Learn How to Split String Columns in PySpark DataFrames

Authored by

Mohammed looti

November 11, 2025

Introduction: Mastering String Manipulation in PySpark

Data cleansing and preparation are fundamental steps in any robust Extract, Transform, Load (ETL) pipeline. Often, crucial pieces of information are concatenated within a single string column, requiring sophisticated techniques to separate them into distinct, usable fields. When dealing with massive datasets, utilizing the distributed processing power of PySpark becomes essential. This article provides an in-depth guide on how to effectively split a string column into multiple columns within a DataFrame, focusing on clarity, efficiency, and the powerful built-in functions provided by the framework. Understanding this technique is vital for analysts and data engineers working with complex, unstructured string data.

The process of splitting strings involves identifying a consistent character or sequence of characters--known as a delimiter--that acts as the boundary between the data segments we wish to isolate. Whether this delimiter is a comma, a pipe, or, as shown in our primary example, a hyphen, the approach remains largely the same. In PySpark, we rely on the functions module, specifically the designated splitting function, to perform this transformation across potentially billions of rows in a parallelized fashion. This capability is what distinguishes distributed computing tools like Spark from traditional, single-machine processing methods, ensuring scalability and performance for big data tasks.

Below is the foundational syntax used to achieve this column separation. This pattern is reusable across various data formats and delimiter types, establishing it as a core skill in PySpark data wrangling. We utilize the withColumn transformation combined with the specialized splitting logic to generate the desired result columns efficiently.

from pyspark.sql.functions import split

```
#split team column using dash as delimiter
df_new = df.withColumn('location', split(df.team, '-').getItem(0))
.withColumn('name', split(df.team, '-').getItem(1))
```

This code snippet demonstrates the high-level methodology: applying the split function to the source column, specifying the hyphen as the separator, and then using array indexing (`getItem(0)` and `getItem(1)`) to pull out the resulting fragments. Specifically, this example takes the data from the **team** column and extracts the team's **location** and its **name** into two distinct new fields. This transformation is crucial for normalizing data structures, allowing subsequent analytical queries or machine learning models to operate on structured, atomic features rather than complex, composite strings.

The Core Mechanics: Leveraging the PySpark Split Function

The efficiency of this operation relies almost entirely on the split function, which is imported from `pyspark.sql.functions`. When this function is applied to a string column in a PySpark DataFrame, it evaluates each row independently and returns an array (or list) of strings. Each element in this array corresponds to a segment of the original string that was separated by the specified delimiter. It is critical to recognize that the output of the splitting operation is not immediately two separate columns, but rather a single column containing an array type, which must then be deconstructed.

To convert this array output into multiple scalar columns, we chain the splitting operation with the array indexing function, `getItem()`. The resulting array is zero-indexed, meaning the first separated string is accessed via `getItem(0)`, the second via `getItem(1)`, and so forth. For every new column we wish to create from the split, we must call the entire `split()` function again, followed by the specific `getItem()` index relevant to that new column. While this might seem redundant, PySpark's Catalyst Optimizer is sophisticated enough to recognize and optimize these repeated calls, ensuring that the underlying splitting computation is performed only once per row, maintaining high performance across the cluster.

We manage the creation of these new columns using the powerful withColumn method. The `withColumn` transformation is a standard operation in PySpark DataFrames used to add a new column or replace an existing one. By combining `withColumn` with the `split(...).getItem(...)` expression, we are instructing Spark to derive the value of the new column (e.g., 'location') from a specific index of the array produced by the splitting of the source column (e.g., 'team'). This declarative approach ensures that the data lineage is maintained and transformations are executed lazily and optimally across the distributed cluster resources.

Practical Demonstration: Setting up the PySpark Environment and Data

To fully illustrate this process, let us consider a concrete example involving basketball player data. Suppose we have records where the team and location information are combined into a single string. Our goal is to cleanly separate these components for easier analysis, such as calculating average points scored per location or grouping statistics by team name. This practical setup helps solidify the theoretical understanding of the functions involved.

The first step is always to initialize the Spark Session, which is the entry point for using Spark functionality. Once the session is active, we define our sample data structure, which in this case is a list of lists, and specify the corresponding column names. This preparation phase simulates the loading of data from an external source, such as a CSV or Parquet file, into the DataFrame structure.

The following code block demonstrates the necessary setup, resulting in a PySpark DataFrame containing information about various basketball players. Notice how the 'team' column contains the hyphenated city and mascot name, which we intend to separate in the subsequent steps.

Example: Splitting a String into Multiple Columns in PySpark

Suppose we have the following PySpark DataFrame that contains information about various basketball players:

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|
+---------------+------+
Dallas-Mavs	18
Brooklyn-Nets	33
LA-Lakers	12
Houston-Rockets	15
Atlanta-Hawks	19
Boston-Celtics	24
Orlando-Magic	28
+---------------+------+

Our objective is clear: to split the strings in the **team** column into two distinct, new columns based on the position of the hyphen (delimiter) within the strings. This transformation will convert the compound data structure into an atomic structure, enabling seamless SQL queries and aggregations.

Implementing the String Split Operation

With the DataFrame prepared, we now execute the core logic using the functions discussed previously. We import the necessary `split` function and apply the chained transformation using withColumn. It is important to note the robustness of this method; even if the string format changes slightly (e.g., containing spaces around the hyphen), the regex capabilities of the splitting function can be leveraged for more complex parsing, though here we use a simple, literal hyphen.

The key to achieving the final separated columns lies in correctly identifying the indices. Since the hyphen splits the original string into two parts, we use index 0 for the first part (the city/location) and index 1 for the second part (the team name/mascot). If the string contained more delimiters, resulting in three segments, we would utilize `getItem(2)`, and so on. Careful verification of the source data format is always required before assigning array indices.

Executing the syntax below generates a new DataFrame, `df_new`, which includes the original columns plus the two newly derived columns, `location` and `name`. Observing the resulting output confirms the successful transformation of the structured string data into normalized fields.

from pyspark.sql.functions import split

#split team column using dash as delimiter
df_new = df.withColumn('location', split(df.team, '-').getItem(0))
.withColumn('name', split(df.team, '-').getItem(1))

#view new DataFrame
df_new.show()

+---------------+------+--------+-------+
| team|points|location| name|
+---------------+------+--------+-------+
Dallas-Mavs	18	Dallas	Mavs
Brooklyn-Nets	33	Brooklyn	Nets
LA-Lakers	12	LA	Lakers
Houston-Rockets	15	Houston	Rockets
Atlanta-Hawks	19	Atlanta	Hawks
Boston-Celtics	24	Boston	Celtics

```
| Orlando-Magic| 28| Orlando| Magic|
+---------------+------+--------+-------+
```

As evidenced by the output, the strings in the **team** column have been successfully decomposed and distributed into the new columns called **location** and **name** based on the hyphen. This successful demonstration highlights the power and simplicity of the distributed split function in handling common data transformation requirements within the PySpark environment. We utilized the **split** function to break down each string into an array of two new strings, and subsequently used **getItem(0)** and **getItem(1)** to extract the required segments for each team record.

Advanced Considerations and Alternative Techniques

While the standard `split` and `getItem` method is ideal for fixed-format string parsing, data engineering often requires more flexible or specialized approaches. For instance, if the delimiter itself might appear within the data we want to preserve, or if the number of segments varies row by row, alternative strategies become necessary. One such strategy involves using regular expressions within the split function, as the function supports regex patterns, providing much greater control over the splitting criteria. When dealing with complex textual data, functions like `regexp_extract` might be preferred, as they allow for direct extraction of segments based on pattern matching groups, rather than relying solely on the delimiter to break up the string.

Another critical aspect is handling errors and missing data. What happens if a row in the original 'team' column does not contain the specified delimiter? In such cases, the array produced by `split` will contain only one element (the original string itself). Attempting to access `getItem(1)` on such a row will result in a `null` value for the corresponding new column. Data professionals must anticipate this behavior and implement defensive coding strategies, perhaps using `when` and `otherwise` clauses to impute defaults or log errors when expected segments are missing. Ensuring data quality post-split is just as important as the split operation itself.

Finally, for extremely large DataFrames where performance is paramount, minimizing the number of times the `split` function is explicitly called can be beneficial, even with PySpark's optimization capabilities. An advanced technique involves splitting the column once and storing the resulting array in a temporary column, and then using the `select` or `withColumn` operation to access the elements of this array column directly using the array index notation (e.g., `col('temp_array_col')`). This is often cleaner syntactically, though the performance difference compared to the chained `split().getItem()` approach is often negligible due to the underlying optimizations within PySpark.

Additional Resources for PySpark Mastery

The process of splitting string columns is a cornerstone of data preparation in distributed systems. To further expand your expertise in PySpark, consider exploring related functionalities that involve column manipulation and string handling. Mastering the various functions within `pyspark.sql.functions` will significantly enhance your ability to perform complex ETL tasks efficiently.

The following tutorials explain how to perform other common tasks in PySpark that complement this string splitting technique:

How to handle null or missing values during transformation.
Advanced techniques for using regular expressions (`regexp_extract`, `regexp_replace`) for non-standard string formats.
Methods for concatenating columns or aggregating string data back together.
Optimizing DataFrame operations using caching and partitioning strategies.

Remember that comprehensive documentation for the PySpark **split** function, along with all other SQL functions, is available on the official Apache Spark website. Utilizing these resources will ensure you are implementing the most current and efficient methods for your data pipelines.