How to Split a String Column into Multiple Columns in PySpark

Authored by

stats writer

February 5, 2026

In PySpark, a string column can be efficiently split into multiple columns by leveraging the specialized `split` function available in the `pyspark.sql.functions` module. This operation is fundamental in data cleaning and feature engineering, especially when dealing with compound fields where critical information is concatenated using a specific character. The split function is designed to handle this transformation seamlessly, requiring explicit definition of the dividing mechanism.

The function fundamentally requires two primary inputs: the column containing the string data to be processed and the delimiter--the character or sequence of characters that signals where the string should be broken apart. While the function supports an optional third argument specifying the maximum number of splits, typically in basic scenarios, defining the input column and the delimiter suffices. Once executed, the `split` function returns an array type column, where each element in the array represents a segment of the original string delimited by the specified character. This intermediate array structure must then be systematically unpacked into the desired individual columns using subsequent transformation steps. This methodology is crucial for organizing, manipulating, and ultimately enabling far easier analysis and processing of complex string data within large-scale data environments managed by PySpark.

PySpark: How to Split a String Column into Multiple Columns

Understanding String Manipulation in PySpark

Data processing often necessitates the decomposition of complex data structures into simpler, more manageable components. In the context of big data frameworks like PySpark, dealing with string columns that contain concatenated values (such as names, addresses, or identification codes) is a routine task. Effective string manipulation is critical for feature engineering, ensuring that analytical models receive granular, atomic pieces of information rather than single, dense strings. The ability to reliably parse these fields hinges on understanding the role of the DataFrame, which serves as the fundamental data structure for structured data operations within the Spark ecosystem.

When approaching a splitting operation, it is essential to first identify the consistent pattern used to separate the elements within the string--this is the target delimiter. Whether it is a comma, a pipe, a space, or a hyphen, the PySpark `split` function relies on this pattern to correctly segment the data. The resulting segments are stored temporarily as elements in a list or array type within the DataFrame. This intermediate step is powerful because it retains the sequence of the original data, allowing for precise extraction of specific components based on their positional index.

The core objective is transforming a single string column (e.g., `City-State-Zip`) into multiple distinct, atomic columns (`City`, `State`, `Zip`). This transformation is not only beneficial for readability but is mandatory for many downstream operations, including joins, aggregations, and filtering, where individual attributes must be isolated. Utilizing functions from `pyspark.sql.functions` ensures that these transformations are performed efficiently across the distributed cluster, maintaining the integrity and performance expected of a Spark operation.

The PySpark split Function Explained

The `split` function is the workhorse for breaking down strings in PySpark. It belongs to the set of Column functions and is imported specifically from `pyspark.sql.functions`. The function requires the column input and the regular expression pattern (or string literal) representing the delimiter. It returns a column of type `Array`. Understanding this return type is paramount because a single function call to `split` does not automatically create new columns; it only transforms the data type of the resulting output into an array structure.

Consider a scenario where a string column contains "Firstname_Lastname". If we use the underscore (`_`) as the delimiter, the `split` function will produce an array like for each row. To turn these array elements into separate, named columns, we must chain additional operations. Specifically, we utilize the indexing capability inherent in array types, accessing elements using methods like `getItem(index)` or direct bracket notation, though `getItem()` is often preferred for clarity and robustness in PySpark syntax.

The optional third argument, `limit`, controls the maximum number of times the split occurs. If `limit` is positive, the pattern will be applied at most `limit - 1` times, and the resulting array will contain at most `limit` strings. If `limit` is zero or negative, the array length is unbounded, and trailing empty strings are discarded or retained, respectively. For most standard data cleaning tasks, omitting the limit means all occurrences of the delimiter are used, ensuring a complete breakdown of the string based on the pattern provided.

Syntax for Splitting Columns and Creating New Fields

To implement the split operation and integrate the results into the existing DataFrame, we must combine the `split` function with the withColumn method. The `withColumn` transformation is used to add a new column or replace an existing one, making it the perfect vehicle for integrating the extracted array elements. Since the `split` function returns an array, we must call `getItem(N)` immediately after the split operation to select the element corresponding to the N-th position (zero-indexed).

The core syntax involves calling `df.withColumn()` multiple times, once for each new column

required. For a string split into two parts (index 0 and index 1), the process looks like this:

You can use the following syntax to split a string column into multiple columns in a PySpark DataFrame:

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 particular example illustrates the necessity of chaining operations. First, we import the required function. Then, we apply the `split` function to the **team** column using a dash (`-`) as the separator. Since the output is an array, we immediately use `.getItem(0)` to capture the first element (the location name) and assign it to the new column `location`. This process is repeated, using `.getItem(1)` to capture the second element (the team nickname) and assign it to the column `name`. It is crucial to remember that array indexing in Python and PySpark starts at zero (0).

While this method involves repeating the `split` function call for every new column created, Spark's Catalyst Optimizer is often smart enough to optimize these repetitive calls, ensuring that the underlying data processing remains efficient. This structure guarantees that the resulting DataFrame, `df_new`, contains the original columns alongside the newly engineered `location` and `name` fields, ready for further analysis.

Setting Up the Example PySpark DataFrame

To demonstrate the effectiveness of this splitting methodology, we will establish a simple PySpark DataFrame containing concatenated team information. This example simulates a real-world scenario where data ingestion results in combined fields that need to be parsed into meaningful components. The dataset includes team names structured as "City-Nickname" and associated points scored.

The following example details the necessary steps to initialize the Spark session, define the data structure, and create the starting DataFrame. This setup is standard practice when working interactively or developing scripts in a PySpark environment, ensuring the necessary context is available for data transformations.

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

As observed in the output, the `team` column contains the location and nickname joined by a dash (`-`). Suppose our analytical requirement is to analyze points scored by location or nickname independently. This requires us to separate the combined string into two distinct fields, `location` and `name`, using the dash as the key splitting delimiter. This transformation prepares the data for more granular analysis, such as regional performance comparisons or filtering based solely on the team nickname.

Executing the Column Split Operation

Having established the source DataFrame, the next step involves applying the transformation logic detailed earlier. We will utilize the `split` and `withColumn` functions to parse the `team` string

column. Since we anticipate exactly two parts from the split--the location followed by the nickname--we will need two chained `withColumn` calls, targeting indices 0 and 1, respectively.

It is important to ensure that the withColumn method is used efficiently. While we could theoretically split the column once and then reuse the intermediate array column, the idiomatic and often optimized PySpark approach shown below is to apply the split operation directly within the definition of each new column. This maintains a functional programming style and allows Spark to manage the lineage effectively.

We use the following syntax to perform the splitting and view the resulting DataFrame:

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

The execution yields `df_new`, which now clearly shows the original `team` and `points` columns, augmented by the new `location` and `name` columns. Each row demonstrates that the dash separator was successfully used to delineate the two components, which were then extracted based on their positional index (0 for location, 1 for name). This outcome validates the usage of the combined `split` and `getItem` sequence for structured string decomposition in PySpark.

Interpreting the Results and Item Selection

The resulting DataFrame, `df_new`, confirms that the transformation successfully restructured the

data. The strings originally contained in the `team` column, such as "Dallas-Mavs", have been systematically separated, yielding "Dallas" in the `location` column and "Mavs" in the `name` column for that specific record. This separation is entirely dependent on the indexing provided by `getItem()`.

It is essential to reiterate the precise role of the indexing function. The `split` function first transforms the string into an array. For example, "Orlando-Magic" becomes . We then use `getItem(0)` to access the element at the initial position, which is "Orlando", and `getItem(1)` to access the subsequent element, "Magic". If the string contained more segments (e.g., "A-B-C"), the resulting array would be , and we would need `getItem(2)` to access 'C'.

Failure to use `getItem()` after the `split` operation would result in the new column being of type Array, which is generally not suitable for direct analytical use unless array operations are specifically intended. Therefore, the combination of `split()` to tokenize the string and `.getItem(N)` to flatten the array elements into standard string columns is the standard pattern for achieving string splitting into multiple fields in PySpark.

Advanced Considerations for String Splitting

While the basic two-part split is common, real-world data often presents complexities that require more robust handling. One such complexity is dealing with inconsistent delimiters or missing segments. If a row in the `team` column only contained "Lakers" (no dash), the `split` function using the dash delimiter would return an array with a single element: . If we subsequently attempt to access `getItem(1)` for this row, PySpark will return a `null` value for that column segment, which is crucial behavior to anticipate when developing data pipelines.

To mitigate issues arising from uneven splits, developers often employ defensive coding practices. One approach is to first calculate the size of the array resulting from the split using the `size` function (also in `pyspark.sql.functions`). Then, using conditional logic (like `when().otherwise()`), they only attempt to access an index if the array size is large enough, ensuring that missing segments are explicitly handled as `null` or a default value, rather than potentially causing runtime errors in highly complex transformations.

Furthermore, when the delimiter is complex or based on variable whitespace, the `split` function interprets the delimiter argument as a regular expression. This grants significant flexibility, allowing for splitting based on patterns like multiple spaces, special characters requiring escaping (e.g., periods or parentheses), or complex sequences. Always consult the official documentation for the precise regular expression syntax supported by the underlying Spark engine to ensure accurate and reliable splitting operations.

Note that we used the **split** function to split each string, which resulted in two new strings. We then used **getItem(0)** to extract the first string and **getItem(1)** to extract the second string for each team.

Note: You can find the complete documentation for the PySpark **split** function online.

Further Learning and Related PySpark Operations

Mastering string manipulation techniques is foundational to effective data engineering in PySpark. Beyond simply splitting columns, related tasks often involve concatenating data (using `concat` or `concat_ws`), trimming whitespace (using `trim`), or extracting patterns based on complex regular expressions (using `regexp_extract`). These functions collectively enable data professionals to mold raw textual data into structured fields suitable for high-performance distributed computing.

For those interested in optimizing these operations, it is worth exploring how to persist or cache the intermediate DataFrame if the split operation is highly resource-intensive and the resulting split columns are used in many subsequent queries. Caching ensures that the transformation logic is only computed once, speeding up iterative development and complex analysis workflows.

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

Understanding how to use `explode` for array columns.
Techniques for using `regexp_extract` for more complex pattern matching.
Methods for converting column types after a split operation (e.g., string to integer).