset representing various professional basketball teams and their respective performance metrics, such as points scored in a series of games. In many instances, team names share common suffixes or prefixes, and an analyst might need to group these teams based on those shared string patterns. The following example demonstrates the creation of a **DataFrame** and the subsequent application of the contains filter:

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

In the initialization phase of this script, a **SparkSession** is created, which serves as the entry point for all **Apache Spark** functionality. We define a list of lists containing our raw data and a list of strings for our

column headers. Using the createDataFrame method, we transform these local Python objects into a distributed dataset. This transition is crucial, as it moves the data from the local memory of the driver to the distributed memory of the Spark cluster, enabling the use of Spark's powerful transformation libraries.

The resulting table displays a variety of teams, including the "Mavs" and "Cavs." Suppose our objective is to isolate all teams whose names end with or contain the substring "avs." This specific pattern is common across several franchise names in this dataset, and manually selecting each one would be inefficient and prone to error, especially if the dataset contained thousands of unique entries. By using a programmatic filter, we ensure reproducibility and scalability in our analysis.

Analyzing Filter Results and Pattern Matching Precision

Once the **DataFrame** has been established, we can apply the contains logic to extract the specific rows of interest. The power of this approach lies in its ability to find the "avs" sequence regardless of its position in the string, although in this case, it appears at the end of

"Mavs" and "Cavs." The execution of the filter is shown below:

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

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

As illustrated by the output, only the rows corresponding to "Mavs" and "Cavs" remain. The teams "Nets," "Kings," "Spurs," and "Lakers" were successfully excluded because the substring "avs" is not present within their names. This demonstrates the precision of the contains method; it acts as a boolean gate, allowing only the data that strictly meets the substring criteria to pass through to the next stage of the data pipeline.

This type of filtering is particularly useful in data auditing. By filtering for specific patterns, data

engineers can quickly identify anomalies or group records that follow specific naming conventions. The contains function effectively acts as a lightweight search engine within your distributed environment, providing a simple yet robust mechanism for navigating large volumes of text-based information. It ensures that the analyst can focus on the relevant 20% of data that often drives 80% of the business insights.

Addressing Case Sensitivity and Character Encoding

One critical technical detail to keep in mind when using the contains function is that it is inherently **case-sensitive**. This means that the string "avs" is treated as distinct from "AVS" or "Avs." In a real-world environment where data entry might be inconsistent, this behavior can lead to missing relevant records if the user is not careful. For example, if you were to search for "AVS" in the basketball dataset, the filter would return an empty **DataFrame**, even though "Mavs" and "Cavs" exist, because the uppercase "A" does not match the lowercase "a" in the source data.

To mitigate issues related to **case sensitivity**, it is standard practice to normalize the casing of the column

before applying the filter. This can be done using the lower() or upper() functions from the pyspark.sql.functions module. By converting the entire column to lowercase and then searching for a lowercase substring, you ensure that your filter captures all variations of the word regardless of how they were originally typed. This defensive programming technique is essential for maintaining data quality in production environments.

Additionally, users should be aware of Unicode and special characters. The contains function works by comparing character encodings. If your data contains accented characters or non-Latin scripts, the substring you are searching for must match those specific characters exactly. In global datasets, handling these nuances is vital to ensure that your data analysis remains inclusive and accurate across different languages and regions.

Comparative Analysis: `contains` vs. `like` and `rlike`

While contains is excellent for simple substring searches, PySpark offers other methods that provide even greater flexibility. The like operator, familiar to

anyone with SQL experience, allows for the use of wildcards such as % (representing zero or more characters). For example, df.team.like('%avs%') would function similarly to contains, but like allows for more complex positioning, such as ensuring a string starts or ends with a certain pattern.

For even more advanced pattern matching, PySpark provides the rlike function, which supports Regular Expressions (regex). Regex is a powerful language for describing search patterns, enabling users to search for digits, whitespace, specific character ranges, or repeating sequences. While contains is faster and easier to read for basic tasks, rlike is the tool of choice when the search criteria involve complex logic that cannot be expressed by a simple substring.

Choosing between these methods depends on the balance between performance and complexity. Contains is highly optimized for literal substring matching and should be the default choice for such tasks. However, as your data engineering requirements evolve to include complex validation or sophisticated text parsing, transitioning to rlike will provide the necessary

power to handle those intricate requirements within the Apache Spark framework.

Performance Optimization for Large-Scale Filtering

When working with Big Data, performance is always a primary concern. Filtering a DataFrame using contains is a narrow transformation, meaning it does not require data to be shuffled across the network between executors. This makes it an inherently efficient operation. However, there are still ways to optimize the process. For instance, applying filters as early as possible in your Spark job (a concept known as predicate pushdown) ensures that only the necessary data is loaded into memory, significantly reducing I/O and processing time.

Another factor to consider is the partitioning of your data. If your dataset is partitioned by a column that you are also filtering on, Spark can skip entire directories of data that do not match your criteria. While contains is often used on high-cardinality string columns that are not suitable for partitioning, combining a contains filter with a filter on a partitioned column (like a date or region) can result in dramatic performance gains. This

strategy is a cornerstone of Query optimization in distributed systems.

Finally, it is worth noting that Spark's lazy evaluation model means that the filter is not actually executed until an action like show(), count(), or save() is called. This allows the Query optimizer to look at the entire chain of transformations and consolidate them. When you chain multiple contains filters or combine them with other operations, Spark creates a single, optimized execution plan that minimizes the number of passes over the data, ensuring maximum efficiency for your Big Data tasks.

Handling Nulls and Edge Cases in Column Operations

Data in production environments is rarely perfect, and null values are a common occurrence in string columns. When the contains function encounters a null value, the result of the evaluation is null, which the filter method treats as False. Consequently, any rows containing null in the target column will be excluded from the results. It is important for data analysts to decide whether these nulls should be ignored or if they need to be handled explicitly using functions like fillna() or coalesce().

Another edge case involves empty strings. An empty string '' is technically contained within every other string. However, searching for an empty string using contains usually yields all non-null rows, which might not be the intended behavior. Understanding these subtle interactions between the data and the function logic is key to avoiding data leakage or incorrect conclusions during the analysis phase. Always validate your data distribution before and after applying filters to ensure the logic aligns with your business requirements.

To handle these situations gracefully, you might use the isNotNull() method in conjunction with contains. By explicitly checking for the presence of data before attempting a substring search, you can create more resilient data pipelines. This level of detail in exception handling and data validation is what separates basic scripts from professional-grade data engineering solutions.

Conclusion and Strategic Data Integration

The contains function in **PySpark** is a versatile and high-performance tool that is indispensable for anyone

working with distributed datasets. Its ability to quickly isolate rows based on substring patterns makes it a foundational element of data cleaning, exploratory analysis, and feature engineering. By mastering the syntax and understanding the underlying mechanics of how Spark processes these filters, you can significantly enhance the efficiency of your data workflows.

As you continue to build your skills in **Big Data**, remember that the contains function is just one part of a larger toolkit. Combining it with other SQL-like operations, handling case sensitivity with string normalization, and optimizing performance through strategic predicate pushdown will allow you to tackle even the most challenging data manipulation tasks. The clarity and structure provided by **PySpark**'s DataFrame API ensure that your code remains readable and maintainable as your projects grow in complexity.

The following tutorials and documentation explain how to perform other common tasks in **PySpark**, offering a deeper look into the world of distributed computing. By staying informed on the latest best practices and API updates from the **Apache Spark** community, you can

ensure that your data engineering expertise remains at the cutting edge of the industry.