How to Remove Characters from Strings in PySpark

Authored by

stats writer

February 7, 2026

Removing specific characters from strings in PySpark is a fundamental requirement during the data cleaning and preprocessing phases of any big data project. Unwanted characters, such as special symbols, extraneous delimiters, or embedded noise, often clutter raw datasets and impede accurate analysis. To address this efficiently within the distributed environment of the PySpark framework, developers primarily rely on powerful, optimized SQL functions.

The primary tool for complex character removal is the built-in function regexp_replace. This function leverages the flexibility of a regular expression pattern to identify and replace matching substrings with a specified replacement string, typically an empty string (`''`) when the goal is pure removal. Additionally, the less common `translate` function can also be used to remove specific characters by replacing them with an empty string, though it is better suited for fixed character substitutions rather than complex pattern matching.

Mastering these functions is essential for maintaining data integrity when working with massive scale DataFrames in PySpark, ensuring that subsequent transformations or machine learning models receive clean, standardized input. This guide will walk through practical examples demonstrating the use of regexp_replace for both single and multiple character removals, providing robust solutions for common data cleaning challenges.

PySpark: Remove Specific Characters from Strings

Introduction to String Manipulation in PySpark

When dealing with large volumes of unstructured or semi-structured data, string manipulation becomes a critical component of any extract, transform, load (ETL) pipeline. DataFrames in PySpark provide powerful, distributed methods to perform these operations without relying on inefficient, row-by-row Python User Defined Functions (UDFs).

To effectively cleanse string columns--such as removing punctuation, proprietary codes, or boilerplate text--we utilize optimized functions available through `pyspark.sql.functions`. The two main functions introduced here offer distinct ways to achieve character removal, catering to different levels of complexity and performance requirements. Selecting the correct tool depends on whether the removal criteria involve simple, fixed characters or complex, dynamic patterns.

We will focus heavily on the regexp_replace function due to its versatility. By accepting a regular expression as its search pattern, it allows for sophisticated matching and removal of characters that adhere to complex rules, far exceeding the capability of simple string replacement methods found in standard Python libraries.

The Power of `regexp_replace`

The `regexp_replace` function is arguably the most flexible tool for string data cleaning in DataFrame operations. It takes three primary arguments: the column to operate on, the regular expression pattern to search for, and the replacement string. When the replacement string is set to an empty string (`''`), the effect is the removal of any text matching the pattern.

The standard syntax for applying this function to an existing DataFrame `df` to create a new column or overwrite an existing one (`team`, in our examples) is structured as follows:

Method 1: Remove Specific Characters from String

from pyspark.sql.functions import*#remove 'avs' from each string in team column
df_new = df.withColumn('team', regexp_replace('team', 'avs', ''))

This method explicitly targets the sequence of characters `avs` and removes them, leaving the rest of the string untouched. It is essential to remember that since we are using regexp_replace, the search string `'avs'` is treated as a literal regular expression pattern.

Advanced Usage: Handling Multiple Character Groups

One of the significant advantages of using regexp_replace is the ability to remove multiple, non-contiguous sets of characters simultaneously using the alternation operator (`|`) within the regular expression. This prevents the need for chaining multiple `withColumn` calls, which can sometimes impact performance and code readability.

By concatenating the specific substrings or patterns we wish to remove using the `|` symbol, we instruct the function to search for an instance of Pattern A OR Pattern B. This results in a single, highly efficient transformation operation applied across the distributed dataset. For instance, if we needed to eliminate both `avs` and `awks` from a column, the approach simplifies dramatically:

Method 2: Remove Multiple Groups of Specific Characters from String

from pyspark.sql.functions import*#remove 'avs' and 'awks' from each string in team column
df_new = df.withColumn('team', regexp_replace('team', 'avs|awks', ''))

This technique is particularly useful when normalizing data that contains various forms of noise or consistent but different abbreviations that need to be stripped out before analysis.

Setting Up the Example PySpark DataFrame

To illustrate the practical application of these methods, we will first define and create a sample DataFrame. This dataset contains information about basketball teams and corresponding points, where the team names contain the specific character groups we intend to remove.

The following code snippet sets up a local Spark session and generates the sample data, which is crucial for testing and validating the string replacement logic discussed above:

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| 18|
| Nets| 33|
|Hawks| 12|
| Mavs| 15|
|Hawks| 19|
| Cavs| 24|
|Magic| 28|
+-----+------+

As observed in the output, teams like 'Mavs', 'Hawks', and 'Cavs' contain the character sequences we target for removal. Our goal is to sanitize the `team` column, resulting in abbreviations or shortened names (e.g., 'M' or 'H').

Practical Application: Removing a Single Character Pattern

Let's apply the first method, focusing on removing the substring "avs" wherever it appears within the `team` column. This is a common requirement when standardizing team names or product identifiers where a consistent suffix or internal code needs elimination.

We use the `withColumn` transformation in conjunction with `regexp_replace` to apply the logic across all records in a distributed manner. The transformation is lazy and executed only upon an action like `df_new.show()`.

Example 1: Remove Specific Characters from String

We can use the following syntax to remove "avs" from any string in the **team** column of the DataFrame:

from pyspark.sql.functions import*#remove 'avs' from each string in team column
df_new = df.withColumn('team', regexp_replace('team', 'avs', ''))

#view new DataFrame
df_new.show()

+-----+------+
| team|points|
+-----+------+
| M| 18|
| Nets| 33|
|Hawks| 12|
| M| 15|
|Hawks| 19|
| C| 24|
|Magic| 28|
+-----+------+

Upon reviewing the resulting DataFrame, observe that "avs" has been successfully removed from "Mavs" (resulting in "M") and "Cavs" (resulting in "C"). Crucially, rows that did not contain the sequence (like "Nets", "Hawks", and "Magic") remain completely unchanged, demonstrating the precision of the function.

Example 2: Removing Multiple Character Groups Simultaneously

Building on the previous concept, we now demonstrate how to clean the data for two distinct patterns, "avs" and "awks", in a single operation. This powerful use case highlights why leveraging regular expression syntax within regexp_replace is efficient for complex data cleaning tasks.

By using the pattern `'avs|awks'`, we ensure that if either sequence is found, it is replaced by an empty string, effectively removing it from the dataset.

We can use the following syntax to remove the strings "avs" and "awks" from any string in the **team** column of the DataFrame:

from pyspark.sql.functions import*#remove 'avs' and 'awks' from each string in team column
df_new = df.withColumn('team', regexp_replace('team', 'avs|awks', ''))

#view new DataFrame
df_new.show()

+-----+------+
| team|points|
+-----+------+
M	18
Nets	33
H	12
M	15
H	19
C	24
Magic	28
+-----+------+

Notice that the strings "avs" and "awks" have both been successfully removed. "Hawks" is now reduced to "H", while "Mavs" remains "M", and "Cavs" remains "C". This confirms that a single invocation of `regexp_replace` is sufficient for multi-pattern replacement scenarios.

Important Considerations for PySpark String Operations

When implementing these character removal strategies in a production environment using PySpark, developers must keep several critical details in mind regarding execution and function properties:

Case Sensitivity: It is crucial to remember that the `regexp_replace` function, by default, performs a **case-sensitive** match. If your input data contains variations (e.g., "Mavs", "MAVS", "mavs"), your regular expression must account for all potential casings, either through explicit alternation (`'avs|AVS'`) or by using flags if supported by the Spark environment's regex engine, or by first normalizing the case of the column (e.g., using `lower()`).

Regular Expression Syntax: Since these functions rely on Java's regular expression engine (as Spark is written in Scala/Java), specialized characters (like `.`, `*`, `+`,) must be properly escaped if they are meant to be treated as literal characters within the string you are trying to remove.

Performance: While `regexp_replace` is highly optimized compared to UDFs, complex regular expressions can still introduce computational overhead. For very simple, fixed-length character removals, especially involving character substitution rather than pattern matching, the `translate` function might offer marginal performance benefits.

Note #1: The **regexp_replace** function is case-sensitive.

Note #2: You can find the complete documentation for the PySpark regexp_replace function on the official Apache Spark documentation website, which provides exhaustive details on arguments and return types.

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